Susanne M. Balle
Simon C. Steely, Jr.

Analyzing Memory Access Patterns of Programs on Alpha-based Architectures

The development of efficient algorithms on today's high-performance computers is far from straightforward. Applications need to take full advantage of the computer system's deep memory hierarchy, and this implies that the user must know exactly how his or her implementation is executed. The ability to understand or predict the execution path without looking at the machine code can be very difficult with today's compilers. By using the outputs from an experimental memory access profiling tool, the programmer can compare memory access patterns of different algorithms and gain insight into the algorithm's behavior, e.g., potential bottlenecks resulting from memory accesses. The use of this tool has helped improve the performance of an application based on sparse matrix-vector multiplications.

The development of efficient algorithms on today's high-performance computers can be a challenge. One major issue in implementing high-performing algorithms is to take full advantage of the deep memory hierarchy. To better understand a program's performance, two things need to be considered: computational intensiveness and the amount of memory traffic involved. In addition to the latter, the pattern of the memory references is important because the success of hierarchy is attributed to locality of reference and reuse of data in the user's program.

In this paper, we investigate the memory access pattern of Fortran programs. We begin by presenting an experimental Atom[1] tool that analyzes how the program is executed. We developed the tool to help us understand how different compiler switches impact the algorithm implemented and to determine if the algorithm is doing what it is intended to do. In addition, our tool helps the process of translating an algorithm into an efficient implementation on a specific machine. The work presented in this paper focuses primarily on a better understanding of the behavior of technical applications. Related work for Basic Linear Algebra Subroutine implementations has been described.[2] In most scientific programs, the data elements are matrix-elements that are usually stored in two-dimensional (2-D) arrays (column-major in Fortran). Knowing the order of array referencing is important in determining the amount of memory traffic.

In the final section of this paper, we present an example of a memory access pattern study and illustrate how the use of our program analysis tool improved the considered algorithm's performance. Guidelines on how to use the tool are given as well as comments about conclusions to be derived from the histograms generated.

Memory Access Profiling Tool

Our experimental tool generates a set of histograms for each reference in the program or in the subroutine under investigation. The first histogram measures

strides from the previous reference, the second histogram gives the stride from the second-to-last reference, and so on, for a total of MAXEL histograms for each memory reference in the part of the program we investigate. By stride, we mean the distance between two memory references (load or store). We chose a MAXEL of five for our case study, but MAXEL can be given any value.

Two variants of this tool were implemented.

1. The first version takes all memory references into account in all histograms.
2. The second version takes into account in the next histogram those memory references whose stride is more than 128 bytes. It does not consider in the $(i + 1)$th histogram $(i = 1,\ldots,5)$ strides that are less than 128 bytes in the ith histogram.

The second version of the tool has proven to be more useful in understanding the access patterns. It highlights memory accesses that are stride one for a while and then have a stride greater than 128 bytes. The choice of 128 bytes was arbitrary; the value can be changed.

The following bins are used in the histograms: 0- through 127-byte strides are accounted for separately. Strides greater than or equal to 128 bytes are grouped into the following intervals: [128 through 255], [256 through 511], [512 through 1,023], [1,024 through 2,047], [2,048 through 4,095], [4,096 through 8,191], [8,192 through 16,383], [16,384 through 32,767], and [32,768 through infinity].

In the next section, we present the output histograms obtained with the second version of this experimental tool for a Fortran loop. In our case study, we chose to perform the histograms on a single array instead of all references in the program. This method provided a clearer picture of the memory access pattern for each array in the piece of the program under consideration. We present separate histograms for the loads and the stores of each array in the memory traffic of the subroutine we investigated.

When looking at memory access patterns, it is important not to include load instructions that perform prefetching. Even though prefetching adds to the memory traffic, its load instructions pollute the memory access pattern picture.

Case Study

In this section, we study and compare different versions of the code presented in Figure 1 using our experimental memory access profiling tool. We show that the same code is not executed in the same way for different compiler switches. Often a developer has to delve deeply into the assembler of the given loop to understand how and when the different instructions

```
1   Q(i)=0, i=1, n
2   do k1= 1, 4
3      index = (k1-1) * numrows
4      do j=1,n
5         p1=COLSTR(j,k1)
6         p2=COLSTR(j+1,k1)-1
7         p3= [snip]
8         sum0=0.d0
9         sum1=0.d0
10        sum2=0.d0
11        sum3=0.d0
12        x1 = P(index+ROWIDX(p1,k1))
13        x2 = P(index+ROWIDX(p1+1,k1))
14        x3 = P(index+ROWIDX(p1+2,k1))
15        x4 = P(index+ROWIDX(p1+3,k1))
16        do k = p1, p3, 4
17           sum0 = sum0 + AA(k,k1) * x1
18           sum1 = sum1 + AA(k+1,k1) * x2
19           sum2 = sum2 + AA(k+2,k1) * x3
20           sum3 = sum3 + AA(k+3,k1) * x4
21           x1 = P(index+ROWIDX(k+4,k1))
22           x2 = P(index+ROWIDX(k+5,k1))
23           x3 = P(index+ROWIDX(k+6,k1))
24           x4 = P(index+ROWIDX(k+7,k1))
25        enddo
26        do k = p3+1, p2
27           x1=P(index+ROWIDX(k,k1))
28           sum0 = sum0 + AA(k,k1)*x1
29        enddo
30        YTEMP(j,k1)=sum0+sum1+sum2+sum3
31     enddo
32     do i = 1, n, 4
33        Q(i) = Q(i) + YTEMP(i,k1)
34        Q(i+1) = Q(i+1) + YTEMP(i+1,k1)
35        Q(i+2) = Q(i+2) + YTEMP(i+2,k1)
36        Q(i+3) = Q(i+3) + YTEMP(i+3,k1)
37     enddo
38  enddo

where n = 14000,
real*8 AA(511350,4), YTEMP(n,4)
real*8 Q(n), P(n)
integer*4 ROWIDX(511350,4), COLSTR(n,4)
```

Figure 1
Original Loop

are executed. The output histograms from our tool ease that process and give a clear picture of the reference patterns. The loop presented in Figure 1 implements a sparse matrix-vector multiplication and is part of a larger application. Ninety-six percent of the application's execution time is spent in that loop. We analyze the loop compiled with two different sets of compiler switches. To illustrate the effective use of the tool, we present the enhanced performance results due to changes made based on the output histograms.

From lines 5 and 6 in the loop shown in Figure 1, we would expect the array *COLSTR* to be read stride one 100 percent of the time. Line 30 of the figure indicates that *YTEMP* is accessed stride one through the whole *j* loop. From lines 33 through 36, we expect *YTEMP*'s stride to be equal to one most of the time and equal to the number of columns in the array every time *k*1 is incremented. *Q* should be referenced 100

percent stride one for both the loads and the stores (lines 33 through 36). As illustrated in lines 12 through 15, 21 through 24, and 27, *ROWIDX* is expected to be accessed with a stride of one between the *p*1 and *p*2 bounds of the *k* loop. Even though it looks like the *k* loop is violating the array bounds of *ROWIDX* in lines 21 through 24 for the last iteration of the loop, this is not the case. We expect array *P* to have nonadjacent memory references since we have deliberately chosen an algorithm that sacrifices this array's access patterns to improve the memory references of *Q* and *AA*.

Original Code

We investigate the memory access patterns achieved by the loop in Figure 1 when compiled with the following switches:

```
f77 -g3 -fast -O5
```

The `-g3` switch is needed to extract the addresses of the arrays from the symbol table. For more information on DIGITAL Fortran compiler switches, see Reference 3.

From Figures 2 and 3, we see that array *Q* is accessed as we expected, 100 percent stride one for the loads and the stores. Since *Q* is accessed contiguously in 100 percent of its memory references, we will not have any entries in the next four histograms. As described in the previous section, we only record in the next histogram the strides that are greater than 128 bytes in the current histogram.

Figure 4 illustrates that *COLSTR* is accessed 50 percent stride zero and 50 percent stride one. This is unexpected since lines 5 and 6 in Figure 1 suggest that this array would be accessed stride one 100 percent of the time. The fact that we have entries only for the strides between the current and the previous loads indicates that the elements of *COLSTR* are accessed in a nearly contiguous way. A closer look at Figure 1 tells us that the compiler is loading *COLSTR* twice. We expected the compiler to do only one load into a register and reuse the register. The work-around is to perform a scalar replacement as described by Blickstein et al.[4] We put $p2 = COLSTR(1,k1) - 1$ outside the *j* loop and substituted inside the *j* loop $p1 = COLSTR(j,k1)$ with $p1 = p2 + 1$. Inside the *j* loop, *p*2 remains the same. Eliminating the extra loads did not enhance performance, and a possible assumption is that the analysis done by the compiler concluded that no gain would result from that optimization.

Figures 5 and 6 show the strides for the loads and the strides for the stores for the array *YTEMP*. One more time, the implementation is not being executed the way we thought it would. In Figure 1, lines 33 through 36 suggested that *YTEMP* would be referenced stride one through the whole *i* loop as well as with a stride

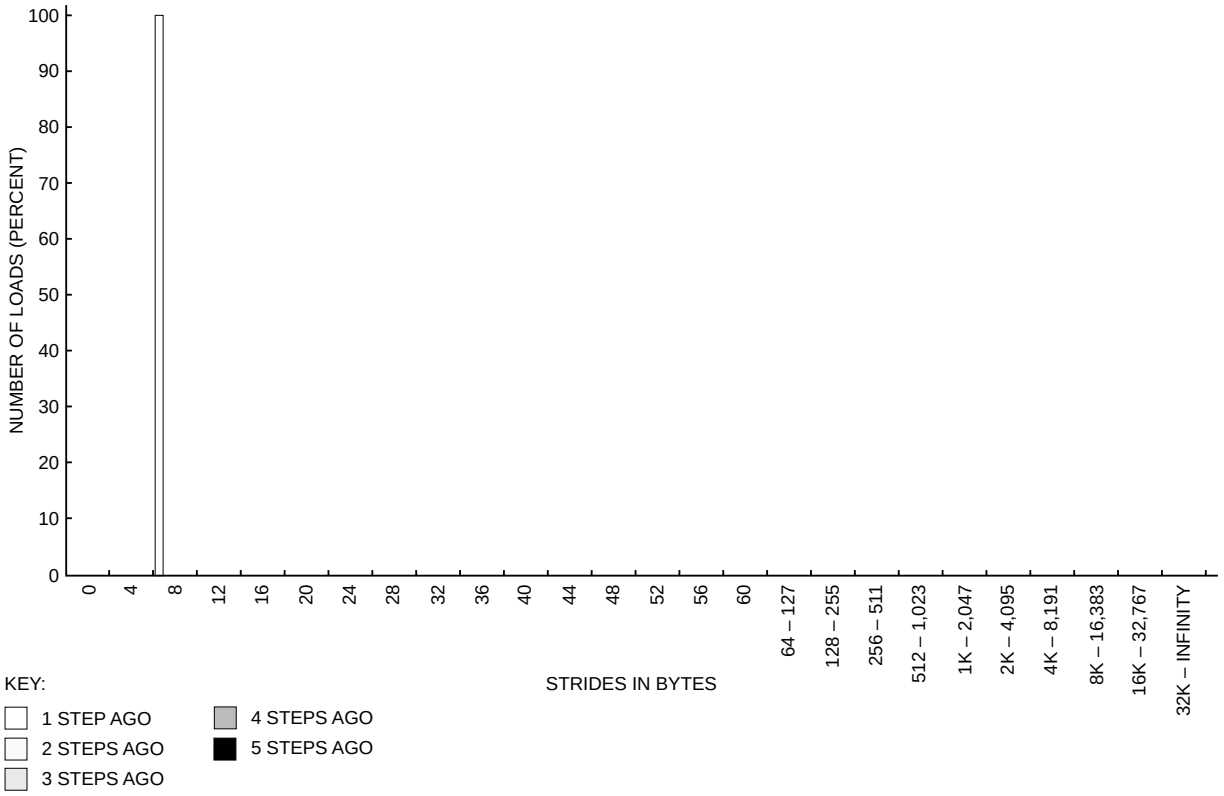

Figure 2
Strides for Array *Q* between the Current Load and the Load One through Five Steps Ago

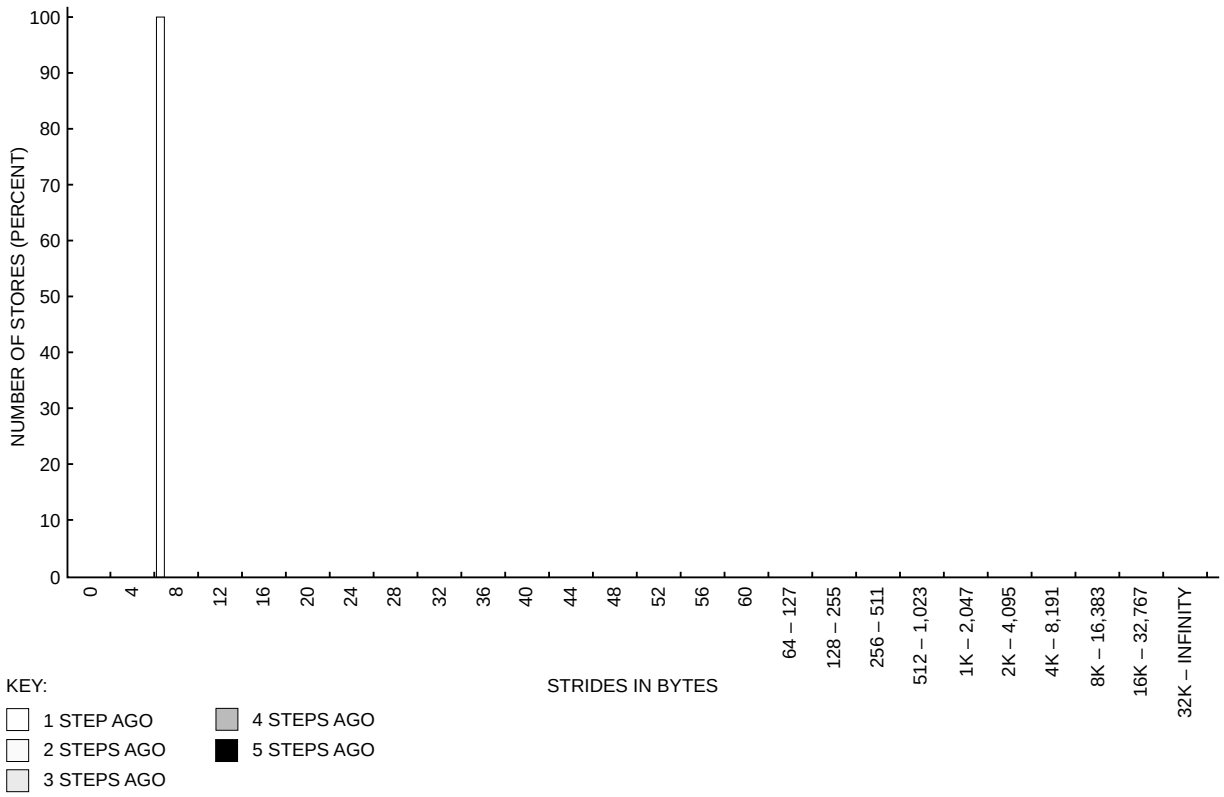

Figure 3
Strides for Array *Q* between the Current Store and the Store One through Five Steps Ago

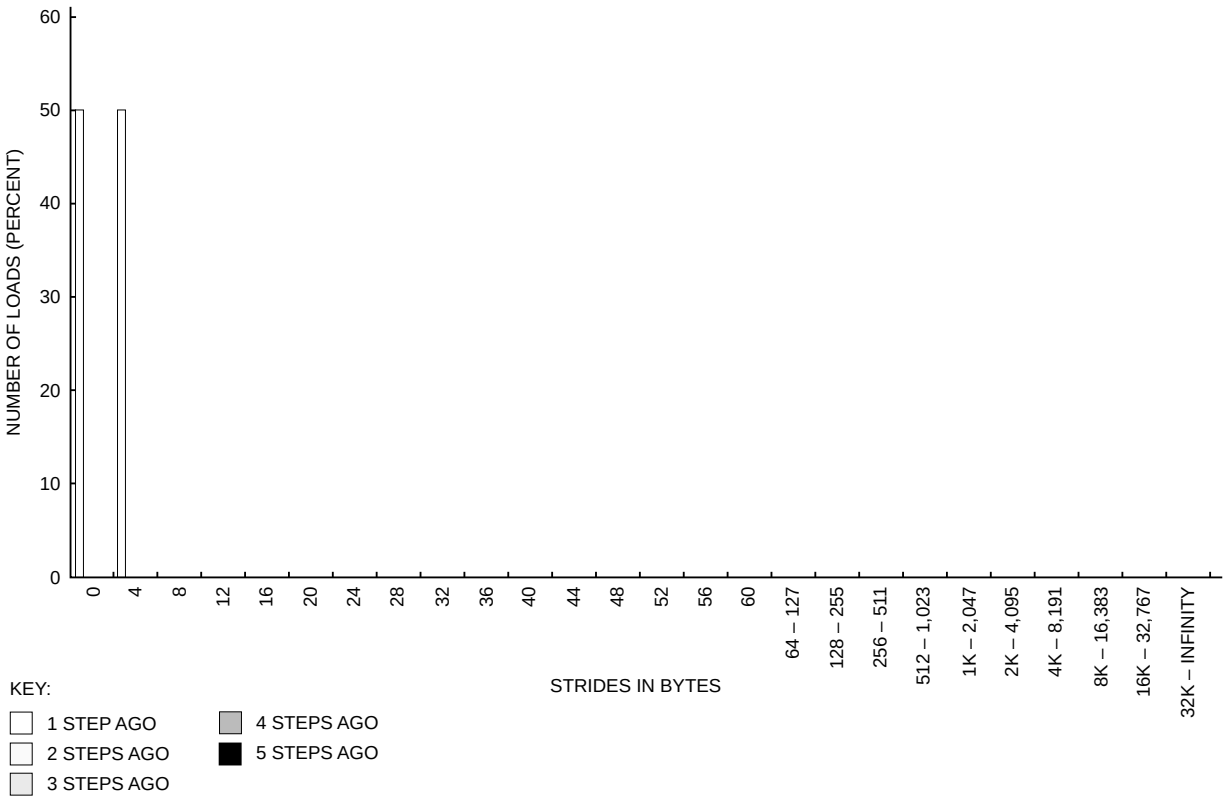

Figure 4
Strides for Array *COLSTR* between the Current Load and the Load One through Five Steps Ago

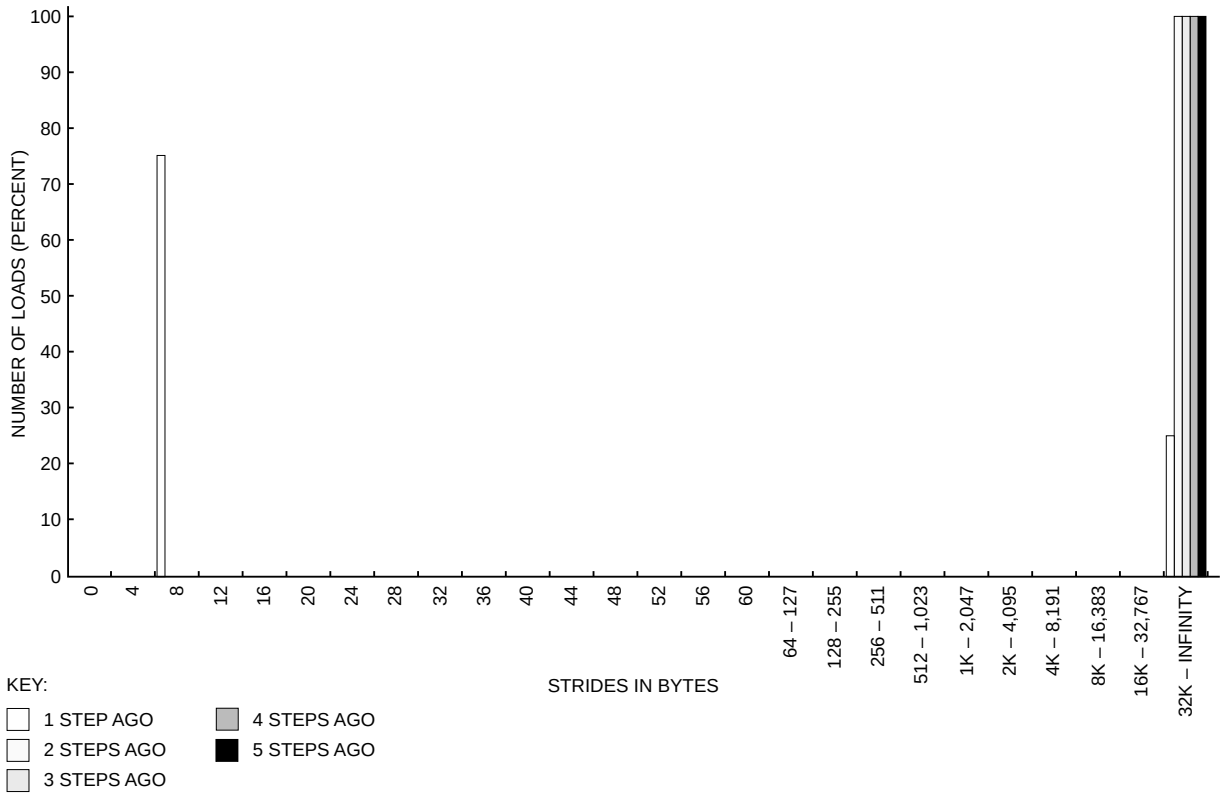

Figure 5
Strides for Array *YTEMP* between the Current Load and the Load One through Five Steps Ago

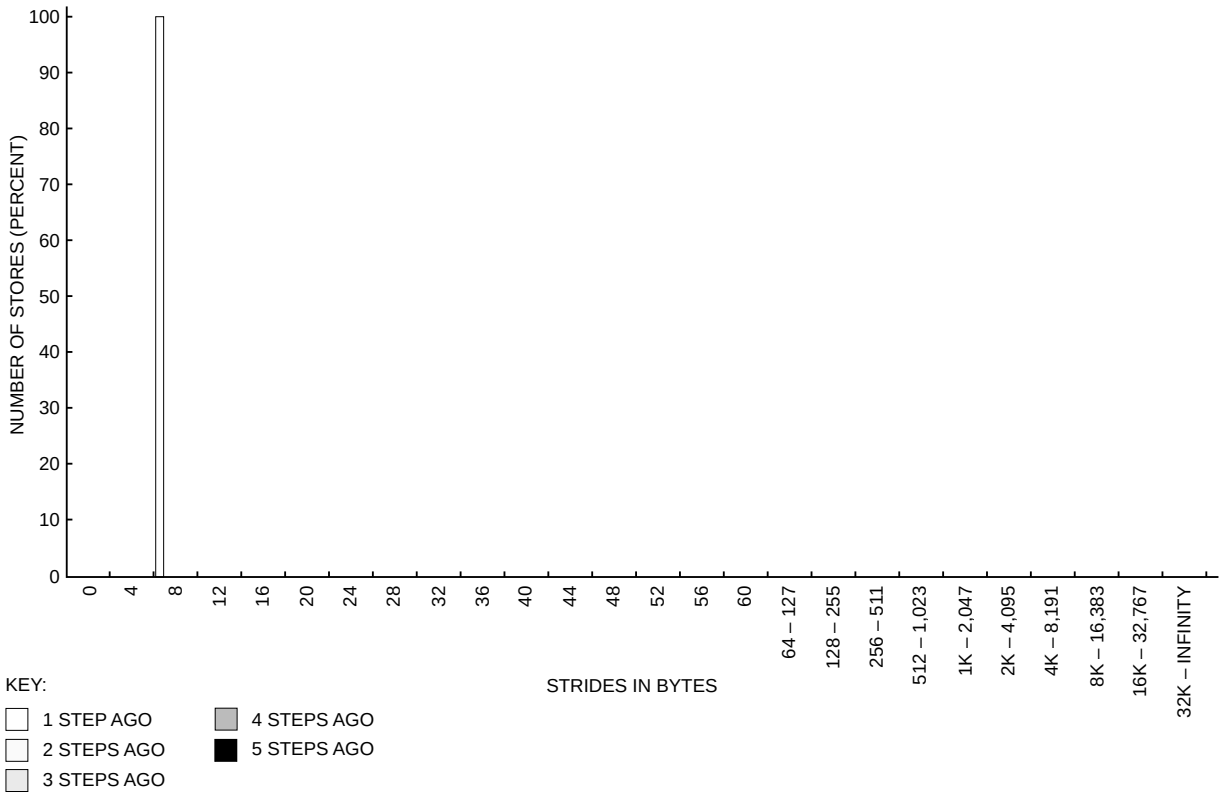

Figure 6
Strides for Array *YTEMP* between the Current Store and the Store One through Five Steps Ago

equal to the number of columns in the array when *k*1 is incremented. By considering Figure 5 along with lines 33 through 36 in Figure 1, we conclude that *YTEMP* is unrolled by four in the *k*1-direction in the *i* loop. The fact that all strides between the current load and the load two loads back or three loads back or four loads back have a stride between 32K and infinity is consistent with traversing a matrix along rows. Figure 6 shows that the *j* loop is not unrolled by four in the *k*1-direction, because all the loads of *YTEMP* are 100 percent stride one. The compiler must split the *k*1 loop into two separate loops, the first consists of the *j* loop and the second consists of the *i* loop. The latter has been unrolled by four in the *k*1-direction thereby eliminating the extra overhead from the *k*1 loop.

Figure 7 shows that the matrix *AA* is accessed as we expected. The strides are not greater than 128 bytes or, in other words, a maximum stride of 16 elements. The fact that there is no stride other than the one between the current load and the previous load in the histograms shows that *AA* is referenced in a controlled way. In this case, *AA* is accessed 39 percent of its total loads in stride one and 23 percent in stride two.

From lines 12 through 15, 17 through 20, and 21 through 24 in Figure 1, we know that the arrays *AA* and *ROWIDX* should have relatively similar behaviors. Only the four extra prefetches of *ROWIDX* in lines 21 through 24 for the last iteration in the *j* loop differentiate the access patterns of the two arrays. Figure 8 confirms that assumption. *ROWIDX* is referenced with controlled strides. Because *ROWIDX* is accessed close to contiguously, we will not have any entries in the next four histograms. As described in the previous section, we only record in the next histogram the strides that are greater than 128 bytes in the current histogram. *ROWIDX* is referenced 24 percent of its total loads in stride one and 34 percent in stride two.

As illustrated in Figure 9, array *P* is accessed exactly the way we expected it. When designing this algorithm, we had to make some compromises. We decided to have *AA* and *Q* referenced as closely as possible to stride one, thus giving up the control of *P*'s references.

By examining these arrays' access patterns, we can see how they are accessed and whether or not the implementation is doing what it is supposed to do. If the loop in Figure 1 is used on a larger matrix [$n = 75{,}000$ and $AA(204427,12)$ has 15 million nonzero elements], the execution time for the total application on a single 21164 processor of an AlphaServer 8400 5/625 system is 1,970 seconds. The application executes 26 x 75 (= 1,950) times the considered loop. When profiling the program, we measured that the loop under investigation takes 96 percent of the total execution time. It is therefore a fair assumption to say that any improvement in this building block will improve the overall performance of the total program.

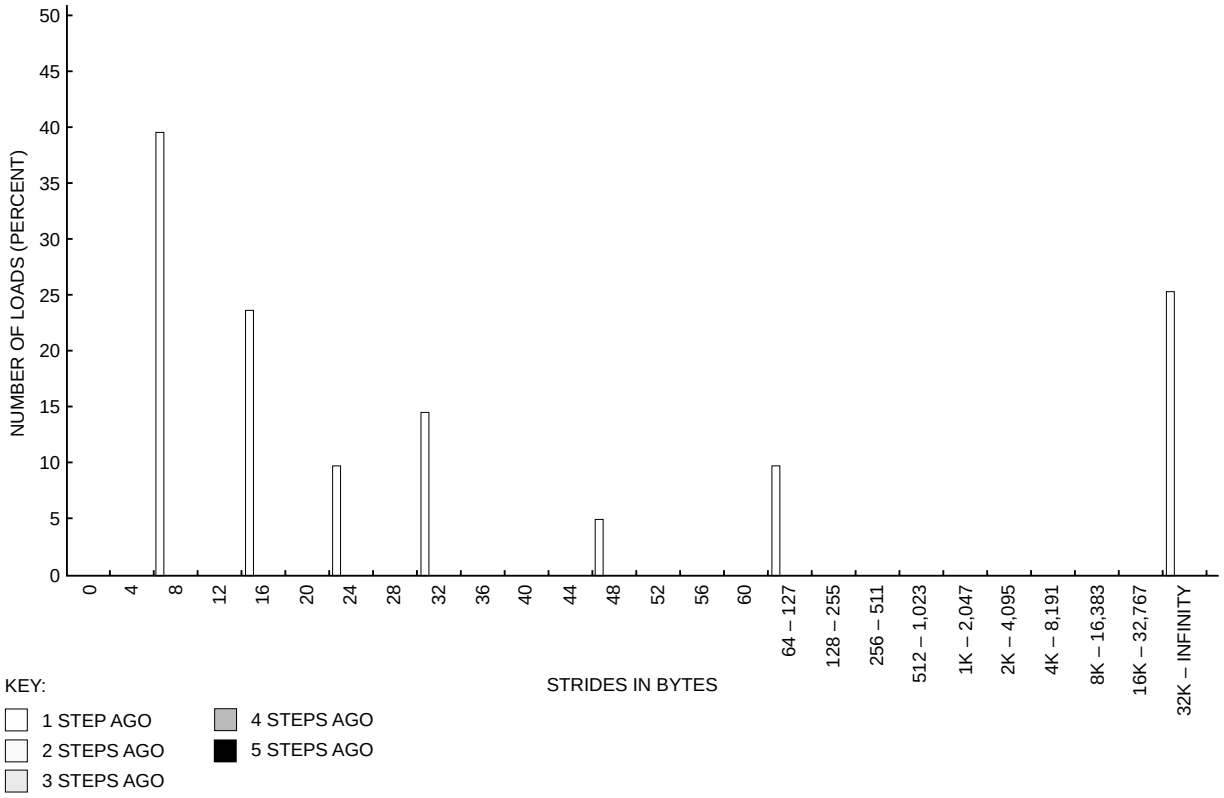

Figure 7
Strides for Array *AA* between the Current Load and the Load One through Five Steps Ago

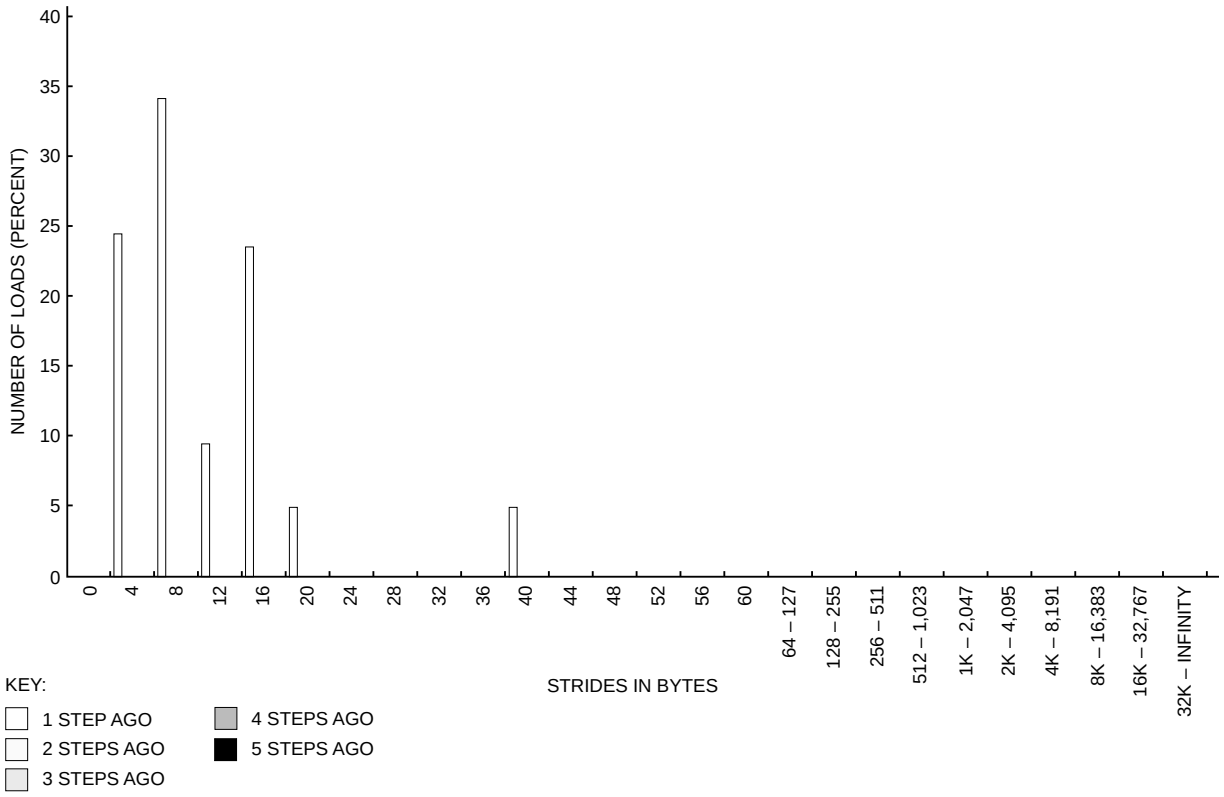

Figure 8
Strides for Array *ROWIDX* between the Current Load and the Load One through Five Steps Ago

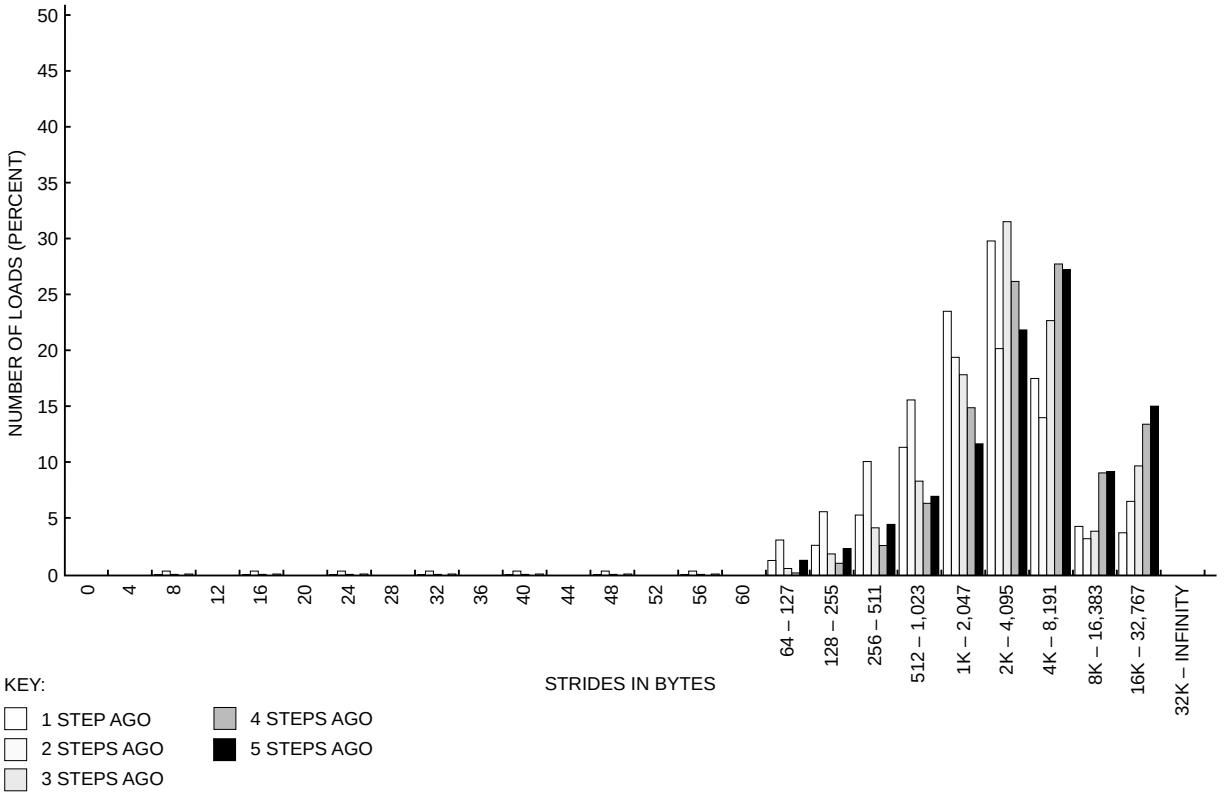

Figure 9
Strides for Array *P* between the Current Load and the Load One through Five Steps Ago

Modified Code

In this section, we describe a new experiment in which we used different compiler switches and changed the original loop to the loop in Figure 10. The code changes were based on the analysis in the previous section as well as on a more extended series of investigations.

In this example, we used the following compiler switches:

```
f77 -g3 -fast -O5 -unroll 1
```

Lines 3, 5, and 6 from Figure 10 show that we implemented the scalar replacement technique as described by Blickstein et al.[4] to avoid *COLSTR* being loaded twice. From Figure 11, we see that array *COLSTR* is now behaving as we expect: 100 percent of the strides for the loads are stride one.

In our first attempt to optimize the original loop, we split the *k*1 loop into two loops in the same way the compiler did as described in the previous section. We then hand unrolled the *YTEMP* array in the *k*1 direction. Further analysis showed that a considerable gain could be made by removing the *YTEMP* array and writing the results directly into *Q*. By replacing the zeroing out of the *Q* array (Figure 1, line 1) with an IF statement (Figure 10, line 30), we further improved the performance of the loop. The last two changes were possible because we decided that, for performance enhancement issues, the serial version of the code was going to be different from its parallel version.

```
1  do k1= 1, 4
2     index = (k1-1) * numrows
3     p2=COLSTR(1,k1)-1
4     do j=1,n
5        p1=p2+1
6        p2=COLSTR(j+1,k1)-1
7        p3= [snip]
8        sum0=0.d0
9        sum1=0.d0
10       sum2=0.d0
11       sum3=0.d0
12       x1 = P(index+ROWIDX(p1,k1))
13       x2 = P(index+ROWIDX(p1+1,k1))
14       x3 = P(index+ROWIDX(p1+2,k1))
15       x4 = P(index+ROWIDX(p1+3,k1))
16       do k = p1, p3, 4
17          sum0 = sum0 + AA(k,k1) * x1
18          sum1 = sum1 + AA(k+1,k1) * x2
19          sum2 = sum2 + AA(k+2,k1) * x3
20          sum3 = sum3 + AA(k+3,k1) * x4
21          x1 = P(index+ROWIDX(k+4,k1))
22          x2 = P(index+ROWIDX(k+5,k1))
23          x3 = P(index+ROWIDX(k+6,k1))
24          x4 = P(index+ROWIDX(k+7,k1))
25       enddo
26       do k = p3+1, p2
27          x1=P(index+ROWIDX(k,k1))
28          sum0 = -sum0 + AA(k,k1)*x1
29       enddo
30       if(k1.eq.1) then
31          Q(j) = sum0 + sum1 + sum2 + sum3
32       else
33          Q(j) = Q(j) + sum0 + sum1 + sum2 + sum3
34       endif
35    enddo
36 enddo

where n = 14000,
real*8 AA(511350,4)
real*8 Q(n), P(n)
integer*4 ROWIDX(511350,4), COLSTR(n,4)
```

Figure 10
Modified Loop

Figures 12 and 13 show that *Q*'s load and store access pattern is 100 percent stride one as we expected it to be. For both *ROWIDX* and *AA*, we see a significant increase in stride one references. Figure 14 shows that *AA* is now accessed 69 percent stride one instead of 39 percent. *ROWIDX*'s stride one increased to 52 percent from 24 percent as illustrated in Figure 15. These two arrays are the reason for using the `-unroll 1` switch. Without it, stride one for both arrays would stay approximately the same as in the previous study. The pattern of accesses of array *P* in Figure 16 is similar to the prior pattern of accesses in Figure 9 as expected.

To better understand the effects of the unrolling, we counted the number of second-level cache misses for 26 calls to the loop, using an Atom tool[1] that simulated a 4-megabyte direct-mapped cache. By considering only these 26 matrix-vector multiplications, we do not get a full picture of what is going on and how the different arrays interact. Nevertheless, it gives us hints about what caused the improvement in performance. Use of the cache tool on the whole application would increase the run time dramatically.

Twenty-six calls to the original loop (Figure 1) have a total of 1,476,017,322 memory references, of which 77,638,624 are cache misses. The modified loop (Figure 10), on the other hand, has fewer references due to the fact that we eliminated an expensive array initialization at each step and removed the temporary array *YTEMP*. The number of cache misses dropped from 77,638,624 to 72,384,348 or a reduction in misses of 7 percent. If we compile the modified loop without the `-unroll 1` switch, the number of cache misses increases slightly. On the 21164 Alpha microprocessor, all the misses are effectively performed in serial. This means that for memory-bound codes like the loop we are currently investigating, execution time primarily depends on the number of cache misses.

The histograms illustrating the access strides for the different arrays helped us design a more suitable algorithm for our architecture. By increasing the stride one references in the loads for the arrays *AA* and *ROWIDX*, eliminating the extra references in *COLSTR* and *Q*, and improving the strides for *Q*, we increased the performance of this application dramatically. Counting the number of cache misses gave us a better understanding as to why the new access patterns achieve enhanced performance. It also helped us understand that not allowing the compiler to unroll the already hand-unrolled loops in the modified loop decreased the number of cache misses. The execution time for this application [n = 75,000 and *AA*(204427,12) has 15 million nonzero elements] decreased from 1,970 seconds to 1,831 seconds on a single 625-megahertz (MHz) 21164 Alpha microprocessor of an AlphaServer 8400 5/625 system. This is an improvement of 139 seconds or 8 percent.

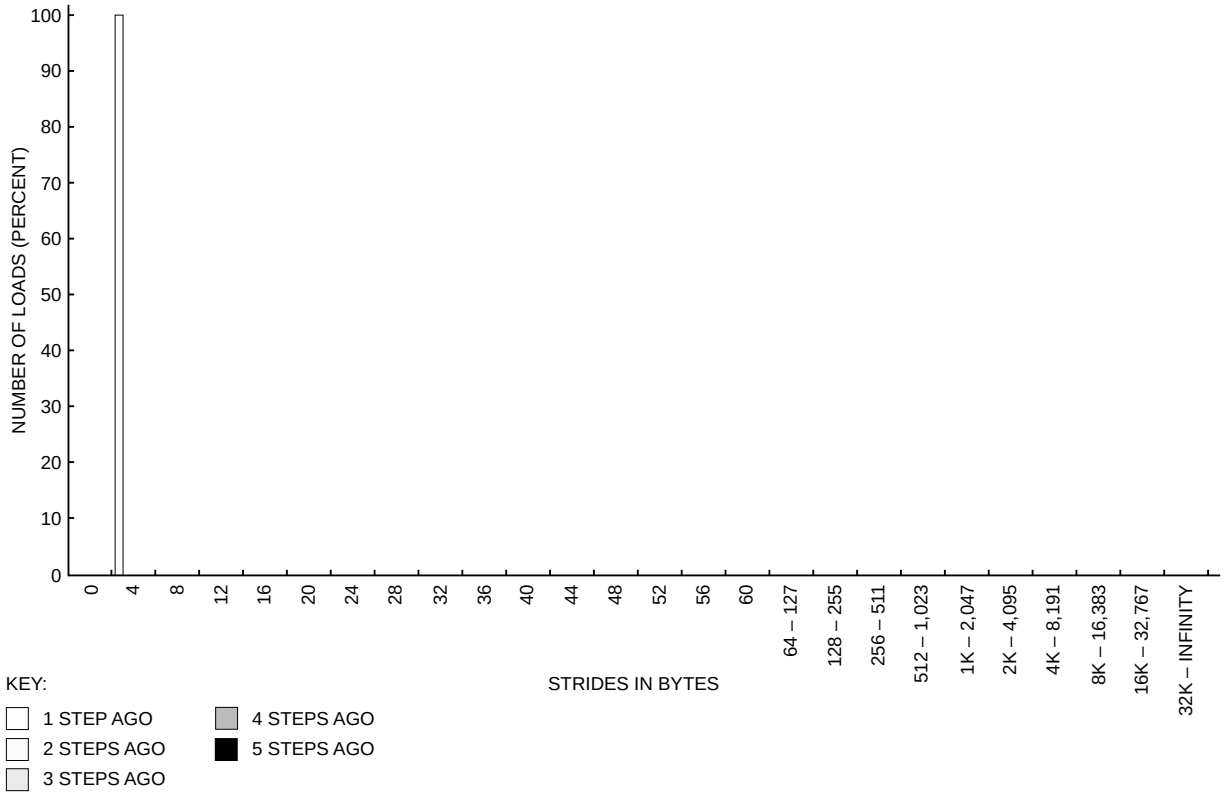

Figure 11
Strides for Array *COLSTR* between the Current Load and the Load One through Five Steps Ago

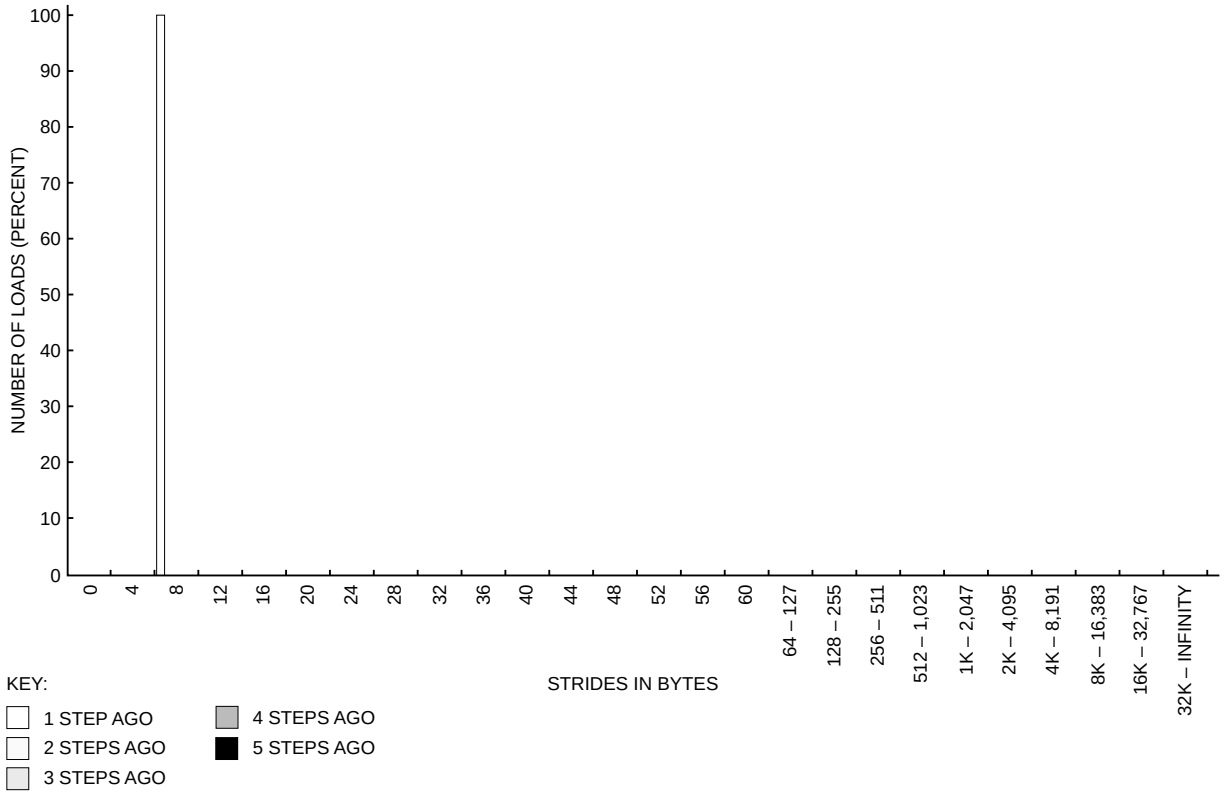

Figure 12
Strides for Array *Q* between the Current Load and the Load One through Five Steps Ago

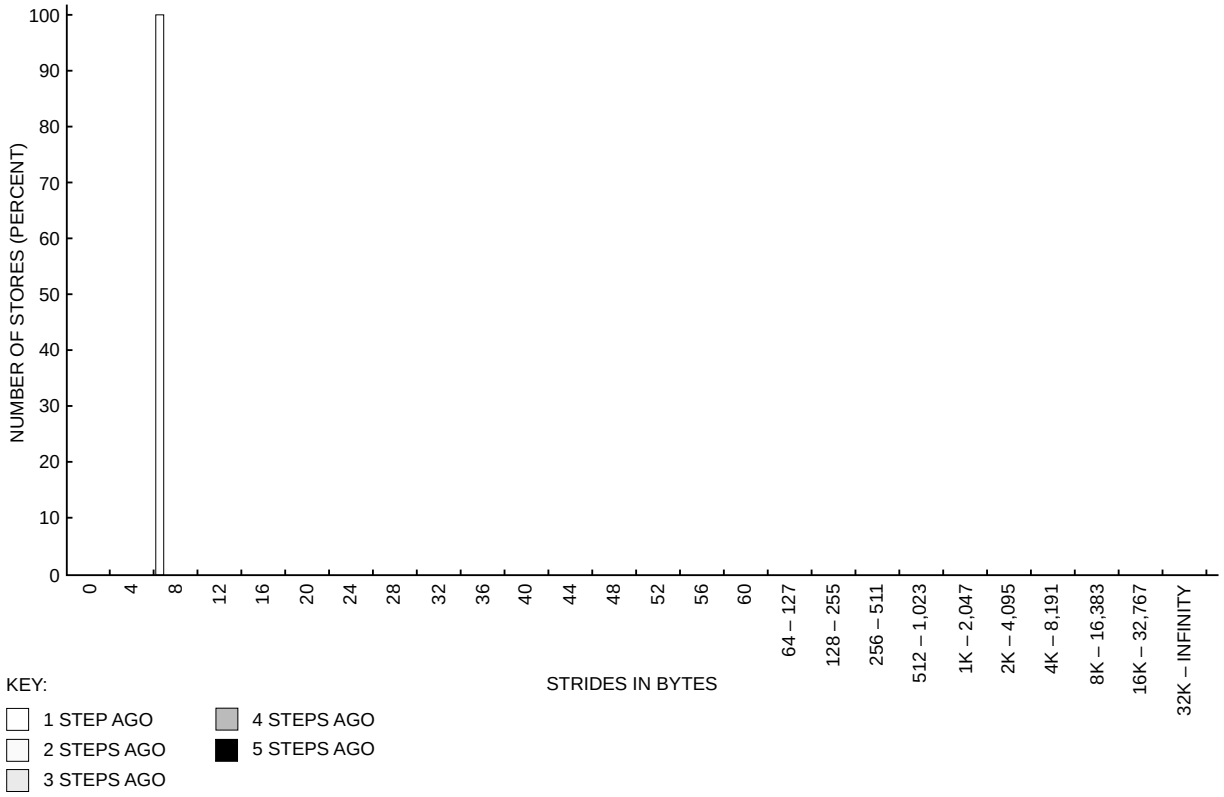

Figure 13
Strides for Array *Q* between the Current Store and the Store One through Five Steps Ago

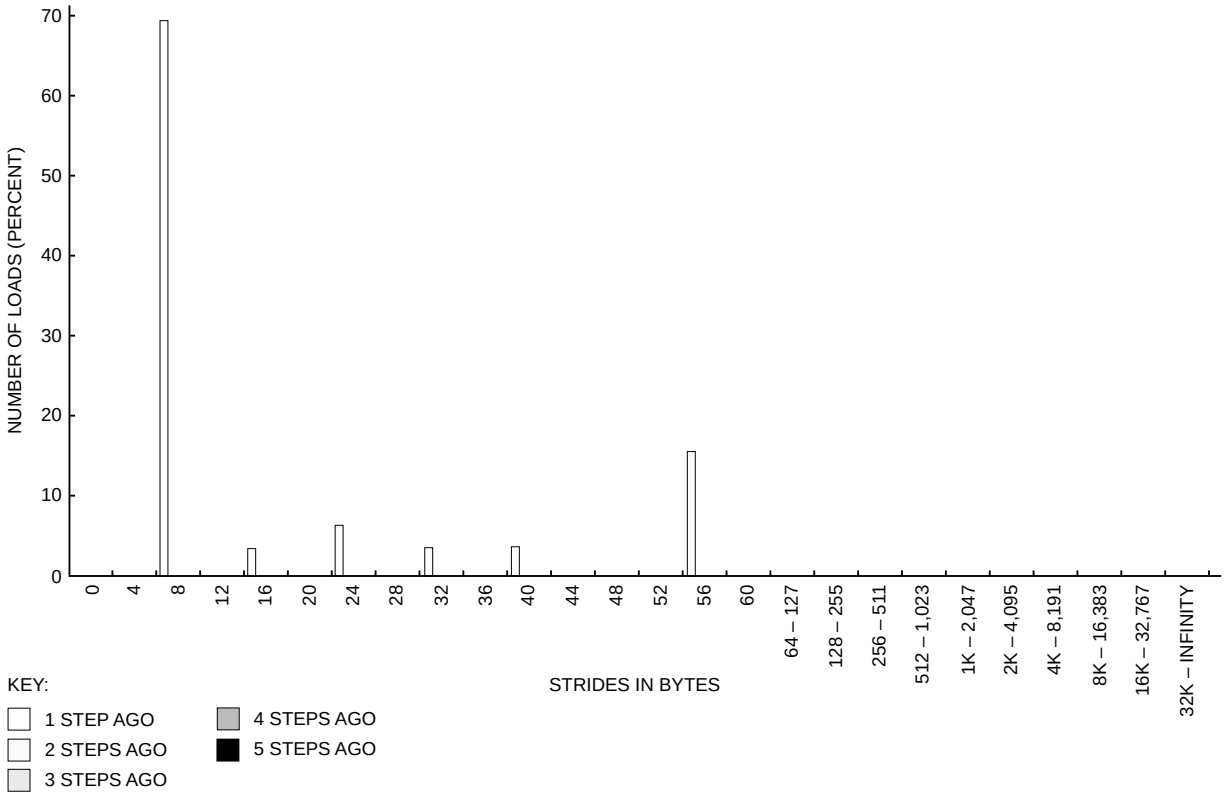

Figure 14
Strides for Array *AA* between the Current Load and the Load One through Five Steps Ago

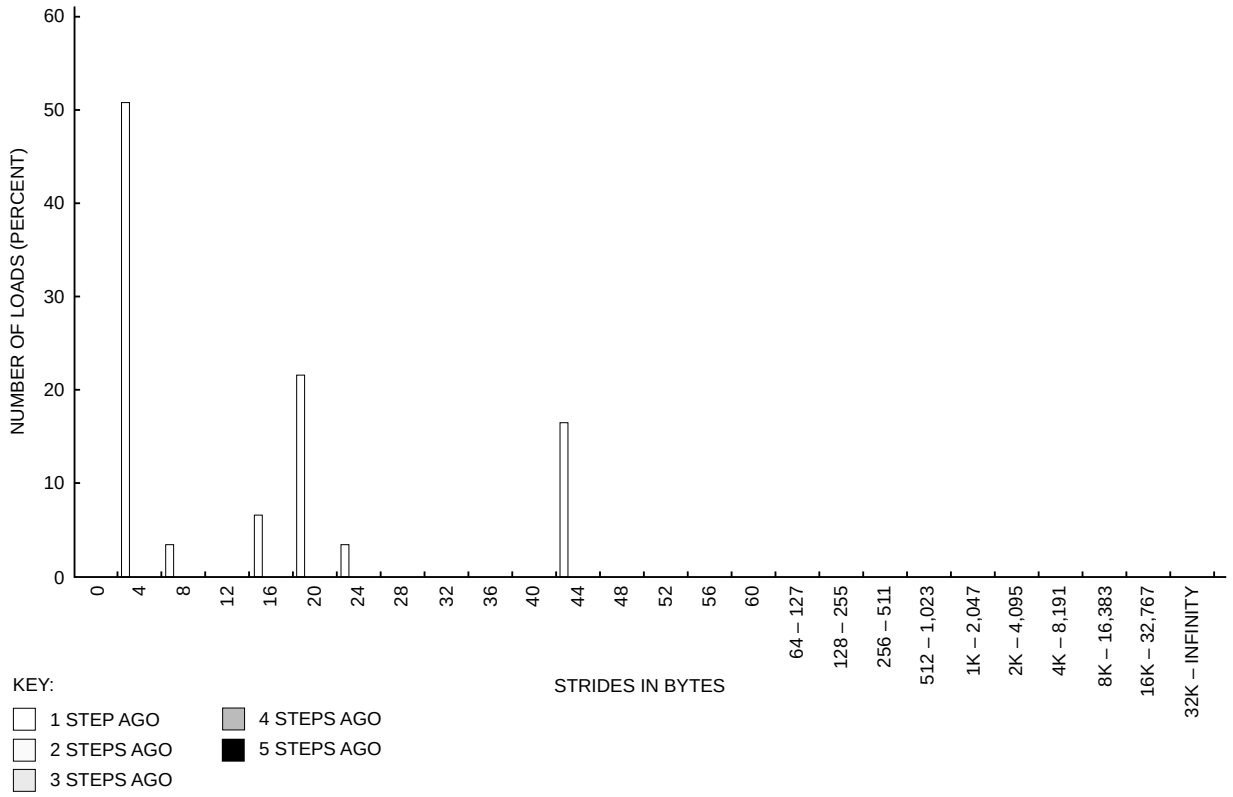

Figure 15
Strides for Array *ROWIDX* between the Current Load and the Load One through Five Steps Ago

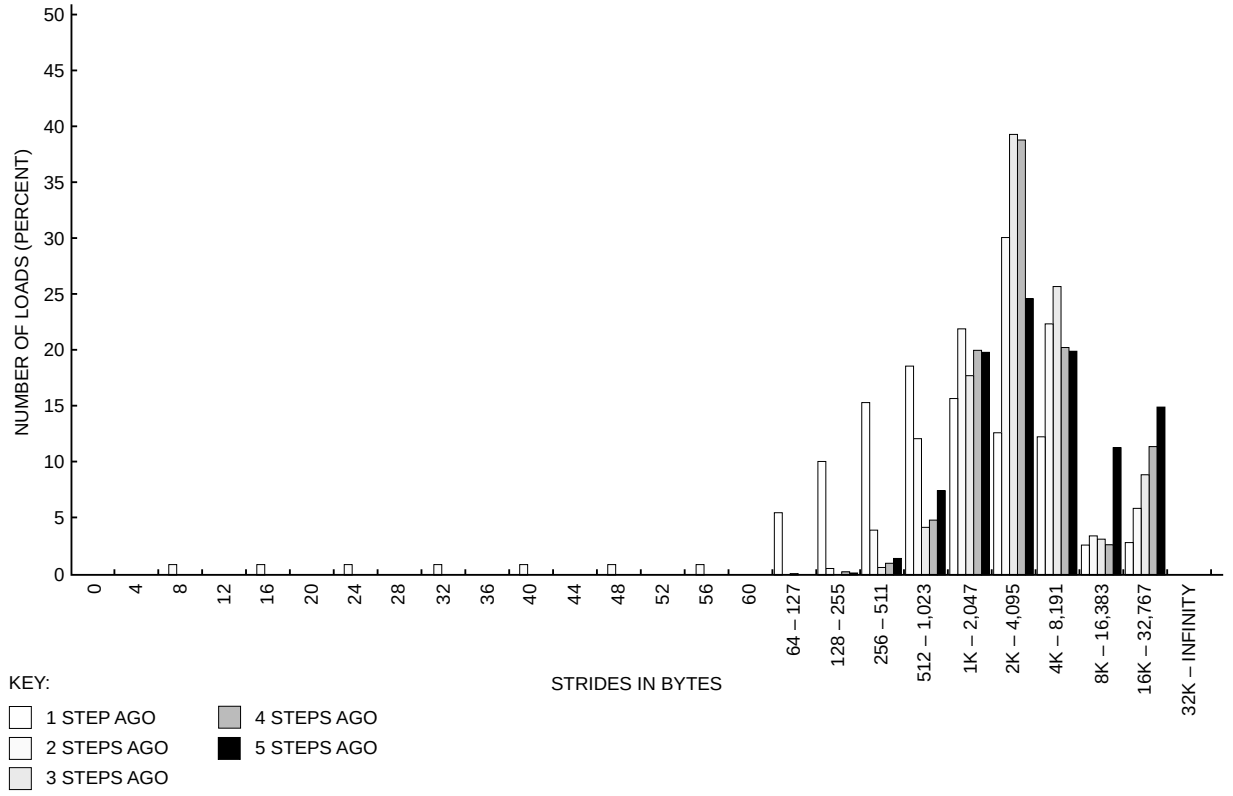

Figure 16
Strides for Array *P* between the Current Load and the Load One through Five Steps Ago

Conclusion

The case study shows that, given the right program analysis tools, a program developer can take better advantage of his or her computer system. The experimental tool we designed was very useful in providing insight into the algorithm's behavior. The approach considered yields an improvement in performance of 8 percent on a 625-MHz 21164 Alpha microprocessor. This is definitely a worthwhile exercise since a substantial reduction in execution time was obtained using straightforward and easy guidelines.

The data collected from a memory access profiling tool helps the user understand a given program as well as its memory access patterns. It is an easier and faster way to gain insight into a program than examining the listing and the assembler generated by the compiler. Such a tool enables the programmer to compare memory access patterns of different algorithms; therefore, it is very useful when optimizing codes. Probably its most important value is that it shows the developer if his or her implementation is doing what he or she thinks the algorithm is doing and highlights potential bottlenecks resulting from memory accesses. Optimizing an application is an iterative process, and being able to use relatively easy-to-use tools like Atom is a very important part of the process. The major advantage of the tool presented in this paper is that no source code is needed, so it can be used to analyze the performance of program executables.

Acknowledgments

The authors wish to thank David LaFrance-Linden, Dwight Manley, Jean-Marie Verdun, and Dick Foster for fruitful discussions. Thanks to John Eck, Dwight Manley, Ned Anderson, and Chuck Schneider for reading and commenting on earlier versions of the paper. Special thanks go to the anonymous referees for comments and suggestions that helped us considerably improve the paper. Finally, we thank John Kreatsoulas and Dave Fenwick for encouraging our collaboration.

References

1. *Programmer's Guide, Digital UNIX Version 4.0*, chapter 9 (Maynard, Mass., Digital Equipment Corporation, March 1996).
2. O. Brewer, J. Dongarra, and D. Sorensen, *Tools to Aid in the Analysis of Memory Access Patterns for Fortran Programs*, Technical Report, Argonne National Laboratory (June 1988).
3. *DEC Fortran Language Reference Manual* (Maynard, Mass., Digital Equipment Corporation, 1997).
4. D. Blickstein, P. Craig, C. Davidson, R. Faiman, K. Glossop, R. Grove, S. Hobbs, and W. Noyce, The GEM Optimizing Compiler System, *Digital Technical Journal,* vol. 4, no. 4 (1992): 121–136.

Biographies

Susanne M. Balle
Susanne Balle is currently a member of the High Performance Computing Expertise Center. Her work areas are performance prediction for 21264 microprocessor-based architectures, memory access pattern analysis, parallel Lanczos algorithms for solving very large eigenvalue problems, distributed-memory matrix computations, as well as improving performance of standard industry and customer benchmarks. Before joining DIGITAL, she was a postdoctoral fellow at the Mathematical Sciences Department at the IBM Thomas J. Watson Research Center where she worked on high-performance mathematical software. From 1992 to 1995, she consulted for the Connection Machine Scientific Software Library (CMSSL) group at Thinking Machines Corporation. Susanne received a Ph.D. in mathematics and an M.S. in mechanical engineering and computational fluid dynamics from the Technical University of Denmark. She is a member of SIAM.

Simon C. Steely, Jr.
Simon Steely is a consulting engineer in DIGITAL's AlphaServer Platform Development Group. In his 21 years at DIGITAL, he has worked on many projects, including development of the PDP-11, VAX, and Alpha systems. His work has focused on writing microcode, designing processor and system architecture, and doing performance analysis to make design decisions. In his most recent project, he was a member of the architecture team for a future system. In addition, he led the team developing the cache-coherency protocol on that project. His primary interests are computer architecture, performance analysis, prediction technologies, cache/memory hierarchies, and optimization of code for best performance. Simon has a B.S. in engineering from the University of New Mexico and is a member of IEEE and ACM. He holds 15 patents and has several more pending.