Unfavorable Strides in Cache Memory Systems

David H. Bailey

December 9, 1994

Ref: *Scientific Programming*, vol. 4 (1995), pg. 53–58

Abstract

An important issue in obtaining high performance on a scientific application running on a cache-based computer system is the behavior of the cache when data is accessed at a constant stride. Others who have discussed this issue have noted an odd phenomenon in such situations: a few particular innocent-looking strides result in sharply reduced cache efficiency. In this paper, this problem is analyzed, and a simple formula is presented that accurately gives the cache efficiency for various cache parameters and data strides.

The author is with the Numerical Aerodynamic Simulation (NAS) Systems Division at NASA Ames Research Center, Moffett Field, CA 94035.

Introduction

Scientists accustomed to running large computationally intensive applications on Cray supercomputers have never had to concern themselves with cache issues. However, with the recent sharp rise in the floating point performance of RISC workstations, many scientists are now using these systems for serious computations, and cache issues can no longer be avoided. Another avenue from which supercomputer scientists have been introduced to cache memories is the recent incorporation of RISC processors into highly parallel supercomputers. In any event, it is clear that serious programmers need to understand better how caches operate, so that they can implement their algorithms in ways that optimize potential performance.

An important concept in this paper is memory stride, i.e. the increment in memory address, measured in words, between successive elements fetched or stored in the important inner loops of an application program. Many important scientific applications do not feature exclusively stride one data access but instead feature large nonunit strides. For instance, many codes perform similar operations on each dimension of a two or three dimensional array. Performing computations in the first dimension of a Fortran program (or the last dimension of a C program) can be done with unit stride, but the strides of the computations in the other dimensions are typically large values, and significantly degraded performance may result when the codes are ported to cache-based systems without change.

One solution to this problem is to rewrite the code to employ array transpositions between the computational steps in each dimension. In this way all computation can be done at unit stride. But such revision may require substantial effort, and it may still not result in significant performance improvement unless the time spent in stride one computation is substantial enough to offset the cost of the array transpositions.

As a result, many problems of this sort are simply ignored, as scientists accept with a certain fatalism their codes will not perform very well. However, for some programs the reduction in performance is sufficiently large that it is worthwile to make an effort to understand and alleviate this problem.

1. Definitions and Notation

To better understand the phenomenon of performance reduction with strides, consider the following model of a cache memory system. First assume that the cache is configured with $R = 2^r$ cache lines, and assume that each cache line contains $W = 2^w$ words, so that a total of RW words can be cached.

It will be assumed that this cache memory system operates as follows. When a word at a virtual address A is fetched, it is placed in cache location Q, where Q is determined by zeroing the bits in the address to the left of the rightmost $r + w$ bits and then shifting the resulting integer to the right by w bits (i.e. dividing by W). Note that this operation produces an integer Q in the range $0 \leq Q < R$. When a single word is requested, all W words of the W-long cache line that it resides in are also fetched.

Many cache-based systems employ "associativity sets." This means that up to C cache lines with the same cache address, as determined by the mapping function described in the

previous paragraph, can be stored simultaneously in the cache. In this way, potentially RC lines or RCW words may be cached. When a request is made for data that is not in cache, its cache line replaces one of the C lines currently stored at the cache location where it is assigned by the mapping function. One some systems the least recently used line is replaced, while on others the line replaced is determined by some unspecified "random" procedure. The above model of an associative cache is satisfied by many, but not all current RISC systems.

If the stride S of a vector fetch is unity, then W consecutive words reside on the same cache line. This is obviously a very favorable situation. The situation is similarly quite favorable if the memory stride is some integer less than W, since in that case many cache lines contain multiple words required by the CPU. Many scientific applications, however, involve strides larger than W, so that each cache line retrieved from memory contains at most one word required by the CPU. This last case will be the focus of this paper.

Unfortunately, at some strides even RC words cannot be cached because some of the associativity sets are overutilized, while others are underutilized. Let us consider a vector fetch of L words with stride S and ask what fraction of the L resulting cache lines remain in the cache when the fetch is complete. This question is of interest for two reasons: (1) a computation may need to access this same set of L words again, and (2) if this vector fetch was a single row of a matrix stored in column major order (as in Fortran), the next W rows of the matrix reside in these same cache lines. Either way, performance will be significantly improved if this data can remain in the cache.

Accordingly, the efficiency E of a vector fetch of length L will be defined as T/L, where T is the number of cache lines that still remain in the cache when the vector fetch operation is complete, and where L is the vector length. For simplicity, in the following it will be assumed that $L = RC$.

An obvious example of an inefficient stride is a large power of two. Then all cache lines will be fetched into the same location of the cache, and the other $R - 1$ locations will be completely unutilized. In other words, at most C lines of this data can be stored in the cache. The resulting efficiency is only $1/R$. Clearly if an application program has arrays whose dimensions are large powers of two, these arrays should be "padded," such as by declaring their leading dimensions (in Fortran) to be slightly larger than a power of two. In this way, accesses of successive rows of data from such an array will have cache addresses that are slightly offset, resulting in much more efficient cache utilization. Most users of Cray systems are familiar with this tuning technique, since it eliminates bank conflicts that may reduce performance by factors as high as 10 or 20 [1].

2. Cache Efficiency with Non-Power-of-Two Strides

It may come as a surprise to some that large power of two strides are not the only particularly unfavorable strides for cache memory systems [4]. To facilitate concrete discussion in the following, we will consider the particular case $R = 32$, $C = 4$ and $W = 16$. These values match the cache parameters of the IBM RS 6000/320 system. We will also assume in the following that the vector length L of the fetch is 128.

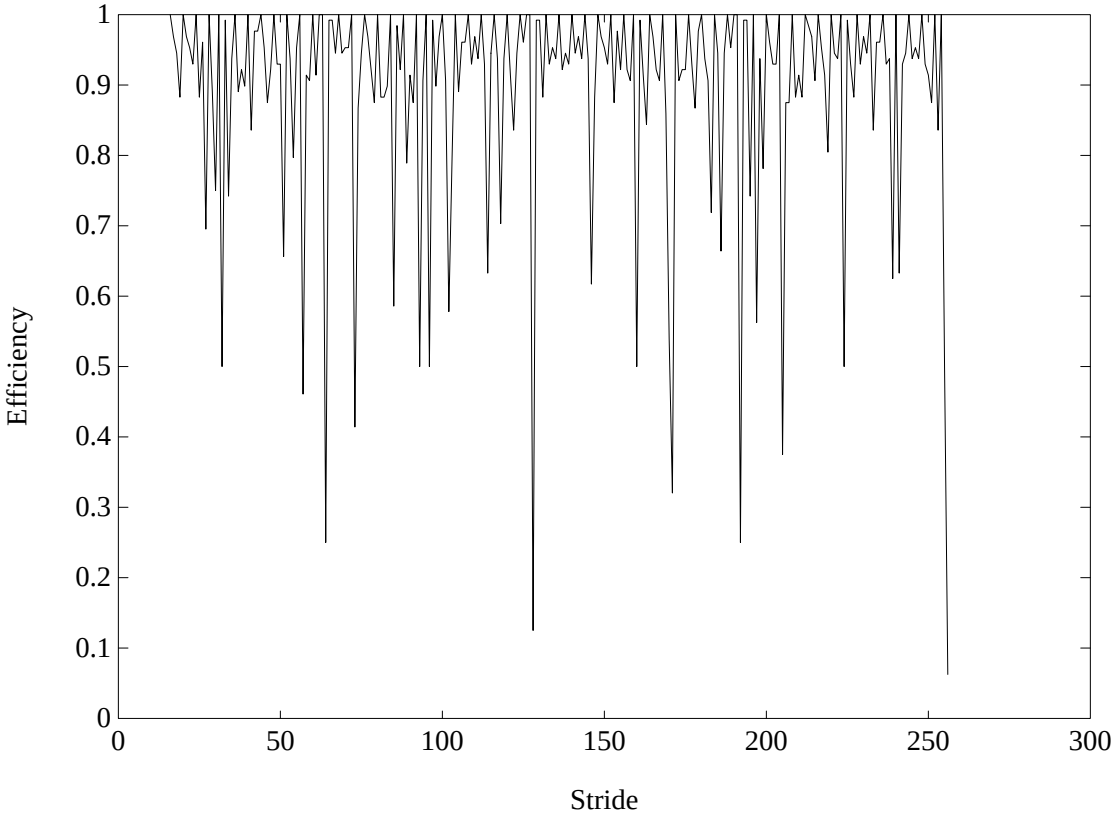

Figure 1: Cache Efficiencies for Various Strides

When $S = 72$, it turns out that in 128 consecutive fetches, the respective cache lines neatly fill the 32×4 array, resulting in perfect utilization of the cache (except that only one word in each cache line may actually be required by the CPU). The resulting efficiency E is unity, even though 72 is divisible by eight, a highly unfavorable situation on many vector computers. Now consider $S = 73$, a completely favorable stride for most vector computers. In this case the cache efficiency is only about 0.414. The efficiencies for strides 16 to 256 are shown in Figure 1. This is obviously a very complicated function.

This curious phenomenon has been noted by others [2, 3, 4, 6]. One way to understand it is to list the cache addresses of consecutively fetched cache lines in a 128-long vector fetch, with stride 73, horizontally in a seven-wide table (see Table 1). This table also includes the notation `R` to indicate instances when a cache replacement would occur. It is clear from examining this table that the root cause of this poor performance is the very nearly periodic behavior of these cache addresses. In particular, these addresses are nearly periodic with a period of seven.

Recall that virtual address bits higher than position $r + w$ are ignored when placing the cache line in the cache. Thus we may in general write the cache address Q of the k-th

4	9	13	18	22	27	31
4	9	13	18	22	27	31
4	9	13	18	22	27	31
4	8	13	18	22	27	31
4R	8	13R	18R	22R	27R	31R
4R	8	13R	17	22R	27R	31R
4R	8	13R	17	22R	27R	31R
4R	8R	13R	17	22R	26	31R
4R	8R	13R	17	22R	26	31R
4R	8R	13R	17R	22R	26	31R
3	8R	13R	17R	22R	26	31R
3	8R	13R	17R	22R	26R	31R
3	8R	12	17R	22R	26R	31R
3	8R	12	17R	22R	26R	31R
3R	8R	12	17R	21	26R	31R
3R	8R	12	17R	21	26R	31R
3R	8R	12R	17R	21	26R	30
3R	8R	12R	17R	21	26R	30
3R	8R					

Table 1: Cache addresses for successive fetches when $S = 73$. Successive fetches are listed along rows, in a table seven wide, so that the nearly periodic behavior can be observed.

word fetched as

$$Q(k) \;=\; \text{int}\left[\frac{1}{W} \bmod (kS, RW)\right]$$

where int denotes the greatest integer function, and where mod denotes the modulo operation (i.e. the remainder when the first argument is divided by the second). The function $Q(k)$ is precisely periodic with period RW. But when the stride S is exactly (or very nearly) a simple fraction of RW, then this function is also precisely (or very nearly precisely) periodic with period $\text{nint}(RW/S)$, where nint denotes the nearest integer function.

In this example, $R = 32,\ W = 16,\ RW = 512$ and $S = 73$. Indeed, the fraction $512/73$ is very close to seven. In fact, $7 \times 73 = 511$, so that consecutive values of $\bmod(7kS, RW)$ differ by only one. Thus it clear, by examining the above formula, that $Q(7k)$ is identical for $W = 16$ consecutive k. But a string of 16 consecutive identical cache addresses results in 12 replacements, since only four of these can be accommodated in a single associativity set of the cache. This explains why the fetches in a single column of Table 1 result in a cache replacement approximately 75 percent of the time. Since this analysis applies to each column of the array shown in Table 1, it follows that the 75 percent figure also applies to the entire table as well.

From these facts one can compute the approximate cache efficiency E for this example (recall that the cache efficiency was defined above as the fraction of cache lines that remain in the cache when the vector fetch is complete). In Table 1, the first $4 \times 7 = 28$ fetches completely fill cache addresses 4, 9, 13, 18, 22, 27 and 31, except that address nine has one line empty. Thereafter approximately $3/4$ of the fetches result in a replacement. Thus we have the approximation

$$E \;=\; \frac{128 - (3/4) \times (128 - 28)}{128} = \frac{53}{128} = 0.4140625$$

which in this case exactly matches the actual efficiency determined by counting replacements in Table 1.

As we have seen, the replacement frequency $G = 3/4$ used in the above calculation results from the fact that $7 \times 73 = 511$ differs from 512 by only one. In general, define the minimum difference D as follows:

$$D \;=\; \min_{0<a,b<R} |bS - aRW|$$

When D is zero (i.e. when S is a large power of two, such as 64), then the corresponding value of G may easily be seen to be unity. When $D = 1$, then $G = 3/4$; when $D = 2$, then $G = 1/2$; when $D = 3$, then $G = 1/4$; and when $D \geq 4$, then $G = 0$. In other words, when D is larger than the set associativity size C, then successive fetches move to a different cache address before a given associativity set is exhausted. In general, the replacement frequency G is given by the formula

$$G \;=\; \frac{1}{C} \max(C - D, 0)$$

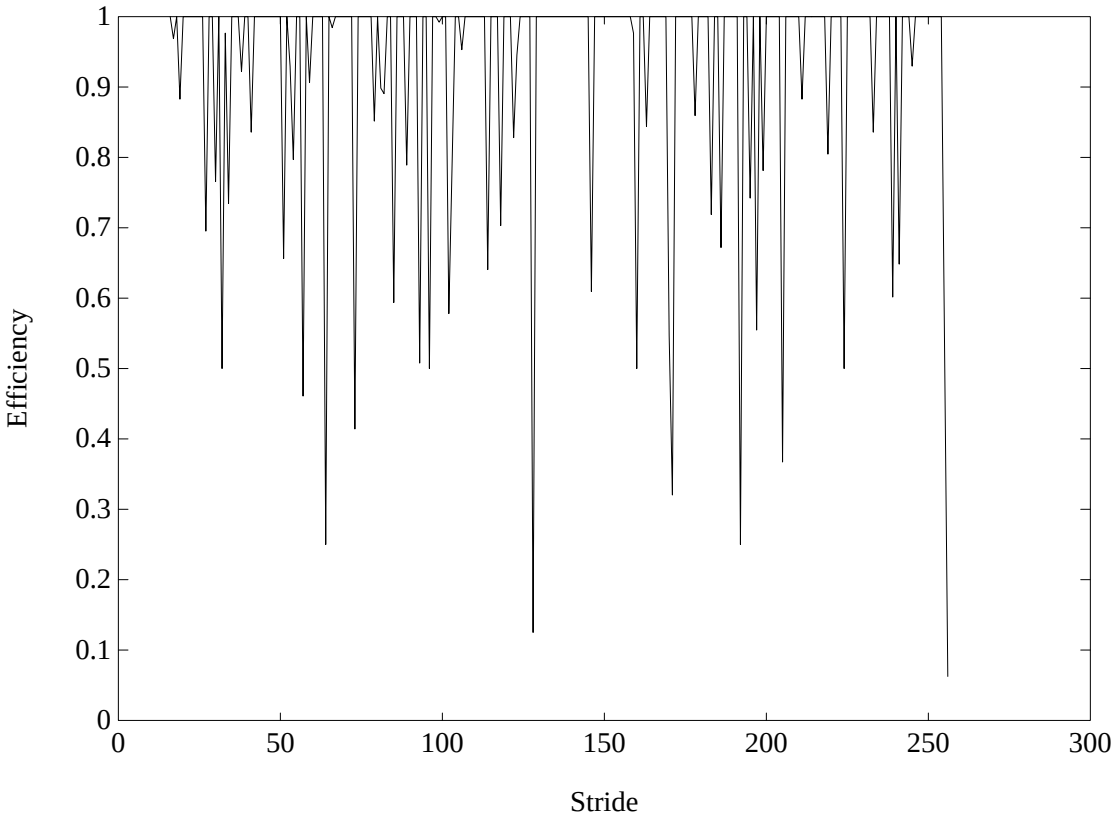

Figure 2: Cache Efficiencies Using the Formula

Suppose that $S/(RW)$ is very close to a simple fraction a/b, $b \leq R$, so that $D = |bS - aRW|$ is small. Compute G from the above formula. Generalizing from the above example, note that the first bC fetches will completely fill the b associativity sets whose addresses are those that nearly repeat. Thereafter, the fraction G (approximately) of the fetches will result in replacements. Thus a general formula that is an approximation to the cache efficiency E for general strides and cache parameters is given by

$$E = \frac{L - G(L - bC)}{L}$$

Note that when L is large, $E = 1 - G$ as expected.

A graph of the efficiencies for various strides in the standard case used above, computed with the above formula, is shown in Figure 2. By comparing Figures 1 and 2, it is clear that this formula is very accurate, particularly at the "spikes", which are the cases of greatest interest. In fact, the replacement count $G(L - bC)$, which is the key subexpression of this formula, is (with one exception) always within one of the actual value whenever G is nonzero.

3. A Random Stride Approximation

When the difference D is greater than C, the formula above gives perfect efficiency, since G in that case is zero. However, the actual efficiency is somewhat less than unity for many such cases, resulting in a low-level background "noise" (compare Figures 1 and 2). This phenomenon can be explained by nothing that when the stride S is a substantial fraction of RW, the operation $\mathrm{mod}(kS, RW)$ is a good pseudorandom number generator, and a certain number of "collisions" can be expected to occur in the resulting cache addresses. In fact, this operation is a member of the widely studied class of linear congruential pseudorandom number generators ([5], p. 9).

If one assumes that the assignment of memory fetches to the R addresses is actually random, then one can compute the expected cache efficiency by applying techniques of probability and statistics. The probability $P(k)$ that an individual address contains exactly k entries after an L-long fetch is given by the formula for a binomial distribution:

$$P(k) \;=\; \left(\begin{array}{c} L \\ k \end{array} \right) p^k (1-p)^{L-k}$$

where $p = 1/R$. The expected number of replacements F is then

$$F = R \sum_{k=C+1}^{L} (k - C) P(k)$$

and the resulting expected efficiency $E = (L - F)/L$. For the example parameters above, this formula yields $E = 0.807714\cdots$. The actual average efficiency, determined from the data in Figure 1, is $0.892334\cdots$. This indicates that the operation $\mathrm{mod}\,(kS, RW)$ actually behaves somewhat better than a true random number generator.

4. Finding Simple Fractions

One detail was omitted from the above discussion: how can one compute the minimum difference D for a given stride, or in other words, how does one determine the best simple fraction approximation a/b to $S/(RW)$? The straightforward scheme of computing $|bS - aRW|$ for all pairs of integers a and b less than R, in order to find the minimum value of this expression, is time-consuming when R is even moderate in size.

A more direct and elegant means to find these rational approximations a/b is to employ the Euclidean algorithm ([5], p. 319), as follows. Start with the 2-long vector $V = (S,\ RW)$ and the 2×2 identity matrix. At a given step let x be the smaller entry of V, let y be the larger entry, and let X and Y be the columns of the 2×2 matrix corresponding to x and y. Compute $q = \mathrm{int}(y/x)$. Then replace y by $y - qx$ and X by $X + qY$. This process continues until one entry of the vector V is zero. At that point one column of the final matrix will contain the original vector (with any common factor divided out) and the other column will contain a close rational approximation. In this application, the Euclidean algorithm may be halted whenever an entry of the matrix exceeds R.

The operation of this algorithm in this application is more easily understood by an example. Let us consider the particular parameters as above, with the stride $S = 197$. In

other words, we wish to find a good simple fraction approximation a/b to 197/512. The algorithm proceeds as shown below. The value of q used in each step (computed from the previous step's vector) is shown at the right.

$$\begin{pmatrix} 197 \\ 512 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}$$

$$\begin{pmatrix} 197 \\ 118 \end{pmatrix} \quad \begin{pmatrix} 1 & 0 \\ 2 & 1 \end{pmatrix} \qquad q = 2$$

$$\begin{pmatrix} 79 \\ 118 \end{pmatrix} \quad \begin{pmatrix} 1 & 1 \\ 2 & 3 \end{pmatrix} \qquad q = 1$$

$$\begin{pmatrix} 79 \\ 39 \end{pmatrix} \quad \begin{pmatrix} 2 & 1 \\ 5 & 3 \end{pmatrix} \qquad q = 1$$

$$\begin{pmatrix} 1 \\ 39 \end{pmatrix} \quad \begin{pmatrix} 2 & 5 \\ 5 & 13 \end{pmatrix} \qquad q = 2$$

$$\begin{pmatrix} 1 \\ 0 \end{pmatrix} \quad \begin{pmatrix} 197 & 5 \\ 512 & 13 \end{pmatrix} \qquad q = 39$$

In this case the desired pair of integers (a, b) is in the next-to-last column generated in the matrix, i.e. (5, 13). Note that $5/13 = 0.384615\cdots$ is indeed an excellent approximation to $197/512 = 0.384765\cdots$.

Here the final column generated, $(197, 512)$, is identical to the original vector. If S is divisible by a power of two, then the final column generated will be the original vector with the common power of two divided out. In that case, and if both entries of the final column are less than or equal to R, then this final column should be selected for (a, b) instead of the previously generated column. If for a given stride S, no pair (a, b), $b \leq R$ is found that satisfies $|bS - aRW| < C$, then the periodic effect does not exist, and the stride may be considered a favorable stride.

In this particular example, where $S = 197$, the resulting values $a = 5$ and $b = 13$ yield $D = 1$, so that $G = 0.75$ and $E = 0.5546875$.

5. Improving Cache Performance of Data Access with Strides

We have demonstrated a fairly simple scheme that can accurately predict the phenomenon of unusual slow-downs for particular strides. It should be emphasized, however, that the above analysis and conclusions depend on the particular model assumed above for an associative cache. This model is satisfied by many, but not all, of the currently popular RISC systems.

What can a programmer do if his or her program features a particularly unfavorable stride? The most straightforward solution is to "pad" (slightly increase) the dimensions of arrays having such dimensions. This solution has the advantage that in most cases only dimension statements need to be changed, and the executable part of the program does not need to be altered. Some space is "wasted" in this manner, but the resulting performance improvement is almost certainly worth the additional memory required.

There does not appear to be a simple formula giving the optimal amount of padding for a given unfavorable stride (i.e. array dimension) S, but in practice it suffices to merely evaluate the efficiency function described above for $S + 1$, $S + 2$, etc. until an efficient stride is found. In examples the author has studied, it appears that a pad of only one or two is effective in most cases.

However, this type of tuning should not be necessary, nor should it be necessary for programmers to analyze whether their strides are unfavorable. By applying techniques such as those described in this paper, compilers should be able to detect unfavorable strides and automatically adjust the appropriate array dimensions. Such adjustments will need to be optional, since they technically depart from the Fortran-77 standard, but they will likely be welcomed by the majority of users who prefer the compiler to shield them from such unsavory features of the underlying architecture.

References

[1] D. H. Bailey, "Vector Computer Memory Bank Contention", *IEEE Transactions on Computers*, vol. C-36, no. 3 (Mar. 1987), p. 293-298.

[2] D. Callahan and A. Porterfield, "Data Cache Performance of Supercomputer Applications", *Proceedings of Supercomputing '90* (Nov. 1990), p. 564 – 572.

[3] J. Ferrante, V. Sarkar and W. Thrash, "On Estimating and Enhancing Cache Effectiveness", IBM T. J. Watson Research Center, P.O. 704, Yorktown Heights, NY 10598, August 1991. Presented at the Fourth Workshop on Languages and Compilers for Parallel Computing, August 7 – 9, 1991, Santa Clara, CA.

[4] A. H. Karp, "What You Don't Know Can Hurt You, or Machine Organization Can Affect Performance", Technical Report G320-3479, IBM Scientific Center, 1530 Page Mill Road, Palo Alto, CA 94304, October 1985.

[5] D. E. Knuth, *The Art of Computer Programming*, Addison Wesley, Menlo Park, 1981.

[6] M. S. Lam, E. E. Rothberg and M. E. Wolf, "The Cache Performance and Optimization of Blocked Algorithms", *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (April 1991).