Improving the memory-system performance of sparse-matrix vector multiplication

by S. Toledo

Sparse-matrix vector multiplication is an important kernel that often runs inefficiently on superscalar RISC processors. This paper describes techniques that increase instruction-level parallelism and improve performance. The techniques include reordering to reduce cache misses (originally due to Das et al.), blocking to reduce load instructions, and prefetching to prevent multiple load-store units from stalling simultaneously. The techniques improve performance from about 40 MFLOPS (on a well-ordered matrix) to more than 100 MFLOPS on a 266-MFLOPS machine. The techniques are applicable to other superscalar RISC processors as well, and have improved performance on a Sun UltraSPARC™ I workstation, for example.

1. Introduction

Sparse-matrix vector multiplication is an important computational kernel in many iterative linear solvers (see [1], for example). Unfortunately, on many computers this kernel runs slowly relative to other numerical codes such as dense-matrix computations. This paper proposes new techniques for improving the performance of sparse-matrix vector multiplication on superscalar RISC processors. We experimentally analyze these techniques, as well as techniques that have been proposed by others, to show that they can improve performance by more than a factor of 2 on many matrices.

Three main factors contribute to the poor performance of sparse-matrix vector multiplication on modern superscalar RISC processors. First, the lack of data locality causes a large number of cache misses. Typically, accesses to the data structures that represent the sparse matrix **A** have no temporal locality[1] whatsoever, but they have good spatial locality (i.e., there is no data reuse, but accesses are in a stride-1 loop). Accesses to the dense vector **x** being multiplied do reuse data, but the access pattern depends on the sparsity structure of **A**. One technique that can reduce the number of cache misses is to reorder the matrix to reduce the number of cache misses on **x**. This technique was proposed by Das et al. [2], analyzed in certain cases by Temam and Jalby [3], and further investigated by Burgess and Giles [4]. We study this technique further in Section 2, and we also show that the effectiveness of the proposed new techniques depends on it.

[1] We say that a sequence of memory accesses has temporal locality if the same memory locations are accessed repeatedly and the repetitions are close together within the sequence. We say that the sequence has spatial locality if adjacent memory locations are accessed close together.

0018-8646/97/$5.00 © 1997 IBM

```
do i = 1,n
  sum = 0.0D0
  do j = rowptr(i), rowptr(i+1)-1
    sum = sum + a(jp) * x(colind(jp))
  end do
  y(i) = sum
end do
```

Figure 1

A sparse-matrix vector multiplication code for matrices stored in a compressed-row format. The N nonzeros of the n-by-n matrix **A** are compressed into a single vector a in a row-wise ordering; the column indices of these nonzeros are compressed into an integer vector colind. The vector rowptr stores the first index of each row of **A** in the vectors a and colind, and its last element contains $N + 1$.

A second factor that limits performance is the tendency of multiple load/store functional units to miss on the same cache line. Many superscalar RISC processors have at least two load/store units. On such processors, when one unit is stalled because of a cache miss, the other unit(s) can continue to load data from the cache. Unfortunately, in stride-1 loops the other units soon try to load data from the cache line that caused the first miss. Consequently, all units are often stalled on the same cache line. The miss is compulsory because the accesses have no temporal locality, so one unit must spend time waiting for the miss to be serviced. However, the misses generated by the other units are not compulsory, and we show in Section 2 how to prevent them by using prefetching. These cache misses can also be prevented by a simple hardware-assisted prefetching mechanism, which is described in Section 4.

Finally, sparse-matrix vector multiplication codes typically perform a large number of load instructions relative to the number of floating-point operations they perform. This phenomenon occurs because of poor data locality, which makes it difficult to reuse data already in registers, and also because the code must load row or column indices in addition to floating-point data. The large number of load instructions places a heavy load on the load/store units that serve as the interface between the register files and the cache, and on the integer ALUs that compute the addresses to be loaded. (These ALUs are sometimes part of the load/store units and sometimes part of the integer execution units.) On most current processors, these units are often the bottleneck in sparse-matrix vector multiplication. The floating-point units are therefore underutilized. We present in Section 2 two techniques that address this issue; the first is a blocking technique that reduces the number of load instructions, and the second is a technique that eliminates indexing instructions. Blocking in sparse-matrix vector multiplication was used in somewhat different forms in [5, 6].

Although the techniques that we propose can be applied separately, they are most effective when combined. In particular, reordering the matrix can enhance or degrade the effect of blocking. Also, without the reduction in the number of cache misses on **x** that reordering yields, our prefetching technique is ineffective on large matrices.

Our techniques are also applicable to other superscalar RISC processors. We describe the techniques in the next section and comment, where appropriate, on their applicability to other current superscalar processors. One of the techniques, prefetching, is difficult to implement in the irregular loops that comprise the sparse-matrix vector multiplication code. Section 3 explains how we implemented this technique.

We have implemented our techniques and evaluated them on an IBM superscalar RISC workstation with a POWER2* processor using a suite of 13 matrices. The matrices are all structurally symmetric, but the code is general and does not exploit symmetry. (A matrix vector code for symmetric matrices can load fewer coefficients and therefore run more efficiently.) Section 5 presents our experimental results. The results show that the techniques are viable and that they significantly enhance performance. Our conclusions from this research are presented in Section 6.

2. Algorithmic techniques

Consider the typical sparse-matrix vector multiplication code shown in **Figure 1.** The code assumes that the matrix is stored in a *compressed-row* format, but the same considerations apply to other storage formats that support general sparsity patterns. The inner loop of the code loads `a(jp)`, `colind(jp)`, and `x(colind(jp))`, and performs one multiply–add operation. While `a` and `colind` are loaded using a stride-1 access pattern, `x(colind(jp))` may be any element of `x`.

There are four potential performance problems in this code. The accesses to `a` and `colind` generate many cache misses, one per cache line (because the stride-1 access ensures that the entire line is used before it is evicted from the cache). Depending on the number of nonzeros in **A** and on the details of the iterative algorithm in which the matrix vector multiplication is used, these cache misses can occur in the first-level cache or in a cache further away from the processor. The accesses to `x` can have poor spatial and temporal locality, and can hence generate even more cache misses. The ratio of data values loaded into registers per floating-point operation is 1.5, which means

that the code's performance is limited by the performance of the processor's load/store units. Finally, the conversion of `colind(jp)` from an integer index to a byte offset from the beginning of `x`, required on most processors for indirect addressing, requires the integer ALUs to perform an additional instruction in every iteration. This section describes four techniques that can cope with these problems.

• *Reducing cache misses through bandwidth reduction*
The bandwidth of a sparse matrix is the maximum distance, in diagonals, between two nonzero elements of the matrix. Matrix-reordering algorithms that reduce the bandwidth of a matrix have been proposed since the late 1960s as a way to reduce fill and work in sparse-matrix factorizations. The first such technique, which is based on a breadth-first traversal of the graph underlying the matrix, was invented by Cuthill and McKee [7] (see [8] for a contemporary description). A simple modification of their technique, which reverses the ordering produced by the Cuthill–McKee algorithm, was later found to be even more efficient in sparse factorizations.

Das et al. [2] proposed the reordering of sparse matrices using a bandwidth-reducing technique in order to reduce the number of cache misses generated by accesses to `x`. Temam and Jalby [3] analyzed the number of cache misses as a function of the bandwidth for certain cache configurations. The technique was investigated further by Burgess and Giles [4], who extended it to other unstructured-grid computations. Burgess and Giles experimentally studied several reordering strategies, including reverse Cuthill–McKee and a greedy blocking method. They found that reordering improved performance relative to a random ordering, but they did not find any sensitivity to the particular ordering method used.

We have performed additional experiments with a larger set of matrices. Our experiments, described in detail in Section 5, essentially validate the results of Burgess and Giles. We have found that compared to a random ordering, bandwidth reduction and nested-dissection orderings reduce cache misses and improve performance. Performance can improve by more than a factor of 3 on large matrices.

When the bandwidth-reduction technique is used alone without blocking or prefetching, the particular ordering method does not matter much to the performance of matrix vector multiplication. Consequently, reordering methods should be selected on the basis of the reordering speed and the effect on ordering-sensitive preconditioners, such as incomplete LU. When the bandwidth-reduction technique is combined with blocking and prefetching, however, different ordering methods yield different performance results. We have found that in this situation Cuthill–McKee ordering is usually the best choice. It is a fast algorithm and has benefits in preconditioning as well, as explained in Section 6.

Because sparse-matrix vector multiplication uses large data structures and has poor data locality, reducing cache misses is an important goal for this code, even on processors in which out-of-order execution can cover the cache-miss latency in other codes.

• *Reducing the number of load instructions through blocking*
We reduce the number of load instructions the code performs by splitting the general sparse matrix **A** into a sum of two or three matrices, some of which are block matrices. It is reasonable to expect the matrix to contain dense blocks, because such matrices arise in many application areas (for example, equations defined on grids with more than one variable per grid point). Multiplying a block sparse matrix by a vector can reduce the number of loads over unblocked multiplication in three different ways. First, blocking enables the code to load fewer row or column indices into registers, because only one index per block is required, rather than one per nonzero. Second, elements of `x` are loaded once and used k times when the matrix is blocked into k-by-l blocks (this is a form of register blocking). Third, since the POWER2 processor has a load-quad instruction that loads two consecutive floating-point doublewords into two registers, 1-by-2 blocks allow the processor to load the two nonzeros in `a` and the two elements of `x` using only two, rather than four, load instructions.

We therefore scan the matrix in a preprocessing phase and extract all the 2-by-2 blocks that we find, and then all the 1-by-2 blocks that we find. The extraction is done in a "greedy" fashion that extracts all the blocks that are found in a row-wise scanning of the matrix. For 2-by-2 blocks, this greedy algorithm is not always optimal, in the sense that it may find fewer 2-by-2 blocks than possible. The greedy algorithm is optimal for 1-by-2 blocks. To preserve data locality in accesses to `x`, we do not multiply by the 2-by-2 blocks, then by the 1-by-2 blocks, and then by the remaining unblocked nonzeros. Instead, we process row after row, where in each row i we perform both the multiplication by the unblocked nonzeros in row i and the multiplication by the blocked nonzeros in rows i and $i + 1$.

On processors without a quad-load instruction (practically all other current RISC processors), some of the benefit of 1-by-2 blocks can be realized by replacing them with 2-by-1 blocks. These require only three floating-point loads, because the element of `x` can be loaded once and used twice.

The balance between the capabilities of the load/store units and the floating-point units in other superscalar

processors is the same as or worse than the balance in the POWER2 processor[2]. The POWER2 processor can perform four FLOPS (by issuing multiply–add instructions) and load four 64-bit words (by issuing two quad-load instructions) in every cycle. Digital's Alpha 21164 [9] and Sun's UltraSPARC** [10] both have floating-point units that can issue one add and one multiply instruction every cycle. The Alpha can issue two load instructions and load two words, giving the same balance as the POWER2, but the UltraSPARC can issue only one load instruction per cycle, giving a worse balance. We therefore conclude that reducing the number of loads is at least as important on other processors as it is on the POWER2.

Our approach to blocking is novel in that our algorithm attempts to find many small, completely dense blocks. Other researchers have proposed algorithms that attempt to find larger (and hence fewer) dense blocks and/or blocks that are not completely dense.

Agarwal, Gustavson, and Zubair [5] describe an algorithm designed for vector processors. Their algorithm tries to find a few large, relatively dense blocks. They divide the rows of the matrix into blocks, and attempt to find one fairly dense rectangular block in every block of rows. The other nonzeros in the block of rows remain unblocked (they may be coalesced into dense diagonals, however). This scheme works well on vector processors in which the vector startup cost is significant.

On the other hand, we realized that on RISC processors small blocks would suffice to obtain good performance, and that the most important objective is to block as many of the nonzeros as possible, even if the blocks are small. In addition, the small vector startup cost on RISC machines implies that it is probably not beneficial to use dense blocks that contain many zeros. The extra time it takes to load these structural zeros into cache and into registers is likely to mask any performance benefit that larger dense blocks might afford. (A simple analysis shows, however, that allowing one zero in a 2-by-2 block might improve performance slightly on the POWER2 processor.)

PETSc, a toolkit for scientific computations [6], uses a blocked sparse-matrix vector multiplication subroutine that attempts to find blocks of rows with the same nonzero structure. Our approach is therefore similar to that of PETSc in that both approaches require the blocks to be completely dense, but there is an important difference. Our algorithm can block most of the nonzeros in a matrix even when no two rows have identical structure, whereas PETSc cannot. The penalty that our algorithm pays for the extra flexibility is quite small: Our 2-by-2 blocked algorithm loads only one column index per six floating-point words loaded. In addition, PETSc allows different numbers of rows in different blocks, which can cause more run-time overhead than fixed-size blocks.

- *Prefetching in irregular loops*

Traditionally, prefetching has been considered to be a technique for hiding latency, in the sense that prefetching can prevent memory-access latency from degrading performance, as long as memory bandwidth is sufficient to keep the processing units busy (see [11–13], for example). In many codes, for example dense-matrix vector multiplication, the ratio of floating-point to load instructions is quite high. This high ratio allows the algorithm to hide the latency of cache misses by prefetching cache lines early. The fact that the load/store unit is stalled for many cycles is negligible, because there are only a few load instructions relative to floating-point instructions.

In matrix vector multiplication, especially with sparse matrices, the ratio of floating-point to load instructions is low, less than one. The memory bandwidth required to keep the processing units busy is therefore high. Since most computers do not have sufficient memory bandwidth, loading data from memory becomes the bottleneck that determines performance in this computation. It is not important here whether the load/store unit stalls early (with prefetching) or late (without), since it is needed all the time.

We can still use prefetching, however, not to hide latency but to improve memory bandwidth. As far as we know, this use of prefetching is novel. In particular, we use prefetching to prevent multiple load/store units from stalling on the same cache line. Since most of the cache misses are generated by accesses to two vectors that are accessed in the same stride-1 loop, multiple load/store units can miss on the same cache line. One unit misses on the first word in a line and stalls. The second unit tries to load the next word in the vector and also stalls. This can happen even when two vectors are accessed, if the second unit loads a word in cache from the second vector and then misses on the first. When two units stall on the same line, the effective memory-to-cache and cache-to-register bandwidths are reduced. It is possible, using prefetching, to avoid this behavior.

Our strategy is to prefetch elements of `a` and `colind` before they are used. The prefetching instruction always misses and hence stalls one of the load/store units. The other unit, however, continues without stalling as long as there are no cache misses on `x` or `y`, which are rare once the matrix has been reordered. The details are quite complicated, however, and are explained in Section 3.

The use of prefetching to prevent multiple load/store units from stalling on the same cache line is likely to have

[2] By balance we refer to number of data values that can be loaded in a cycle relative to the number of floating-point operations that can be performed in a cycle.

a beneficial effect on other RISC processors that have multiple load/store units.

- *Eliminating integer-to-pointer conversions*

The expression `x(colind(jp))` is compiled into an instruction sequence that loads `colind(jp)` into a register and at least two more load instructions that load `x(colind(jp))`. The expression `colind(jp)` can be loaded in one instruction, since `colind` is accessed in a stride-1 loop, by an instruction that loads it and increments the pointer to `colind(jp)` by the size of one integer. Loading `x(colind(jp))` requires two instructions, because the instruction that loads a word with a given offset from the beginning of `x` requires a byte offset, not a word offset. The expression `colind(jp)` must therefore be multiplied by the size of a word in bytes to yield a byte offset.

This multiplication is done by an arithmetic shift instruction that executes in one cycle on one of the integer ALUs. On processors where the integer ALUs also compute the addresses for load instructions, such as the IBM POWER2 and the Digital Alpha 21164, this extra integer instruction places additional overhead on the fixed-point units, which are already overloaded with load instructions[3].

We can avoid this overhead by performing a preprocessing phase in which we replace the integer indices with pointers to elements of `x`. We show in Section 5 that this optimization alone can improve performance by as much as 38%. This optimization was also used in the IBM implementation of the NAS benchmarks [14].

This optimization amounts to removing a transformation on the column-indices vector out of the inner loop of Figure 1. This optimization can, in principle at least, be performed by a compiler that is capable of interprocedural analysis. Such a compiler optimization may be possible only if the iterative algorithm and the matrix vector multiplication kernel are compiled at the same time. It also requires the compiler either to recognize that the column indices are not used elsewhere and can be overwritten by pointers, or to allocate a large temporary array to store the pointers.

On processors with a dedicated virtual-address adder, such as Sun's UltraSPARC, this optimization is likely to have no effect at all because the load and shift instructions do not compete for the same functional units. Experiments on an UltraSPARC workstation, described in Section 5, verify that this optimization has no effect on the performance of the code on that processor.

[3] One of the referees of this paper regards this behavior of the POWER2 Architecture* as an architectural flaw: "Since the floating-point load instructions specify the data length (single, double, etc.) in the instruction format, the instruction unit can pass this information to the addressing unit, which should handle the shift implicitly. This, in fact, was done on the IBM 370/390 Vector Facility."

3. Prefetching in an irregular loop

To implement our prefetching strategy, we need to perform a single prefetch instruction after a certain number of iterations of the inner loop in Figure 1, or in its blocked counterparts. We assume that cache lines on our target machine contain 32 doublewords or 64 integers. Ideally, we would prefetch one cache line of `a` every 32 iterations and one cache line of `colind` every 64 iterations. A simpler scheme is to prefetch both vectors every 32 iterations. The prefetch in iteration `jp` would be for `a(jp+64)` (two cache lines ahead), and the prefetch in iteration `jp' = jp + 16` would be for `colind(jp'+128)` (also two cache lines ahead). The constants depend on the cache configuration, but we use these numbers here for illustration.

It is difficult to implement such a precise strategy, or even the simpler one, without introducing extra overhead into the inner loop. The problem is that the number of iterations that the inner loop performs depends on the length of row `i`, which is usually different for different rows. Sorting rows by length is not a viable option, because it conflicts with ordering rows and columns for data locality. Padding rows with zeros so that their length is divisible by 16 is also not attractive, because computing on these padding zeros can constitute a significant overhead for very sparse matrices.

To overcome this problem, we use an approximation. The main idea is to unroll the inner loop, say 32 times, and to place one prefetching instruction for `a` before the first unrolled instance and another for `colind` after 16 instances. If rows are very long, this would implement our simple prefetching strategy. This approximation does not implement our strategy of prefetching every 32 iterations when rows are short. If the row ends shortly after a prefetch for `a`, the prefetching at the beginning of the next row usually stalls the second load/store unit on the same cache line. When rows are shorter than 16 elements, `colind` is not prefetched. Nevertheless, our experiments, shown in Section 5, demonstrate that the technique is effective.

To preserve the semantics of the original loop, we compare `jp` to `rowptr(i+1)` after every unrolled instance and skip the rest of the instances if `jp` is greater. This ensures that we do not step beyond the end of a sparse row as a result of the unrolling.

Even this approximation to precise algorithmic prefetching is difficult to implement in a high-level language like C or FORTRAN. The difficulty revolves around the need to compare `jp` to `rowptr(i+1)` and conditionally branch without slowing down the iterations. Our solution is to compile the algorithm, which is written in C, into assembly language, then modify the assembly language and compile it into object code. We perform the

unrolling, the insertion of prefetching instructions, and the conditional branching in assembly language. Working in assembly language allows us to exploit the branch-on-counter instruction and to unroll this irregular loop without any overhead. We have been able to perform these modifications in assembly language in about one day on both an IBM POWER2 workstation and a Sun UltraSPARC workstation.

4. Hardware-assisted prefetching

We propose a simple hardware-assisted prefetching mechanism that can handle prefetching in both regular and irregular loops. We propose two new instructions called `prefetch-start` and `prefetch-stop`. Both instructions serve as hints to the processor, just like regular prefetch instructions, and both take one argument, the name r of an integer register. The `prefetch-start` instruction indicates to the processor that the register r serves as a pointer in a stride-1 loop. Therefore, the hardware should attempt to prefetch cache lines that are ahead of the current value of r. The `prefetch-stop` instruction indicates that the register r is no longer used as a pointer in a loop, and that prefetching based on its content should stop. Prefetching in iteration structures more complex than stride-1 loops can perhaps be supported by variants of the `prefetch-start` instruction.

One possible implementation is to generate a signal when an addition to r generates a carry from bit k to bit $k + 1$ in r, where cache lines are 2^k bytes wide. This signal is then used to trigger prefetching for a cache line ahead of the current value of r. This scheme ensures that the prefetching signal is generated when a small increment (at most a cache line) to r causes it to point to a new cache line.

This form of hardware-assisted prefetching has several advantages over regular prefetching instructions that are inserted by the compiler, or even by hand. Hardware-assisted prefetching works equally well in regular and irregular loops, because it eliminates the need to prefetch every fixed number of inner iterations. Hardware-assisted prefetching enables precise prefetching using a single machine-code binary, even when that binary is executed on machines with several cache-line sizes. For example, machines with IBM POWER2 processors come in at least three primary cache configurations, which require different prefetching rates because they have different cache-line sizes.

5. Experimental results

In this section we present the results of the extensive experiments we carried out in order to assess the effectiveness of our strategy. Most of the experiments were performed on a superscalar IBM RS/6000* workstation. We repeated some of the experiments on a Sun UltraSPARC workstation in order to determine how general we can expect the techniques to be. We believe that most of our findings would also apply to other RISC processors.

- *Experimental platform*

The experiments were carried out on an RS/6000 workstation with a 66.5-MHz POWER2 processor [15], a 256KB, four-way set-associative data cache, a 256-bit-wide memory bus, and 512 megabytes of main memory. The POWER2 processor has 32 architected and 54 physical floating-point registers, two floating-point units, two integer units that also serve as load/store units, and a branch unit. The floating-point units are each capable of executing one multiply–add instruction in every cycle for a peak performance of 266 million floating-point operations per second (MFLOPS). For data in the cache, the integer units are each capable of loading or storing in one cycle one integer register, one floating-point register, or even two floating-point registers using a so-called quad-load or quad-store instruction. The cache is capable of servicing hits under a miss, so that one integer unit can continue to load and store data to and from the cache while the other is waiting for a cache miss to be serviced. The separate branch unit enables the processor to perform many branch instructions without stalling the other execution units at all. These so-called *zero-cycle branches* include virtually all unconditional branches and branch-on-counters, the kind of branch instruction used in the innermost loop of a set of nested FORTRAN `do`-loops.

To put our results in perspective, we note that on this machine the dense-matrix multiplication subroutine (DGEMM) in the IBM Engineering and Scientific Subroutine Library (ESSL) achieves performance of about 250 MFLOPS when applied to matrices that do not fit in the cache. The performance of the ESSL dense-matrix vector multiplication subroutine (DGEMV) on this machine is about 170 MFLOPS when applied to large matrices.

The performance of the algorithms was assessed using measurements of both running time and cache misses. Time was measured using the machine's real-time clock, which has a resolution of one cycle. The number of cache misses was measured using the POWER2 performance monitor [16]. The performance monitor is a hardware subsystem in the processor capable of counting cache misses and other processor events. Both the real-time clock and the performance monitor are oblivious to time sharing. To minimize the risk that measurements may be influenced by other processes, we ran the experiments when no other users used the machine (but it was connected to the network). We later verified that the measurements were valid by comparing the real-time-clock

Table 1 Characteristics of the test-suite matrices. The Boeing matrices are all stiffness structural engineering matrices from Roger Grimes of Boeing, and the NasGraph matrices are from the 1994 partitioning benchmarks of NASA's Numerical Aerodynamics Simulations Department.

Matrix	*Dimension*	*Nonzeros*	*Source*	*Model*
bcsstk32	44609	2014701	Boeing	Model of an automobile chassis
msc10848	10848	1229778	Boeing	Test matrix KNUCKLEF8.OUT2 from MSC/NASTRAN†
msc23052	23052	1154814	Boeing	Test matrix SHOCKF8.OUT2 from MSC/NASTRAN
ct20stif	52329	2698463	Boeing	CT20 engine block
crystk02	13965	968583	Boeing	FEM crystal-free vibration
crystk03	24696	1751178	Boeing	FEM crystal-free vibration
bcsstk35	30237	1450163	Boeing	Automobile seat frame and body attachment
bcsstk36	23052	1143140	Boeing	Automobile shock absorber assembly
bcsstk37	25503	1140977	Boeing	Track ball
cojack	188384	875446	NasGraph	Cooling water jacket of a BMW engine
hsctl	31736	253816	NasGraph	Large model of a high-speed civil transport
pwt	36519	253069	NasGraph	Unstructured grid
rock1	133904	214086	NasGraph	Large model of a rock

measurements with the user time reported by the `getrusage` system call on an experiment-by-experiment basis. All measurements reported here (except ordering and blocking times) are based on an average of ten executions.

We coded most of the algorithms in C and compiled them using the IBM `xlc` C compiler Version 1.3. We used compiler options `-O3` and `-qarch=pwr2`, which cause the compiler to optimize the code and to utilize instructions specific to the POWER2 processor, most significantly quad-loads. Under these optimization options, the compiler unrolls the inner loop so that it usually contains four multiply–adds. We implemented prefetching by modifying the compiler's assembly-language output. Specifically, we unrolled the inner loops further by hand by a factor of 8 and placed one prefetching instruction in the beginning of this unrolled loop for an element of `a` and another in the middle of the loop, after 16 multiply– adds, for an element of the pointer vector `colind`.

• *Methodology and test matrices*

The algorithms were tested on a suite of 13 sparse, symmetric stiffness matrices. Nine of the matrices are structural engineering matrices from Boeing, which were donated by Roger Grimes to the Harwell–Boeing matrix collection or to Tim Davis's matrix collection.[4] The four other matrices are from the 1994 partitioning benchmarks of NASA's Numerical Aerodynamics Simulations Department. The characteristics of the test matrices are described in **Table 1**. We use structurally symmetric matrices from two sources. Matrices that were contributed by Grimes represent three-dimensional structures with several degrees of freedom (variables) per grid point, typically at least three. They seem to have been ordered using a band- or profile-ordering algorithm applied to the original grid, so that all of the degrees of freedom associated with a single grid point remain contiguous. Matrices from the NasGraph collection also represent three-dimensional models, but have only one degree of freedom per grid point. As a consequence, they are sparser and do not have contiguous dense blocks. We do not know how they were ordered before they were placed in the matrix collection.

For each matrix, we measured the performance of all combinations of ten different matrix vector multiplication codes and five different orderings. The ten codes are described in **Table 2**. The five orderings are random ordering, a nested-dissection-type ordering (denoted by WGPP), reverse Cuthill–McKee (RCM), Cuthill–McKee (CM), and the original ordering of the matrix as stored in the matrix collection. The WGPP (Watson Graph-Partitioning and Sparse-Matrix-Ordering Package) ordering code was written by Gupta [17, 18].

In each experiment we measured the running time of the matrix vector multiplication code, the number of load instructions of various kinds it performed, and the number of cache and TLB misses. Each measurement is an

[4] Roger Grimes is with Boeing Computer Services, Seattle, Washington; Tim Davis is with the University of Florida, *http://w.w.w.cise.ufl.edu/~davis.*

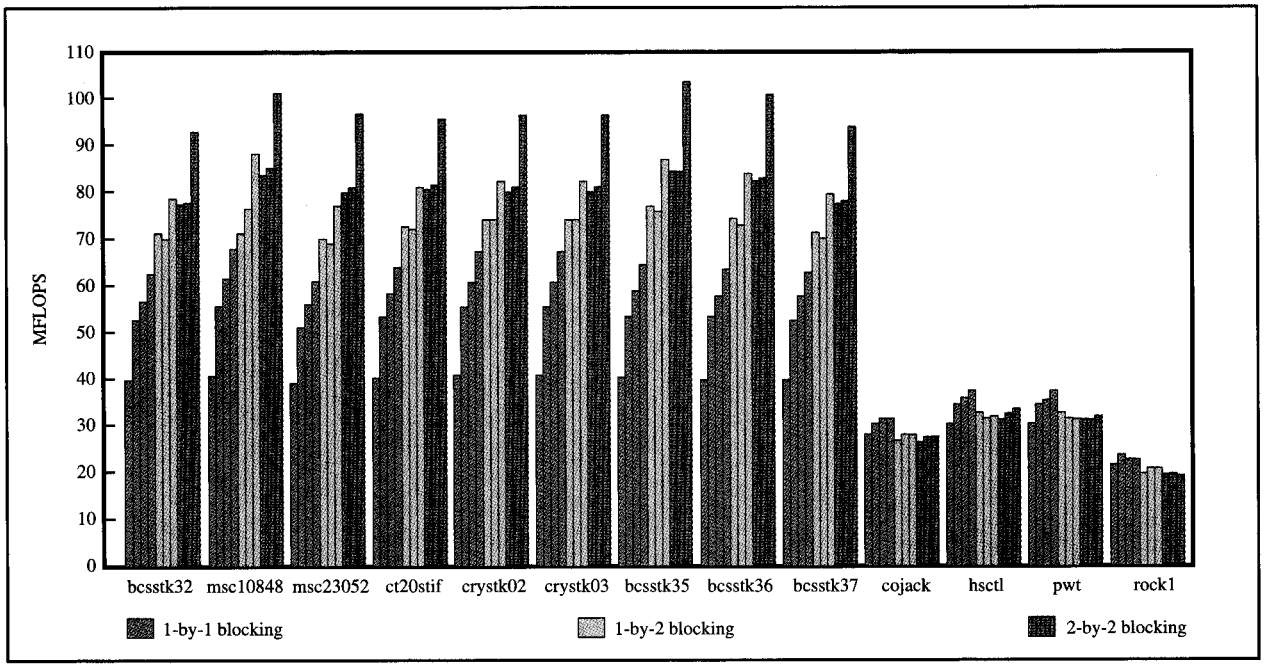

Figure 2

The performance in millions of floating-point operations per second (MFLOPS) of the sparse-matrix vector multiplication codes. The performance on each test matrix is represented by 10 bars, one for each code. The 10 bars are organized into three groups, one for 1-by-1 blocking (i.e., no blocking), one for 1-by-2 blocking, and one for 2-by-2 and 1-by-2 blocking. Each group is represented by a different color. The leftmost bar with no blocking represents the performance of C code with integer indices. The next three bars, as well as the three bars in the other two groups, represent the performance of C code with pointer indices, assembly language code with pointer indices, and assembly language code with pointer indices and prefetching. The original orderings of the matrices in the matrix collections are used.

Table 2 Characteristics of the matrix vector multiplication codes.

No.	*Mnemonic*	*Language*	*Indices*	*Blocking*	*Prefetching*
1	C-I-1×1-N	C	Integers	No	No
2	C-P-1×1-N	C	Pointers	No	No
3	A-P-1×1-N	Assembly	Pointers	No	No
4	A-P-1×1-P	Assembly	Pointers	No	Yes
5	C-P-1×2-N	C	Pointers	1-by-2	No
6	A-P-1×2-N	Assembly	Pointers	1-by-2	No
7	A-P-1×2-P	Assembly	Pointers	1-by-2	Yes
8	C-P-2×2-N	C	Pointers	2-by-2 and 1-by-2	No
9	A-P-2×2-N	Assembly	Pointers	2-by-2 and 1-by-2	No
10	A-P-2×2-P	Assembly	Pointers	2-by-2 and 1-by-2	Yes

average over ten multiplications. The first ten multiplications were not used at all, in order to eliminate any influences of cold starts that are usually not significant in iterative algorithms. All of the matrices were too large to remain in the cache between multiplications. Since the cache can hold 32 768 doublewords, in some of the experiments the vector x could fit within the cache and remain there between multiplications.

To assess the cost of reordering and blocking the matrices relative to the potential benefits, we recorded the running time of the ordering and blocking algorithm we used to preprocess the matrices. These measurements are of single runs.

- *Effects of blocking and prefetching*

Figure 2 shows the performance of the ten codes. The matrices are all ordered in the original ordering from the

matrix collections, which proved to be the best or close to the best ordering for all of them (see below for more details on the effect of ordering). The most striking feature that emerges from the figure is that the behavior of the nine Boeing matrices is very different from the behavior of the four NasGraph matrices. This difference, which is mostly due to differences in the sparseness of the matrices, is explored below in more detail.

Each of the algorithmic improvements we have introduced boosts performance on the Boeing matrices. Replacing integer indices with pointers increases performance from about 40 MFLOPS to about 52–55 MFLOPS. With no blocking, the extra unrolling in the assembly-language code improves performance by 5 to 6 additional MFLOPS. Prefetching improves performance by another 5 to 6 MFLOPS. With blocking, there is essentially no difference between C and assembly-language versions. The 1-by-2 blocking with no prefetching improves performance to 70–77 MFLOPS, and prefetching combined with 1-by-2 blocking boosts performance to 78–98 MFLOPS. The 2-by-2 blocking with no prefetching gives similar performance. Finally, 2-by-2 blocking combined with prefetching yields performance of 93–104 MFLOPS.

On the NasGraph matrices, using pointers, unrolling, and prefetching all help, but blocking does not improve performance. In fact, in most cases blocking degrades performance. On these matrices, the best-performing code yields a performance of only 23–39 MFLOPS, and the differences between the different codes are smaller than the differences in the Boeing matrices. Since the NasGraph matrices are much sparser than the Boeing matrices, blocking the multiplication code introduces overhead without delivering a significant benefit. **Figure 3** shows the correlation between the sparseness of the matrices and the benefit yielded by blocking and prefetching. The most probable reason that the NasGraph matrices do not benefit from blocking is that they represent models with only one degree of freedom per grid point. As a consequence, the underlying graphs have only a few cliques, so the number of dense blocks is small no matter how the matrices are ordered.

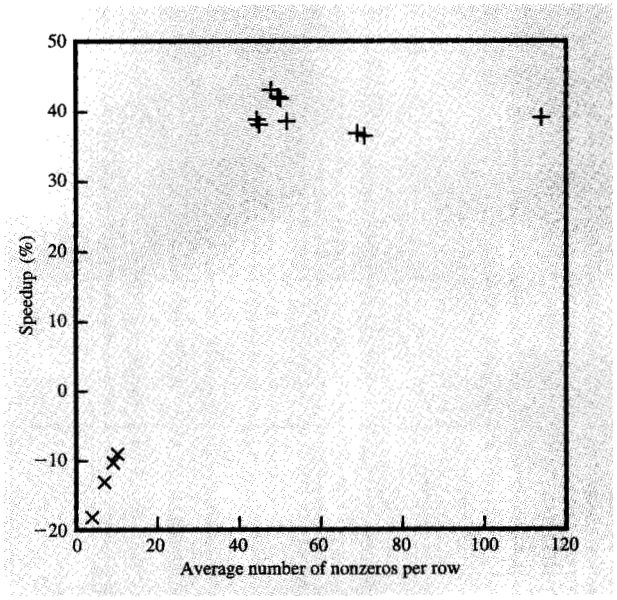

Figure 3

Scatter plot of the percentage speedup in running time due to blocking and prefetching versus the average density of the matrix. Each mark represents one matrix, with Boeing matrices represented by + and NasGraph matrices by ×. The speedup is computed as $(T_1 - T_2)/T_1$, where T_1 is the running time with no blocking and no prefetching, and T_2 is the running time with blocking and prefetching. The original ordering of the matrices is used.

- *Effects of ordering*

Figure 4 shows the effect of reordering matrices on the matrix vector multiplication codes. In all cases the performance on ordered matrices is better than on randomly permuted matrices. The differences are especially large on large matrices, such as bcsstk32, ct20stif, cojack, and rock1. **Figure 5** shows the correlation between the order of the matrix and the improvement in performance due to ordering. Since the plot indicates that the Boeing matrices benefit more from ordering than the NasGraph matrices, we hypothesize that the benefit of ordering depends both on the order of the matrix and on its density.

Without blocking, the various ordering methods yield roughly the same performance. This level of performance is similar to the level that is achieved by the original ordering of the test matrices. When blocking is used, the performance on the NasGraph matrices degrades slightly, but the performance on the Boeing matrices improves. With blocking, different ordering methods yield different performance levels. The random ordering is still always worst, followed by the RCM ordering, the WGPP ordering, the CM ordering, and finally the original ordering.

Figure 6 explains the variations in performance of the blocked codes with various ordering methods. We analyze mostly the denser Boeing matrices, because the performance impact of blocking on the sparser NasGraph matrices is marginal. The figure shows that the random and reverse-Cuthill–McKee orderings result in matrices with no or very few 2-by-2 and 1-by-2 blocks. Therefore, using a blocked code with a randomly permuted matrix or a matrix which has been RCM-ordered does not improve

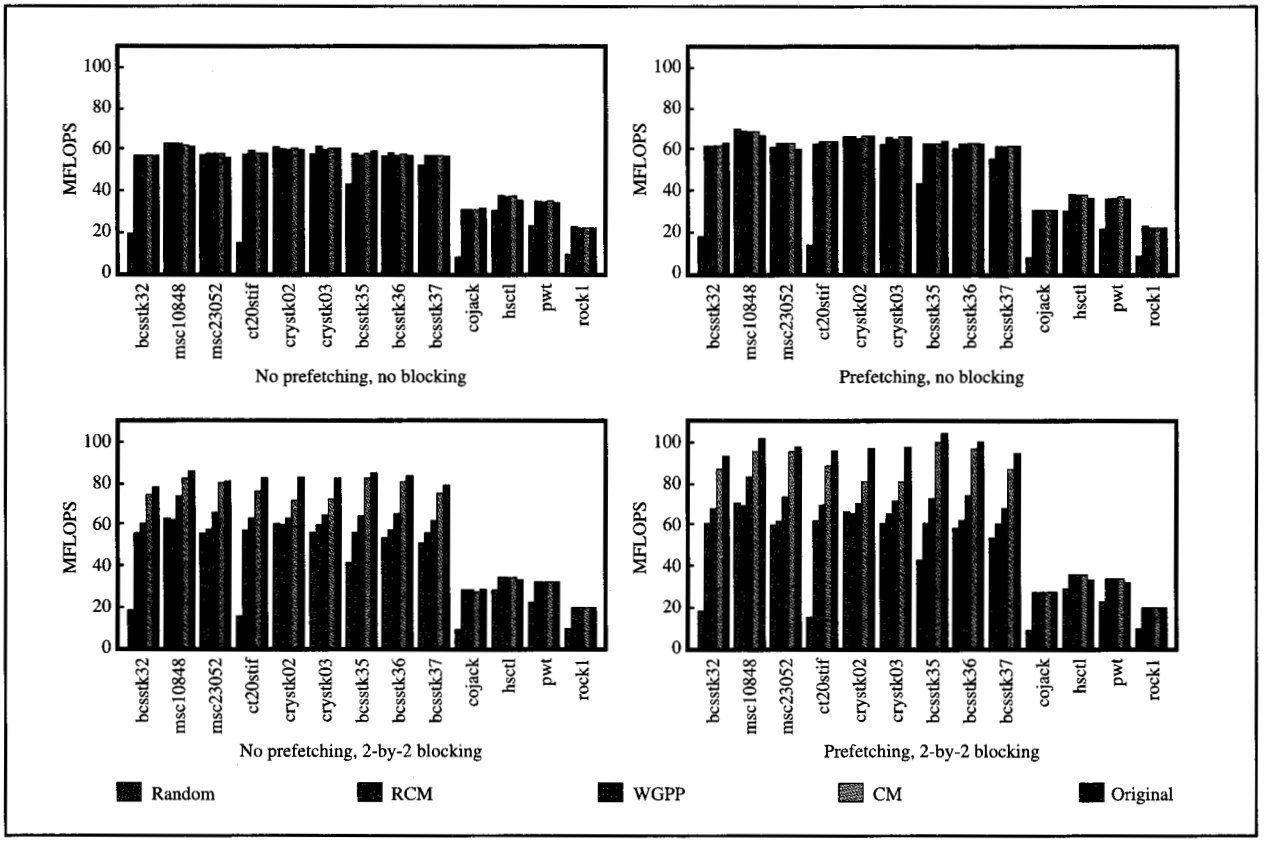

Figure 4

Performance in MFLOPS of four assembly-language matrix vector multiplication codes. The top two depict the performance with no blocking and the bottom two with both 2-by-2 and 1-by-2 blocking. The graphs on the left do not use prefetching; the graphs on the right do. Each graph shows the performance on each matrix, using five different orderings represented by different colors.

performance. The WGPP ordering creates more blocks. In the Boeing matrices, 44–70% of the nonzeros are in blocks; most of them are usually in the smaller 1-by-2 blocks. The Cuthill–McKee orderings result in many more nonzeros in blocks, between 73 and 97%. The fraction that is in 2-by-2 blocks is better than with the WGPP ordering, but it is still sometimes less than one half. In the original ordering, even more nonzeros are in blocks, and the vast majority of them are in 2-by-2 blocks.

The differences between the fractions of nonzero blocks in the different orderings can be traced to several factors. The original ordering yields better blocking than all of the other orderings, probably since it was applied to grid points rather than individual degrees of freedom (variables). Consequently, dense blocks in the matrix that was produced by the grid generator remained dense. When we applied the other orderings to the matrices, some of these dense blocks disappeared. The differences between the Cuthill–McKee and the reverse-Cuthill–McKee may be due to the fact that our blocking algorithm is greedy and scans the matrix from top to bottom and from left to right within rows.

- *Cost of reordering and blocking*

Figure 7 shows the time it takes to reorder the randomly permuted test matrices. The time is shown relative to the basic matrix vector multiplication time (`C-I-1x1-N`) with a randomly permuted matrix. The Cuthill–McKee and reverse Cuthill–McKee orderings take one to three times longer than matrix vector multiplication; therefore, they are well worth their cost. The WGPP ordering costs 20 to 200 times more than matrix vector multiplication. (This ordering code was designed as a fill-reducing mechanism for direct factorizations, which are much more expensive than single matrix vector multiplications.) It is therefore not appropriate for our application.

Figure 7 also shows the time it takes to block the reordered test matrices into 2-by-2 and 1-by-2 blocks.

The time is shown relative to the basic matrix vector multiplication time (`C-I-1x1-N`) with a randomly permuted matrix. The blocking usually costs 4 to 15 times more than basic matrix vector multiplications. The graph shows that in several experiments blocking took significantly more time than that because of paging. The fact that this phenomenon occurs only when blocking follows an ordering step (in which additional memory is allocated) shows that the problem is indeed paging and memory management.

We now estimate the number of matrix vector multiplications that must be performed following a blocking step to pay for the cost of blocking. We assume that blocking costs 15 times more than basic matrix vector multiplication, and about 25 times more than the best unblocked code (`A±P±1x1±P` with prefetching). We also assume that blocking reduces the matrix vector multiplication time by one third, which is a conservative estimate for the Boeing matrices. From these assumptions it follows that if the matrix is used in more than 75 multiplications, blocking will reduce the overall running time.

• *Comparison to a direct solver*

To put our results in a broader perspective, we compare them to the performance of a multifrontal, sparse direct solver. The solver was written by Anshul Gupta, who provided us with the timings below. Directly solving a linear system $Ax = b$, where A is our test matrix bcsstk32 with one right-hand side, took 15.7 seconds (on the same machine used for the rest of the experiments). Of the 15.7 seconds, ordering the matrix took 4.5 seconds, symbolic factorization took 1.5 seconds, numerical factorization took 9.3 seconds, and the triangular solve took 0.36 seconds. The number of floating-point operations in the factorization was 1630 million, giving a computational rate of about 175 MFLOPS for the numerical factorization alone and 104 MFLOPS for the entire solution.

In comparison, our best matrix vector multiplication code took 0.0433 seconds on the same matrix. The cost of the numerical factorization is therefore equivalent to that of about 215 matrix vector multiplications. The total cost of the direct solution, 15.7 seconds, is equivalent to 1.7 seconds for reordering and blocking plus 323 matrix vector multiplications.

• *Portability experiments*

We have repeated some of the experiments reported above on another superscalar RISC computer, a Sun UltraSPARC workstation. The machine has a 143-MHz UltraSPARC I processor with a 16KB direct-mapped primary data cache, a secondary off-chip 512KB cache, a 288-bit-wide memory bus, and 96 megabytes of memory. The UltraSPARC I processor has 32 floating-point registers and can issue one floating-point multiply and one floating-point add per cycle. We used the IBM GNU C compiler and used the `-O3` and `-funroll-loops` optimization options (preliminary testing showed that the GNU C compiler produced faster code than the Sun C compiler).

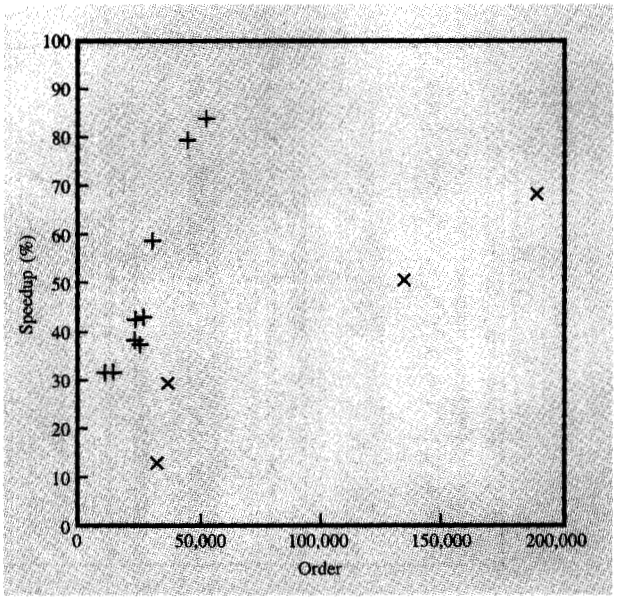

Figure 5

Scatter plot of the relative reduction in running time due to ordering versus the order of the matrix. Each mark represents one matrix, with Boeing matrices represented by + and NasGraph matrices by ×. The reduction in running time is computed as $(T_3 - T_4)/T_3$, where T_3 is the running time with blocking and prefetching using a random ordering, and T_4 is the running time with blocking and prefetching using the original ordering.

The C language subroutines compiled and ran without a problem. It took about a day to produce assembly-language versions of the routines and to introduce prefetching into them.

Our performance results, which are based on the performance of the msc10848 and msc23052 matrices, can be summarized as follows. The basic C version on randomly permuted matrices ran at 10.6–11.4 MFLOPS. The same version on the original ordering of the matrices ran at 16.5–16.8 MFLOPS. The relative improvement due to reordering alone is larger than on the POWER2 machine (for the same matrices), because of the smaller size of the primary cache on the UltraSPARC I. Replacing integer indices with pointers did not improve performance, and neither did prefetching; also, these optimizations did not slow down the algorithm. Blocking the matrices into 2-by-2 and 1-by-2 blocks improved performance to 20.8–22.2 MFLOPS.

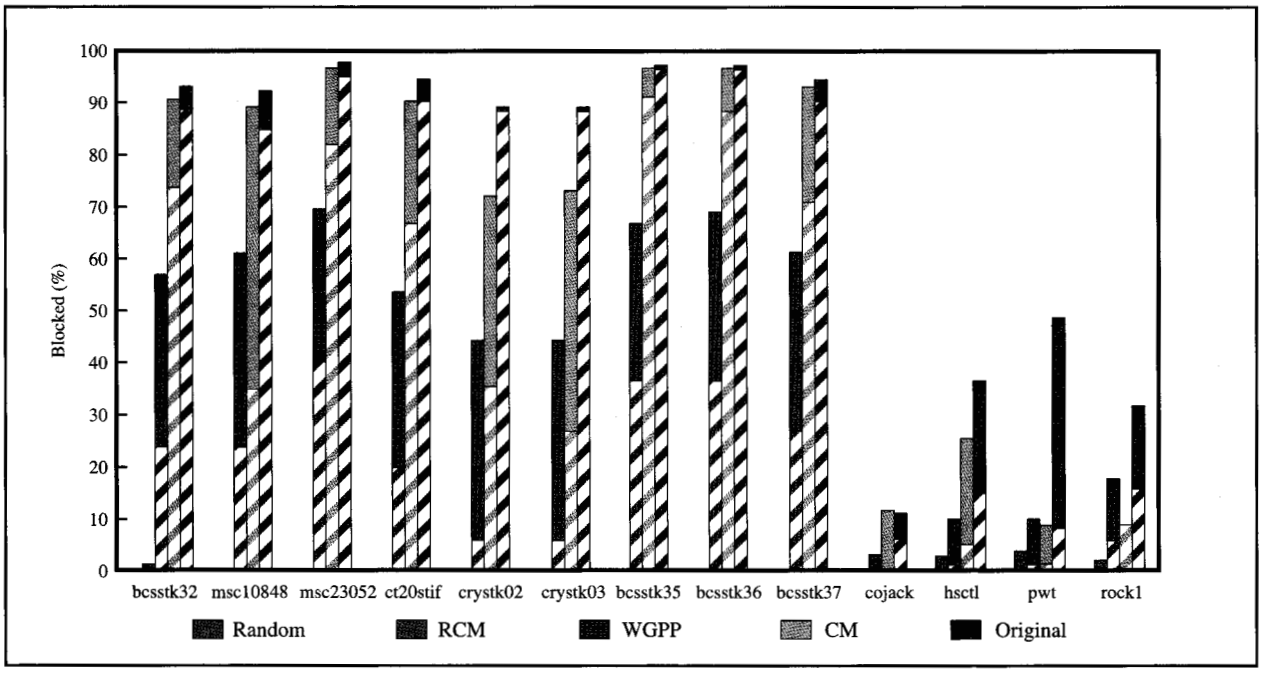

Figure 6

Fraction of matrix nonzeros that is blocked in 2-by-2 and 1-by-2 blocks. The graph shows the fraction of nonzeros in blocks for five different orderings of each matrix (the first two usually result in very low levels of blocking). The striped portion of each bar represents the fraction of nonzeros in 2-by-2 blocks, and the solid portion the fraction of the remaining nonzeros that are blocked in 1-by-2 blocks. The levels of blocking when only 1-by-2 blocks are used are slightly higher than the combined levels shown here.

The integer-to-pointer conversion did not help, probably because the UltraSPARC I has a dedicated address adder, so address calculations and integer-to-pointer conversions (i.e., shift instructions) do not compete for the same functional units. The most likely reason that prefetching did not improve performance is that the UltraSPARC I has only one load/store unit, whereas the prefetching techniques are meant to prevent multiple units from stalling on the same cache line.

The performance improvements shown by our experiments verify that our techniques, with the possible exception of prefetching, are general and should improve performance on most superscalar RISC machines. We note that the overall performance level in these experiments, which may seem low compared to the peak performance of the processor, is in fact quite similar to the performance of similar numerical codes on the UltraSPARC. For example, the double-precision dot product subroutine in the FORTRAN BLAS [19], as well as the same subroutine in the Sun Performance Library, runs at less than 20 MFLOPS on this machine. The main reason for this performance level seems to be the small primary cache coupled with the short cache lines (16 bytes).

6. Conclusions

We have presented four techniques for accelerating sparse-matrix vector multiplication on superscalar RISC processors. One of the techniques, precomputing addresses for indirect addressing, is trivial but important. The technique of reordering the matrix to reduce its bandwidth and hence reduce cache misses has been proposed by Das et al. [2] and investigated further in two other papers [3, 4]. We have explored it further and found that the Cuthill–McKee technique yields excellent results on a variety of matrices. Two other techniques, representing nonzeros in small dense blocks and prefetching to allow cache-hits-under-miss processing, are novel (others have proposed representing nonzeros in larger blocks [5, 6]). They, too, improve performance significantly on many matrices. On an IBM RS/6000 workstation, the combined effect of the four techniques can improve performance from about 15 MFLOPS to more than 95 MFLOPS, depending on the size and sparseness of the matrix.

Our techniques, with the exception of prefetching, are general for superscalar RISC architectures. The ordering and blocking techniques should improve performance on

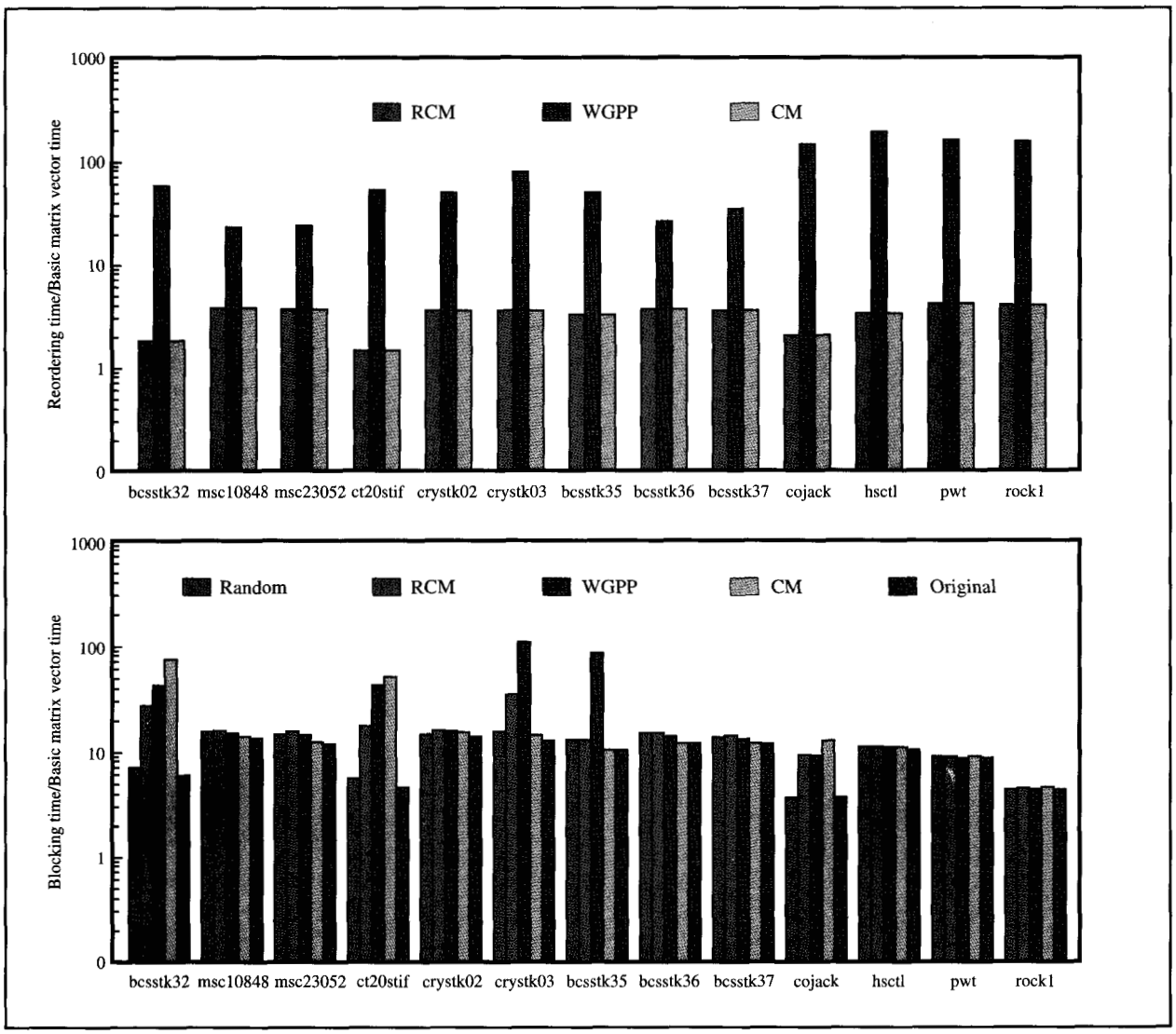

Figure 7

Cost of reordering the matrices (top) and of blocking them (bottom). The cost is shown in terms of time relative to the basic matrix vector multiplication time with a randomly permuted ordering of the matrix. The longer blocking times were caused by excessive paging.

most RISC processors, as shown by our experiments and the experiments of Burgess and Giles [4]. Replacing integers with pointers should improve performance on RISC machines that use the same functional units for address calculations and for shifts. The code that implements these three techniques is portable. The prefetching technique should improve performance on superscalar RISC processors that have more than one load/store unit. Our implementation method for this technique can be duplicated on most machines, but the technique cannot be considered portable.

Although even basic matrix vector multiplication codes perform better when the matrices are not extremely sparse (<10 nonzeros per row), highly optimized codes are even more sensitive to the sparseness of the matrices.

Reordering sparse matrices using the Cuthill–McKee ordering has another benefit in sparse iterative solvers. When a conjugate gradient solver uses an incomplete Cholesky preconditioner, the ordering of the matrix affects the convergence rate. Duff and Meurant [20] compared the convergence rate of an incompletely Cholesky-preconditioned conjugate gradient with 17 different

orderings on four model problems. In their tests the Cuthill–McKee and the reverse Cuthill–McKee resulted in convergence rates that were best or close to best. Although it is possible to use different orderings for preconditioning and matrix vector multiplication, doing so requires permuting a vector twice in every iteration. This extra step renders each iteration more expensive. Using a Cuthill–McKee or similar ordering for both steps eliminates the need to permute vectors in every iteration, leads to few cache misses in the matrix vector multiplication step (and very likely in the preconditioning step as well), enables blocking, and accelerates convergence.

Our proposed hardware-assisted prefetching can eliminate the somewhat complicated coding technique that we used to implement prefetching in the irregular loop over the sparse rows of the matrix. Mowry [13] compared compiler-based software-directed prefetching to prefetching by hardware with no software intervention, as proposed by Lee [21], Porterfield [12], and Baer and Chen [22]. Our proposal lies somewhere between the two extremes. It enjoys very little instruction overhead, since the prefetching instructions are outside, rather than inside, the inner loop. It does not suffer from excessive memory contention overhead, because prefetching is enabled and disabled by software. Finally, our proposal enables prefetching in irregular loops, which Mowry did not consider. The main disadvantages of our proposal are the need to augment the instruction set and the hardware cost. A more detailed study is required, however, in order to assess the effectiveness of our proposed hardware-assisted prefetching scheme.

Acknowledgments

Part of this work was done while the author was a postdoctoral fellow at the IBM Thomas J. Watson Research Center. Fred Gustavson, Bowen Alpern, and David Burgess read and commented on early versions of this paper; their comments helped improve the paper considerably. Thanks to Anshul Gupta for details regarding the performance of his sparse factorization code and for assistance with the use of his matrix reordering package, WGPP. Thanks to Ramesh Agarwal for stimulating discussions, and to the anonymous referees for comments that helped us to further improve the paper.

References

1. R. Barret, M. Berry, T. Chan, J. Demmel, J. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. van der Vorst, *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*, SIAM Press, Philadelphia, 1993.
2. R. Das, D. J. Mavriplis, J. Saltz, S. Gupta, and R. Ponnusamy, "The Design and Implementation of a Parallel Unstructured Euler Solver Using Software Primitives," *AIAA J.* **32,** 489–496 (1994).
3. O. Temam and W. Jalby, "Characterizing the Behavior of Sparse Algorithms on Caches," *Proceedings of Supercomputing '92*, IEEE Computer Society Press, Piscataway, NJ, 1992, pp. 578–587.
4. D. A. Burgess and M. B. Giles, "Renumbering Unstructured Grids to Improve the Performance of Codes on Hierarchical Memory Machines," *Technical Report 95/06*, Numerical Analysis Group, Oxford University Computing Laboratory, Oxford, England, May 1995.
5. R. C. Agarwal, F. G. Gustavson, and M. Zubair, "A High-Performance Algorithm Using Pre-Processing for Sparse Matrix-Vector Multiplication," *Proceedings of Supercomputing '92*, IEEE Computer Society Press, Piscataway, NJ, November 1992, pp. 32–41.
6. Satish Balay, William Gropp, Lois Curman McInnes, and Barry Smith, *PETSc 2.0 Users' Manual, Technical Report ANL-95/11, Revision 2.0.15*, Argonne National Laboratory, Argonne, IL, 1996.
7. E. Cuthill and J. McKee, "Reducing the Bandwidth of Sparse Symmetric Matrices," *Proceedings of the 24th National Conference of the Association for Computing Machinery*, New York, 1969, pp. 157–172.
8. Yousef Saad, *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company, New York, 1996.
9. John H. Edmondson, Paul Rubinfeld, Ronald Preston, and Vidya Rajagopalan, "Superscalar Instruction Execution in the 21164 Alpha Microprocessor," *IEEE Micro*, pp. 33–43 (April 1995).
10. Marc Tremblay and J. Michael O'Connor, "UltraSparc I: A Four-Issue Processor Supporting Multimedia," *IEEE Micro*, pp. 42–49 (April 1996).
11. R. C. Agarwal, F. G. Gustavson, and M. Zubair, "Improving Performance of Linear Algebra Algorithms for Dense Matrices Using Algorithmic Prefetch," *IBM J. Res. Develop.* **38,** 265–275 (1994).
12. A. K. Porterfield, "Software Methods for Improvement of Cache Performance on Supercomputer Applications," Ph.D. thesis, Rice University, Houston, May 1989.
13. Todd C. Mowry, "Tolerating Latency Through Software-Controlled Data Prefetching," Ph.D. thesis, Stanford University, March 1994.
14. R. C. Agarwal, B. Alpern, L. Carter, F. G. Gustavson, D. J. Klepacki, R. Lawrence, and M. Zubair, "High-Performance Parallel Implementations of the NAS Kernel Benchmarks on the IBM SP2," *IBM Syst. J.* **34,** 263–272 (1995).
15. S. W. White and S. Dhawan, "POWER2: Next Generation of the RISC System/6000 Family," *IBM J. Res. Develop.* **38,** 493–502 (1994).
16. E. H. Welbon, C. C. Chan-Nui, D. J. Shippy, and D. A. Hicks, "The POWER2 Performance Monitor," *IBM J. Res. Develop.* **38,** 545–554 (1994).
17. A. Gupta, "Fast and Effective Algorithms for Graph Partitioning and Sparse-Matrix Ordering," *IBM J. Res. Develop.* **41,** 171–184 (1997).
18. Anshul Gupta, "WGPP: Watson Graph Partitioning (and Sparse Matrix Ordering) Package," *Technical Report RC-20453*, IBM Thomas J. Watson Research Center, Yorktown Heights, NY, May 1996.
19. C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh, "Basic Linear Algebra Subprogram for Fortran

Usage," *ACM Trans. Math. Software* **5,** 308–323 (1979).
20. Iain S. Duff and Gérard Meurant, "The Effect of Ordering on Preconditioned Conjugate Gradient," *BIT* **29,** 635–657 (1989).
21. R. L. Lee, "The Effectiveness of Caches and Data Prefetch Buffers in Large-Scale Shared Memory Multiprocessors," Ph.D. thesis, University of Illinois at Urbana-Champaign, May 1987.
22. J.-L. Baer and T.-F. Chen, "An Effective On-Chip Preloading Scheme to Reduce Data Access Penalty," *Proceedings of Supercomputing '91*, IEEE Computer Society Press, Piscataway, NJ, 1991, pp. 176–186.

Received November 25, 1996; accepted for publication August 1, 1997

Sivan Toledo *Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304 (toledo@parc.xerox.com).*
Dr. Toledo received the B.Sc. degree in mathematics and computer science and the M.Sc. degree in computer science from Tel-Aviv University in Israel, both in 1991. He received the Ph.D. degree in computer science from MIT in 1995. He was a postdoctoral fellow in the Mathematical Sciences Department at the IBM Thomas J. Watson Research Center, where he worked on high-performance mathematical software. Since October 1996 Dr. Toledo has been a postdoctoral associate at the Xerox Palo Alto Research Center, where he works on solvers for large sparse nonsymmetric linear systems.