Fourth Intern. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS IV)
Palo Alto, California, April 9-11, 1991

The Cache Performance and Optimizations of Blocked Algorithms

Monica S. Lam, Edward E. Rothberg and Michael E. Wolf
Computer Systems Laboratory
Stanford University, CA 94305

Abstract

Blocking is a well-known optimization technique for improving the effectiveness of memory hierarchies. Instead of operating on entire rows or columns of an array, blocked algorithms operate on submatrices or *blocks*, so that data loaded into the faster levels of the memory hierarchy are reused. This paper presents cache performance data for blocked programs and evaluates several optimizations to improve this performance. The data is obtained by a theoretical model of data conflicts in the cache, which has been validated by large amounts of simulation.

We show that the degree of cache interference is highly sensitive to the stride of data accesses and the size of the blocks, and can cause wide variations in machine performance for different matrix sizes. The conventional wisdom of trying to use the entire cache, or even a fixed fraction of the cache, is incorrect. If a fixed block size is used for a given cache size, the block size that minimizes the expected number of cache misses is very small. Tailoring the block size according to the matrix size and cache parameters can improve the average performance and reduce the variance in performance for different matrix sizes. Finally, whenever possible, it is beneficial to copy non-contiguous reused data into consecutive locations.

1 Introduction

Due to high level integration and superscalar architectural designs, the floating-point arithmetic capability of microprocessors has increased significantly in the last few years. Unfortunately, the increase in processor speed has not been accompanied by a similar increase in memory speed. To fully realize the potential of the processors, the memory hierarchy must be efficiently utilized.

While data caches have been demonstrated to be effective for general-purpose applications in bridging the processor and memory speeds, their effectiveness for numerical code has not been established. A distinct characteristic of numerical applications is that they tend to operate on large data sets. A cache may only be able to hold a small fraction of a matrix; thus even if the data are reused, they may have been displaced from the cache by the time they are reused.

This research was supported in part by DARPA contract N00014-87-K-0828.

1.1 Blocking

Consider the example of matrix multiplication for matrices of size $N \times N$:

```
for i := 1 to N do
  for k := 1 to N do
    r = X[i,k]; /* register allocated */
    for j := 1 to N do
      Z[i,j] += r*Y[k,j];
```

Figure 1(a) shows the data access pattern of this code. The same element *X[i,k]* is used by all iterations of the innermost loop; it can be register allocated and is fetched from memory only once. Assuming that the matrix is organized in row major order, the innermost loop of this code accesses consecutive data in the Y and Z matrices, and thus utilizes the cache prefetch mechanism fully. The same row of Z accessed in an innermost loop is reused in the next iteration of the middle loop, and the same row of Y is reused in the outermost loop. Whether the data remains in the cache at the time of reuse depends on the size of the cache. Unless the cache is large enough to hold at least one $N \times N$ matrix, the data Y would have been displaced before reuse. If the cache cannot hold even one row of the data, then Z data in the cache cannot be reused. In the worst case, $2N^3 + N^2$ words of data need to be read from memory in N^3 iterations. The high ratio of memory fetches to numerical operations can significantly slow down the machine.

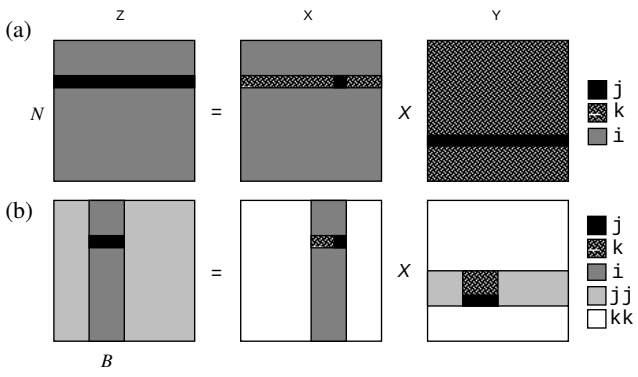


Figure 1: Data access pattern in (a) unblocked and (b) blocked matrix multiplication.

It is well known that the memory hierarchy can be better utilized if scientific algorithms are *blocked* [1, 5, 6, 8, 10, 11, 12].

Blocking is also known as tiling. Instead of operating on individual matrix entries, the calculation is performed on submatrices. Blocking can be applied to any and multiple levels of memory hierarchy, including virtual memory, caches, vector registers, and scalar registers. As an example, when blocking is applied at both the register and cache levels, we observe that matrix multiplication speeds up by a factor of 4.3 on a DECStation 3100, and a factor of 3.0 on an IBM RS/6000, a machine with a relatively higher performance memory subsystem. The matrix multiplication code blocked to reduce cache misses looks like this:

```
for kk := 1 to N by B do
  for jj := 1 to N by B do
    for i := 1 to N do
      for k := kk to min(kk+B-1, N) do
        r = X[i,k]; /* register allocated */
        for j := jj to min(jj+B-1, N) do
          Z[i,j] += r*Y[k,j];
```

Figure 1(b) shows the data access pattern of the blocked code. We observe that the original data access pattern is reproduced here, but at a smaller scale. The *blocking factor*, B, is chosen so that the $B \times B$ submatrix of Y and a row of length B of Z can fit in the cache. In this way, both Y and Z are reused B times each time the data are brought in. Thus, the total memory words accessed is $2N^3/B + N^2$ if there is no interference in the cache.

Blocking is a general optimization technique for increasing the effectiveness of a memory hierarchy. By reusing data in the faster level of the hierarchy, it cuts down the average access latency. It also reduces the number of references made to slower levels of the hierarchy. Blocking is thus superior to optimizations such as prefetching, which hides the latency but does not reduce the memory bandwidth requirement. This reduction is especially important for multiprocessors since memory bandwidth is often the bottleneck of the system.

Blocking has been shown to be useful for many algorithms in linear algebra. For example, the latest version of the basic linear algebra library (BLAS 3) [4] provides high-level matrix operations to support blocked algorithms. LAPACK [2], a successor to LINPACK, is an example of a package built on top of the BLAS 3 library.

Previous research on blocking focused on how to block an algorithm manually and automatically [5, 7, 11, 12]. The procedure consists of two steps [12]. The first is to restructure the code to enable blocking those loops that carry reuse, and the second is to choose the blocking factor that maximizes locality. It is the latter step that is sensitive to the characteristics of the level of memory hierarchy in question. Since the benefit of blocking increases with the block size, previous approaches suggested choosing blocking factors such that the faster memory hierarchy is fully occupied by data to be reused. For example, the optimal blocking factor is roughly $\sqrt{C}$ for matrix multiplication on a machine with a local memory of C words [9]. This is appropriate for registers or local memories, where the data placement is fully controlled. This is also a reasonable approach for fully associative caches with a least recently used (LRU) replacement policy.

1.2 Impact of Cache Behavior on Blocking

In practice, caches are direct mapped or have at most a small degree of set associativity. The address mapping may map multiple rows of a matrix to the same cache lines, making it infeasible to try to fully use the cache. This address mapping has a significant effect on the performance of blocked code, causing it to deviate from the simple trend of increased performance with increased block size. Moreover, this performance varies drastically with small changes to the matrix size.

Shown in Figure 2(a) is the performance of blocked matrix multiplication on a DECstation 3100. For reference, an unblocked matrix multiplication achieves roughly 0.9 MFLOPS. The DECstation 3100 has an 8K double word direct-mapped cache and can hold all the words reused within an 88×88 block. The graph plots the performance levels obtained for three slightly different matrix sizes across a range of blocking factors. We use two different codes; one blocks for both the cache and registers [3], while the other blocks only for the cache. While the performance curves for the 300×300 matrix multiplication are well behaved, those for the other two drop sharply starting at different blocking factors depending on the matrix size. More significant is the magnitude of the variation for similarly sized matrices. Matrix multiplication using a 56×56 block for a 300×300 matrix runs at twice the rate of that of a 293×293 matrix.

The variation in performance is due to the interference misses in the cache. Shown in Figure 2(b) are the miss rates of the runs for the cache-blocked code of Figure 2(a), obtained via simulation. The decrease in performance correlates perfectly with the increase in cache misses. These two sets of data suggest that the behavior of a cache has a major impact on blocked code performance, and must be considered when choosing the size of the block.

1.3 Paper Overview

Our research focuses on optimizing cache performance via blocking. The approach is to first discover the behavior of caches under blocking, then to improve its performance via software and/or hardware techniques. The sensitivity of the miss rates to the size of the input matrix makes it impossible to use a purely experimental approach. It is inadequate to simulate a sample of data points and infeasible to simulate all possibilities. Our methodology is to combine theory and experimentation together in understanding the behavior of the cache. Drawing insights from the experimental data and the theory of data locality from our compiler research [12], we have developed a model of data conflicts that has been validated by simulating several representative data points. Using this model, we are able to explain the cache performance observed, derive the performance of all possible data sizes, evaluate existing methods on how to choose a block size and propose new methods and optimizations that can fully utilize a cache.

2 Data Locality in Blocked Algorithms

In this section, we present our cache model for the simple case of a direct-mapped cache with one-word cache lines, and illustrate the model with blocked matrix multiplication. The extension to set-associative caches and multiple-word line sizes is described in Section 5.

The *reuse* of a reference is *carried* by a loop if the same memory locations or cache lines are used by different iterations of that loop. There are two forms of reuse: *temporal* and *spatial* reuse. Temporal reuse occurs when the same data are reused. For example, in matrix multiplication, the temporal reuse of variables X, Z and Y are carried by the innermost, middle and outermost loops respectively. In this case, each variable is reused N times, where N is the size of each loop. We say that N is the reuse factor. Spatial reuse occurs when data in the same cache line are

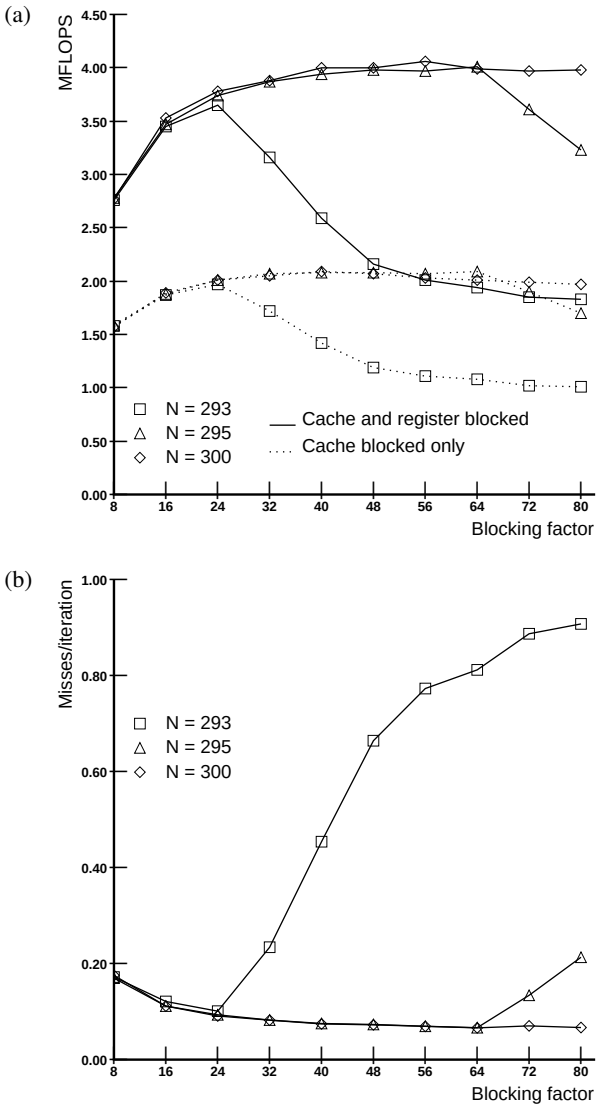


Figure 2: (a) Performance and (b) miss rates for blocked matrix multiplication, DECstation 3100

used. For a cache with line size l, the reuse factor is l if the data is accessed in a stride one manner.

Reuse of data translates to a saving in memory accesses only if intervening references between reuse do not displace the data from the cache. If the iteration count of the innermost loop is large, only reuse within the innermost loop can be exploited. Blocking *localizes* iterations across the outer dimensions by limiting the intervening iterations executed from the innermost loop so that cached data is reused before it is replaced. By choosing the blocking factor suitably, reuse carried by all loops within a block can be exploited. The temporal reuse factor of data within a blocked loop is simply the blocking factor, or the number of iterations in a blocked loop.

2.1 Modeling Cache Interference

If all the data to be reused map to different cache locations, then the number of cache misses per variable is simply $D(v)/R(v)$, where $D(v)$ is the total number of memory references for variable v and $R(v)$ is the reuse factor of variable v. These are the *intrinsic misses*. Generally, we also have interference misses; if the reuse of the variable v misses at a rate of $M(v)$, then the total number of misses for v is

$$D(v)\left(\frac{1}{R(v)} + \frac{R(v)-1}{R(v)}M(v)\right) \tag{1}$$

A reused variable will miss in the cache if any of the memory references between reuse occupies the same cache location. We assume in this simple model that the interference is independent. Suppose the locality of variable v is carried by loop p, and let V be the set of variables used in the loop. The miss rate in reusing v within loop p is

$$M_p(v) = 1 - \prod_{u \in V} \big(1 - I_p(u,v)\big),$$

where $I_p(u,v)$ is the probability that accesses to data u within one iteration of loop p interferes with the reuse. We partition interferences into two cases: *cross interference*, interference between two different variables, and *self interference*, interference between elements of the same array variable.

In the case of cross interference, we assume that the location of reuse is unrelated to the the cache locations of the interfering accesses. We estimate the interference by the probability that the location of reuse falls in the *footprint* of the variable. The footprint of a variable $F_p(u)$ for loop p is defined to be the fraction of the cache used by variable u in one iteration of loop p. This footprint measures the number of distinct elements of u used in one iteration of loop p if these elements map to unique positions in the cache. If u is accessed in a stride-one manner, uniqueness is guaranteed unless the total size exceeds the cache size.

In the case of self interference, we can no longer ignore the positioning of the reuse location and those of other elements within the same array. There are two common cases. If the accesses are of stride one, then no interference is possible as long as the number of data accessed is smaller than the cache capacity. Otherwise, accesses to the other elements in the same array can significantly interfere with the reuse. The self interference $S_p(v)$ is defined to be the fraction of accesses that map to non-unique locations in the cache within one iteration of loop p.

In sum, the miss rate on the reuse of data v is

$$M(v) = 1 - \big(1 - S(v)\big) \prod_{u \in V - \{v\}} \big(1 - F(u)\big) \tag{2}$$

Table 1: Matrix multiplication parameters.

	Reuse R_p			Self-Interference S_p			Footprint F_p			References
	i	k	j	i	k	j	i	k	j	
X	B	-	-	0	-	-	-	$1/C$	B/C	N^3/B
Z	-	B	-	-	0	-	-	-	$2B/C$	N^3
Y	-	-	N	-	-	$S_i(Y)$	-	B/C	-	N^3

It is straightforward to extract the parameters of the reuse factors and the footprints from the code of a blocked algorithm. The self interference term, however, can depend on the cache size and the input matrix size, and is the factor that makes the cache behave erratically.

2.2 Extracting the Model Parameters

Matrix multiplication is an interesting case study because locality is carried in three different loops by three different variables. Similar reuse patterns can be observed in various other important matrix operations as well, including Gaussian Elimination (without pivoting) and Cholesky Factorization.

All the relevant parameters of the matrix multiplication code are shown in Table 1. For example, the same element of Y is reused in loop i; between each reuse, B distinct words of *X* and $2B$ distinct words of Z would have been used, and hence the corresponding footprints. These parameters are easy to determine, with the exception of self interference for variable Y in loop i. We simply represent the term as $S_i(Y)$ and will show how to derive its value in the next section. Quantities that are of no interest for this discussion are left blank.

The miss rates for each of the variable can now be easily evaluated. Since the variable *X[i,j]* is allocated to a register, the total number of references to elements of the array is N^3/B and its miss rate is simply 0. Substituting the parameters in Table 1 into Equation 2, the miss rates for *Y* and *Z*, $M(Y)$ and $M(Z)$, are

$$\begin{aligned} M(Y) &= 1-\left(1-S_i(Y)\right)\left(1-\frac{2B}{C}\right)\left(1-\frac{B}{C}\right) \\ &\approx S_i(Y)+\frac{3\left(1-S_i(Y)\right)B}{C} \\ M(Z) &= 1-\left(1-\frac{1}{C}\right)\left(1-\frac{B}{C}\right)\approx\frac{B}{C} \end{aligned}$$

The total number of cache misses are therefore

$$\begin{aligned} &\frac{N^3}{B}+N^3\left(\frac{1}{N}+\frac{N-1}{N}M(Y)\right)+N^3\left(\frac{1}{B}+\frac{B-1}{B}M(Z)\right) \\ &\quad\approx N^3\left(\frac{2}{B}+S_i(Y)+\frac{3\left(1-S_i(Y)\right)B}{C}+\frac{B}{C}\right) \end{aligned} \tag{3}$$

According to this equation, there are $2N^3/B$ intrinsic misses, misses that are intrinsic to the algorithm given the blocking factor and cannot be avoided even if the address mapping is perfect. The factor $S_i(Y)$ is due to self interference of variable *Y* on itself. The other two terms are due to cross interference between different variables.

Thus far, we have modeled only the reuse within the block. Theoretically, variables X and Z can be reused in the first and second loop respectively. Reapplying the same modeling procedure, we can refine our estimate to reflect this level of reuse. The reuse is unlikely, however, except when the caches are large with respect to N and the block size is small, since this would entail the reused data surviving at least $2NB$ other references. Our equation, as it stands, will overestimate the number of misses under those circumstances.

3 Interference with Regular Stride

In this section, we examine self interference for the common case where an array is accessed at a constant non-unit stride. The reference pattern of a constant-stride array access is regular and so is its self interference pattern. Intuitively, the data accessed might not interfere with each other, but if they do, the interference is regular and severe. We show that self interference can increase the cache miss rate so drastically that it should be avoided altogether, if possible. We have developed an algorithm to find the largest blocking factor that avoids self interference for a given matrix size. Tailoring the block size according to the matrix size yields much better cache performance than trying to use a fixed block size.

3.1 Computing Self Interference Misses

We assume that the address of *Y[i,j]*, written *&Y[i,j]*, is $Y^* + Ni + j$, where Y^*, the start address of array *Y*, and N, the matrix size, are run-time constants. Two words use the same cache location when *&Y[i,j]* $\equiv$ *&Y[i',j']* $\pmod{C}$. (This is valid even for caches that use physical addresses, because many operating systems, including those of MIPS and SGI, allocate frames so that a physical address maps to the same cache location as its virtual address.) Therefore if *Y[i,j]* and *Y[i',j']* use the same location in the cache, then so must *Y[i+a,j+b]* and *Y[i'+a,j'+b]*.

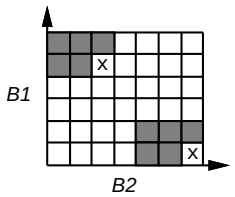


Figure 3: Self interference in a direct-mapped cache.

The interference that results from this pattern is shown in Figure 3. Suppose a $B1 \times B2$ block of the matrix is to be reused, but that the two words of the matrix marked with an 'x' fall in the same cache location. Then, because of periodicity, all of the shaded words of the array in the upper left hand portion of the block will also interfere with the corresponding shaded words in the lower right hand corner of the block. Likewise, if the word in the lower left hand corner of the block interferes with any other word, then a rectangle in the lower left hand corner will interfere with a rectangle in the upper right hand corner. In either case,

once two words in a block interfere with each other, then increasing the block size will lead to an increase in self interference. Consequently, the largest block size that does not suffer from any self interference, which we refer to as the critical blocking factor B_0, plays an important role in determining the total number of misses.

Based on these observations, we have developed an efficient algorithm that determines the amount of self interference given a matrix size, a cache size and the blocking factors. The algorithm executes in $O(N/\sqrt{C})$, where N is the matrix size and C is the cache size.

3.2 Analyzing the Cache Misses

With the algorithm above, we can determine the self interference parameter in Equation 3 and derive the predicted number of total cache misses. The overall cache miss rate is a combination of three kinds of misses: intrinsic misses, self-interference misses and cross-interference misses. Figure 4 shows a breakdown of the misses into the three categories for blocked matrix multiplication on a cache of 1K words.

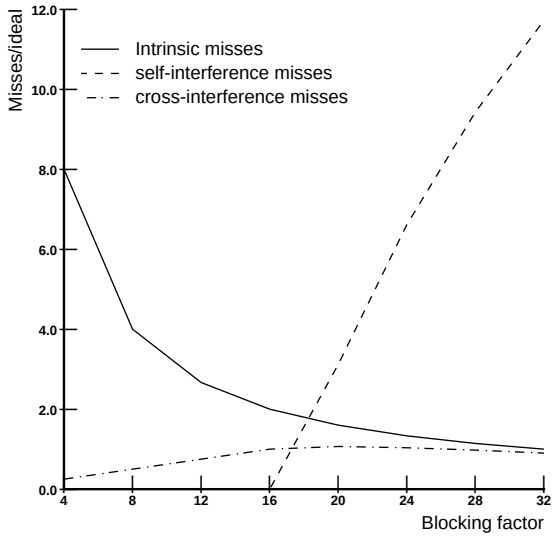


Figure 4: Breakdown of misses, 1K word cache, $N = 295$.

To highlight the effect of the address mapping function of a cache, the figure compares the cache miss numbers to those we would have obtained if the machine had a local memory of the same size. With a local memory, software has full control over data placement, and can thus use the storage more effectively. We refer to the misses occurring with a local memory as the "ideal" number of misses. For a local memory of size C, the optimal blocking factor is roughly $\sqrt{C}$.

The first component refers to the intrinsic misses for particular block sizes. From Section 2, we know that the number of intrinsic misses is inversely proportional to the blocking factor. As the blocking factor approaches $\sqrt{C}$, it reaches the ideal number of misses.

Self interference misses are misses incurred due to conflicts among the elements in the *Y* array. As discussed above, there is a critical blocking factor B_0 such that blocking factors greater than B_0 lead to significant self interference. Any blocking factor lower than B_0 causes no self interference. The critical blocking factor can be very different for similar matrix sizes and identical cache sizes.

The third component is cross interference, the interference between different arrays. The cross interference is a function of the footprint and is linear with respect to B. Since we classify a reused data item suffering from both self and cross interference as a self interference miss, the cross interference curve may appear to taper off after self interference begins.

The self interference term explains the cache behavior shown in Figure 2(b). The total cache miss curves decline smoothly with the blocking factor until the critical blocking factor is reached. Beyond the critical blocking factor, the self interference component dominates and causes the overall miss rate to rise sharply.

To validate our model, we compare the predicted cache misses to actual cache misses for a blocked matrix multiplication across different cache sizes and blocking factors (Figure 5). The figure shows that the model predicts the cache behavior accurately, except when the cache is large and the blocking factors are small, as discussed in Section 2.2.

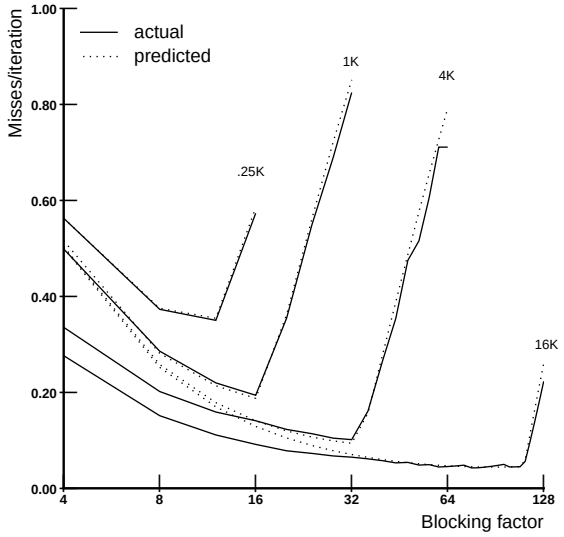


Figure 5: Actual versus predicted cache misses across different cache sizes (in words), $N = 295$.

3.3 Using a Fixed Blocking Factor

With our model, we are now ready to evaluate different strategies for choosing the blocking factor. Clearly, trying to use the entire cache, as we would in the case of a local memory, would be disastrous. Let us first evaluate the simple strategy of attempting to use only a fixed fraction of the cache.

Assuming that the matrix size, N, is randomly distributed, we compute an expected value for each possible blocking size by averaging the performance over all possible N's. The average value is not particularly interesting to a user per se since he or she is interested only in the performance for the particular matrix size used. Thus we include also the standard deviation to indicate the likelihood for achieving a particular miss rate.

Figure 6 shows the average cache behavior of matrix multiplication on a 1K word cache for a given blocking factor. The average is obtained by calculating the misses using our model for each N in the range C to $2C - 1$, which captures all possible self interference patterns in the cache, where C is the cache size. The

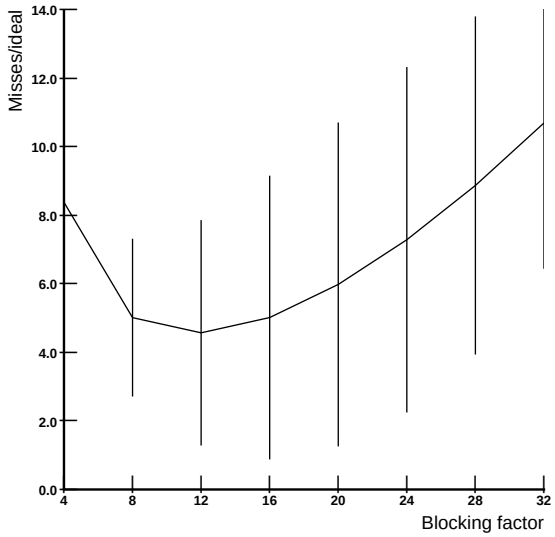


Figure 6: Average misses in 1K word cache. Vertical lines indicate the standard deviation.

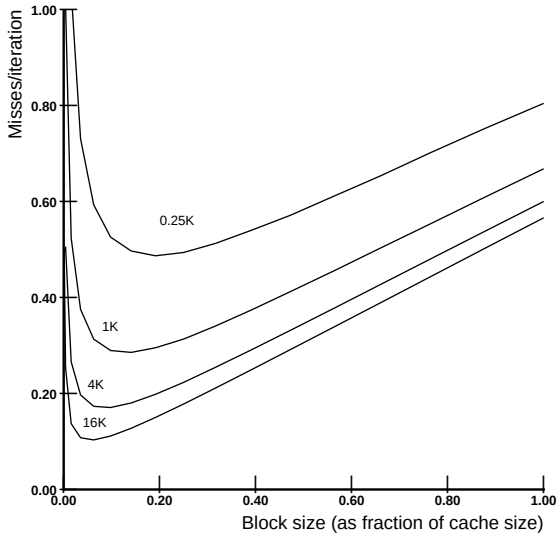


Figure 7: Miss rate (averaged over all matrix sizes) versus block size, across different cache sizes (in words).

average and standard deviation of the miss rates were calculated for blocking factor B from four to $\sqrt{C}$ in increments of four. If the block is chosen to use the entire cache, i.e. $B = \sqrt{C}$, the miss rate is more than 10 times the optimal, and has a large standard deviation. A blocking factor of 12 gives the best average; whereas a blocking factor of 8 has only a slightly higher average, but a substantially lower standard deviation. If we want to express the targeted block size as a fraction of the cache, the number is extremely small (6% or 14% for a blocking factor of 8 or 12, respectively).

The actual miss rates for a range of cache sizes are shown in Figure 7. Without blocking, the miss rate is approximately 2 misses per iteration if the cache cannot hold a few rows in the cache, and 1 miss per iteration if it can. Blocking reduces the miss rate to about an average of 0.5 and 0.1 misses per iteration for a relatively small (.25K word) and large (16K word) cache, respectively. This reduction typically translates to a significant impact on overall system performance.

Several unexpected conclusions can be drawn from these graphs. First, the trends of the curves are very different from the ideal. The ideal is simply an inverse function of the block factor; the larger the block, the better the performance. The curves in the figure indicate that increasing the blocking factor can in fact degrade performance significantly. The block size choice that minimizes the average cache miss rate uses a small fraction of the cache. The fraction of the cache used in the optimal case decreases with increasing cache sizes, from 15% for a 0.25K word cache to 3% for a 16K word cache. More importantly, as shown in Figure 6, there is a large standard deviation associated with the average case. This means that the execution time can vary significantly for different matrix sizes.

3.4 Tailoring the Blocking Factor

Since the miss rates are highly sensitive to the problem size, we now consider the approach of tailoring the blocking factor to the problem size. By doing so, we hope to improve the average miss rate and, more importantly, to reduce the variance.

From our model, we know that the cache miss rate hinges on the critical blocking factor B_0, the maximal factor with no self interference. Due to the periodicity in the addressing of a direct-mapped cache and the constant-stride accesses, it is relatively easy to determine B_0. The algorithm to determine the largest square block with no self interference is presented in Figure 8.

```
algorithm FindB(N,C: integer) return integer;
   addr,di,dj,maxWidth: integer;

   maxWidth := min(N,C);
   addr := N/2;
   while true do
      addr := addr + C;
      di := addr div N;
      dj := abs((addr mod N) - N/2);
      if di ≥ min(maxWidth,dj) then
         return min(maxWidth,di);
      maxWidth := min(maxWidth,dj);
   end while;
end algorithm;
```

Figure 8: Algorithm to compute the largest square block without self interference.

Our approach is based on a few simple observations. The self interference pattern of any block of computation is identical since the conflict between a pair of data depends only on the difference of their addresses. Similarly, if *&Y[i,j]* $\equiv$ *&Y[i+di,j+dj]* (mod C), then B_0 can be no larger than max($|di|, |dj|$). The algorithm begins with the array word *&Y[0,N/2]*, assuming $Y^* = 0$ (mod C), which is valid since the absolute location of the array in memory is not significant. The algorithm then finds which array words of the form *Y[di,N/2±dj]* are mapped to the same location in the cache. Each new array word mapping to the same location puts a further restriction on how large the block size may be before self interference begins. The blocking factor cannot be larger than $\sqrt{C}$, so the run time of the algorithm is $O(N/\sqrt{C})$, which

is fast enough to use in a compiler or at run time if the matrix size is not known statically. This algorithm can be easily extended to find the largest rectangular block.

The different maximal blocking factors for the entire range of strides are shown in Figure 9. The function is periodic; the interference is at its maximum if the matrix dimension is a multiple of the cache size. This figure explains why the optimal fixed block size is so small. If B_0 is large, then there is a diminishing return for choosing larger and larger block sizes. However, if B_0 is small, then the self interference becomes very large even for moderate block sizes. The data indicate that the critical blocking factor is sensitive to small changes in the matrix dimension N. Moreover, for many array dimensions, a fairly large blocking factor can be used. Choosing a fixed block size to optimize the average case penalizes those cases for which a good blocking factor exists.

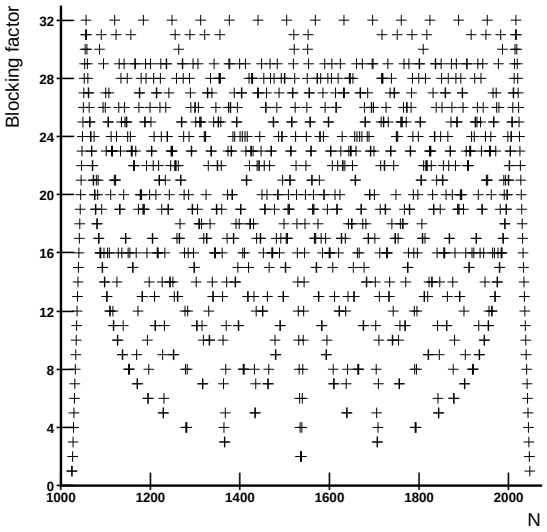


Figure 9: The largest square block without self interference, 1K word cache.

When there is no self interference, the number of misses from Equation 3 becomes $N^3(2/B + 4B/C)$, which has a minimum at $B = \sqrt{C/2}$. Thus, even if B_0 exceeds $\sqrt{C/2}$, cross interference limits the optimal blocking factor to $\sqrt{C/2}$. In this case, the number of misses is $2\sqrt{2}$ or 2.8 times the ideal number of misses. If we had used a blocking factor of $\sqrt{C}$, the misses would be three times the optimal even if there were no self interference. If self interference begins before $\sqrt{C/2}$, the self interference will dominate the cache behavior, in which case it can be shown that the optimal blocking factor is typically B_0 but also can be $B_0 + 1$.

Figure 10 shows the ratio of misses for each N with the best block size to the ideal number of misses for that cache size. For this graph the block size for a given N is the block size from Figure 9 or $\sqrt{C/2}$, whichever is smaller.

We can average over the number of misses in Figure 10 to determine the average performance obtained by choosing a block size that depends on N. The average is 3.4 ± 2.1 times the ideal. If instead a blocking factor of 12 were always used (the best choice from Figure 7), then the number of misses is 4.6 ± 3.3 of ideal. Both the mean and the standard deviation of misses are significantly lower when block sizes are chosen using N. Figure 11 shows the results of this analysis for a variety of cache sizes. For comparison, Figure 11 also includes data for the best data-dependent rectangular block without self interference. It shows that best square/rectangular schemes outperform the fixed block size approach in terms of both the average misses and variance over a wide range of cache sizes. As the cache gets larger, the miss rate for the best fixed block size gets farther and farther from ideal, with an increasing standard deviation. However, if N is used in considering the block size, then on average the behavior of the block remains within a constant of the optimal.

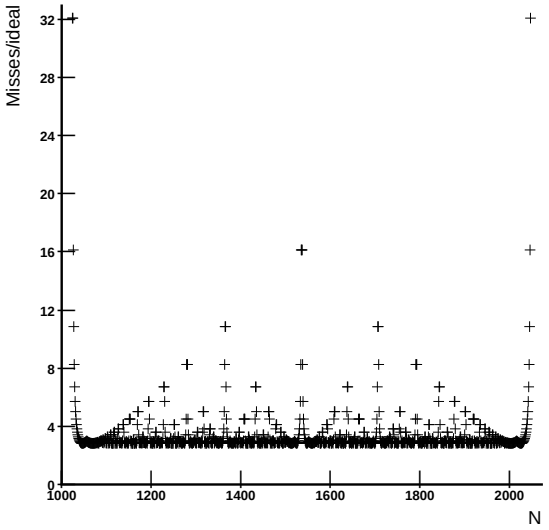


Figure 10: Miss rates for best square block, 1K word cache.

4 Self Interference for Non-Constant Strides

After analyzing the cache behavior with constant stride data accesses, we now turn to other access patterns. We first study the cache behavior for triangular matrices, then pushing the irregularity of data access to the limit, we consider the case when the stride is totally random.

4.1 Triangular Matrices

A typical data organization of a triangular matrix is to store the rows of data consecutively. The stride of a column access varies from 1 to N where N is the matrix dimension. Thus we expect that the cache behavior of the entire multiplication to be similar to the average of cache misses obtained for square matrices.

In Figure 12, we compare the cache miss figures for multiplying two triangular matrices with 1000 elements on a side with the average obtained for multiplication of square matrices for the entire range of strides. Indeed, the simulation confirms the prediction.

Unlike square matrices, the cache performance of a blocked triangular matrix multiplication algorithm is predictable for any array size, provided that it is relatively large. That is, the performance obtained is similar to the average we compute for square matrices. The optimal block size is small; about 10% of the cache for a 1K cache, and 3% for a 16K cache. Since the miss rate function is highly sensitive to the choice of the blocking factor, it is important to choose this number correctly. Since the variance in optimal block size is high even for small changes in N, it is

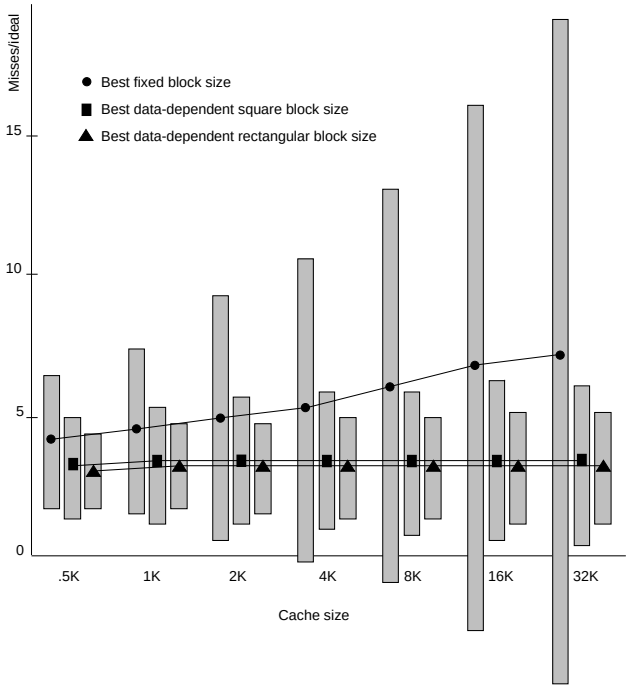


Figure 11: Miss rates using best fixed block size, best data-dependent square block size, and best data-dependent rectangular block size.

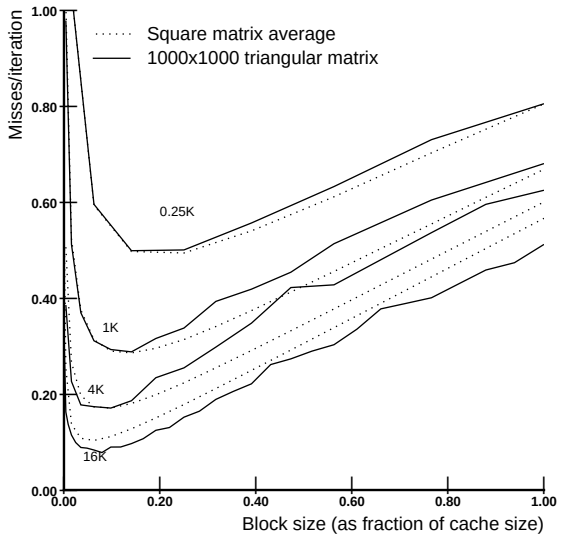


Figure 12: Comparing miss rates of triangular matrix multiplication with the average miss rates for square matrix multiplication, across different cache sizes (in words).

difficult to improve the cache performance by modifying the block size for different parts of the triangular matrix.

4.2 Random Stride

Given that we have shown caches may behave rather poorly for regular strides, we wish to investigate if they would behave any better if the rows were randomly placed. We model the accesses to each row in the same array as independent accesses. Each access to B consecutive locations of data decreases the hit rate by a factor of $1 - B/C$. Thus the self interference of accessing $B - 1$ rows of Y before the same data is reused is

$$S_i(Y) = (1 - \frac{B}{C})^{B-1}.$$

Substituting this self interference term into Equation 3 and varying the cache and block sizes, we obtain the data in Figure 13. Differentiating the expression of misses with respect to the blocking factor B, we find that the asymptotic minimum number of cache misses occurs when B is $\sqrt[3]{C}$. The fraction of the cache used is thus $C^{\frac{2}{3}}$. The cache use decreases from about 12% in the 1K cache case to about 4% in a cache of 16K words. These figures are just slightly lower than those obtained for the average for square matrices.

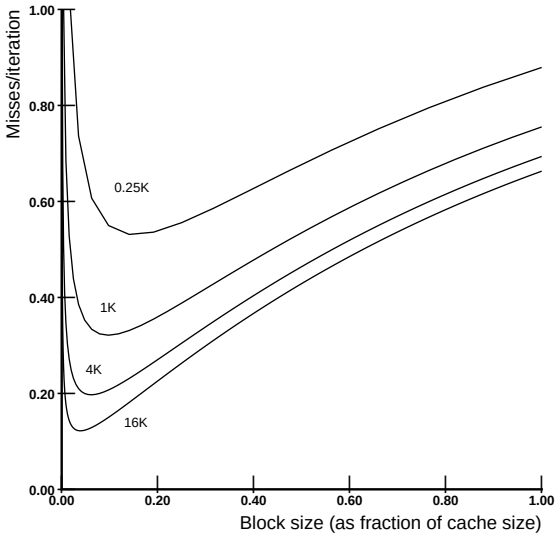


Figure 13: Miss rate versus block size for random stride, across different cache sizes (in words).

In summary, when accesses are random, reuse of data is unlikely unless the number of intervening independent data accesses is small. Constant but non-unit stride accesses found commonly in matrix operations do not in general improve the cache performance. However, if the stride is regular, it is possible for a compiler or run-time library to tailor the blocking factor according to the stride. The improvement is significant but there is still a relatively high variance in cache misses for different matrix sizes.

5 Different Cache Parameters

So far in this paper we have considered only direct-mapped caches with single-word cache lines. In this section we consider two common variations in cache design, a higher degree of associativity

and longer cache lines, and discuss how they affect the cache behavior of blocked algorithms. We re-apply the same techniques used in analyzing direct-mapped caches with single-word lines and use the resulting models to generate data for different cache sizes, matrix sizes, and block sizes. Again, we compare the performance of the strategies of choosing a data independent block size and a data dependent block size. We then draw conclusions about the impact of set associativity and multiple-word line sizes on blocked algorithms.

5.1 Set Associativity

Since conflict misses play a significant part in the cache behavior of blocked algorithms, it is natural to examine if set associativity can reduce the misses significantly. We consider a relatively large but reasonable set associativity of four. The interference model presented in Section 2 extends readily to set-associative caches. Briefly, with an a-way set-associative LRU cache, a set holding more than a block entries reused at the same level is assumed to miss on all accesses to those entries; otherwise, they all hit. The validity of our model is shown in Figure 14(a). As before, the model overpredicts misses when the cache size is large and the block size is small.

Figure 14(b) shows the performance of a set-associative cache if a fixed block size is chosen for all problem sizes. Unlike the direct-mapped cache, the average miss rates remain relatively flat as the block size increases. This is because the critical blocking factors now have much larger values. Although the fraction of cache used remains small, it is larger than that of the direct-mapped cache and yields a significantly lower average miss rate. While the average remains fairly flat, the standard deviation (not shown in the graph) increases steadily with the block size, thus favoring a smaller block size. This high standard deviation means that the execution time for some problem sizes can be significantly worse than others.

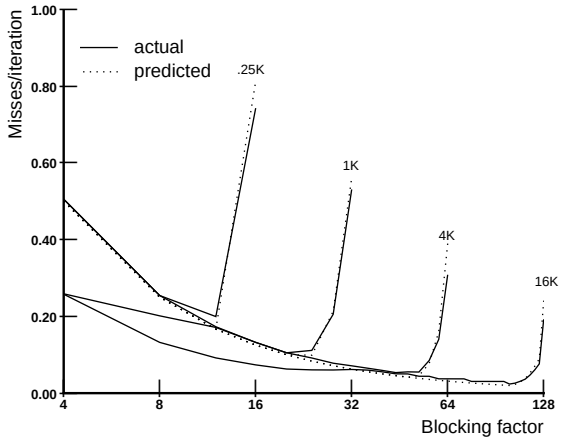
Figure 14(c) focuses on the averages and standard deviations of the miss rates for a variety of cache sizes. Again, we compare two strategies for choosing block sizes: the fixed square block scheme and the problem size dependent square block scheme [1]. While the average ratio to the ideal increases for the fixed block size scheme, it remains quite constant when we vary the block sizes for different problem sizes. More importantly, it has a much smaller standard deviation. If we take the strategy of choosing our block sizes based upon N, an associativity of 4 allows us to drop the miss rate by an average of over 30% and cut the standard deviation in half relative to the same strategy on a direct-mapped cache.

5.2 Line Size

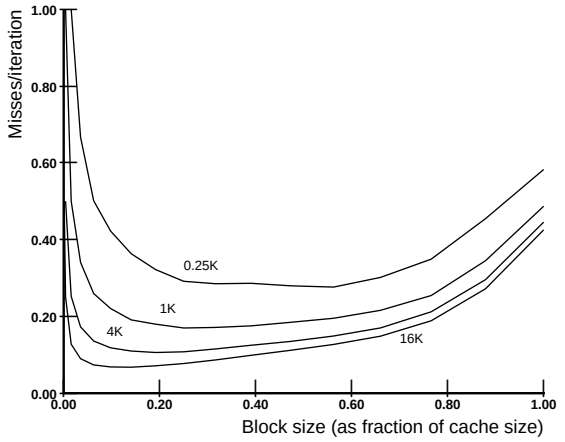
To exploit spatial locality, most caches today have multi-word cache lines. Spatial locality can be modeled in a manner similar to temporal locality. We say that spatial locality is carried by a loop if the stride of data access is less than the cache line size. The reuse factor is simply the cache line size divided by the stride. The miss rate can be calculated from the footprint and self interference in a similar manner, although the footprint and self interference

[1] In the direct-mapped case, the model shows that the best square block is not larger than $\sqrt{C/2}$ on a side. The set-associative model cannot make as precise a prediction. We use the heuristic that the blocking factor should not be larger than $\sqrt{Ca/(a+1)}$ if the cache has associativity a, since cross interference diminishes with increasing set associativity.

(a) Actual versus predicted cache misses across different cache sizes (in words), $N = 295$.



(b) Miss rate (averaged over all matrix sizes) versus block size, across different cache sizes (in words).



(c) Miss rates using best fixed block size and best data-dependent square block size.

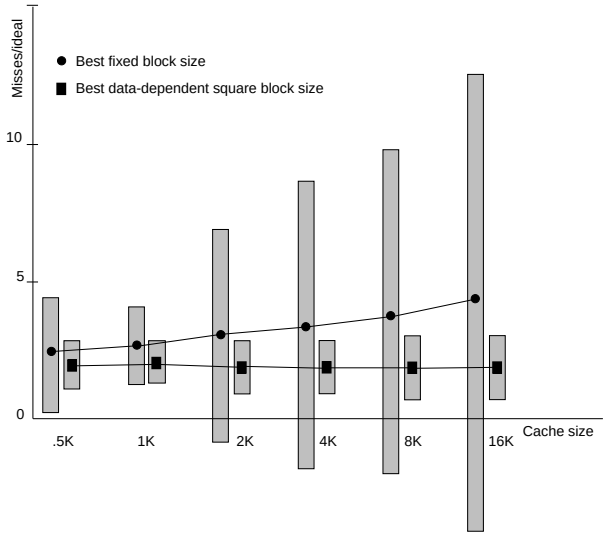
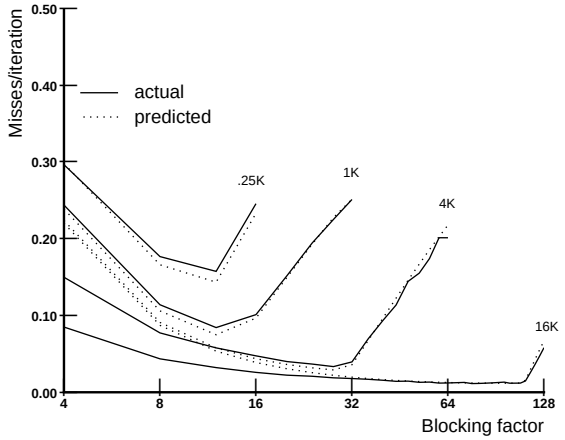
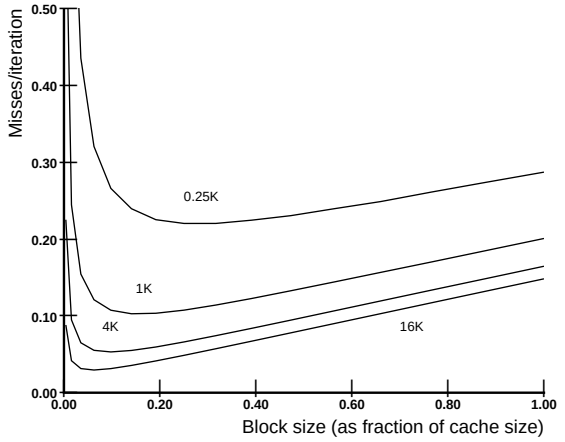


Figure 14: Data corresponding to Figures 5, 7 and 11 for a four-way set-associative cache with single-word line size.

(a) Actual versus predicted cache misses across different cache sizes (in words), $N = 295$.



(b) Miss rate (averaged over all matrix sizes) versus block size, across different cache sizes (in words).



(c) Miss rates using best fixed block size and best data-dependent square block size.

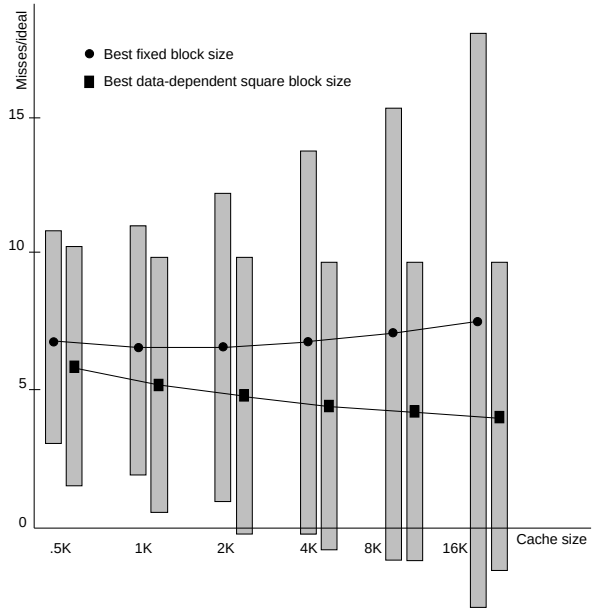


Figure 15: Data corresponding to Figures 5, 7 and 11 for a direct-mapped cache with four-word line size.

terms need to be adjusted to account for the fact that every access brings in an entire cache line. In Figure 15 we show the results of applying our model to a cache with a four-word line size.

In the ideal case, a line size of l words would reduce the miss rate by a factor of l. However, such a reduction is not observed when we compare Figure 15(b) with Figure 7; the miss rate is only reduced by a factor of just over two for the smallest caches and a factor of three for the largest one. The reason for the less than ideal behavior is as follows: when a row of a blocked matrix is brought into the cache, the beginning and end of the row do not necessarily align to cache line boundaries. This results in unaccessed data being loaded into the cache, effectively increasing the row length. This excess data can knock out useful data, increasing the miss rate. In particular, self interference starts at a somewhat smaller blocking factor. The relative cost of cache misalignment decreases as the block size is increased, which is why the decrease in the miss rate at the best block size is closer to four for larger caches than for smaller caches. This also explains why the miss rates for the data-dependent block strategy decrease with increasing cache size (Figure 15(c)): the alignment effect becomes less significant. The ratios of miss rates to ideal are still in each case greater than those in Figure 11 because the miss rate for the longer line size is not reduced by a full factor of l.

6 Copy Optimization

Neither associativity nor multiple-word line sizes can eliminate the large variance in the performance of a blocked algorithm. We now investigate a totally different approach that eliminates self interference altogether, thus guaranteeing a high cache utilization for all problem sizes. The approach is to copy non-contiguous data to be reused into a contiguous area [6]. By doing so, each word within the block is mapped to its own cache location, thus making self interference within a block impossible. This technique, when applicable, can bound the cache misses to a factor of two of the ideal. If the reuse factor is large, then the cost incurred in copying the data is negligible.

Let us first consider copying the $B \times B$ block of Y data in matrix multiplication to contiguous locations. By setting the $S_i(Y)$ term in Equation 3 to zero, the number of misses becomes

$$\frac{2N^3}{B} + \frac{4N^3B}{C}.$$

Figure 16 compares the cache miss numbers predicted by this model with the misses observed in simulation. As discussed in Section 2, the cache miss function achieves its minimum value at $B = \sqrt{\frac{C}{2}}$, and the minimum number of misses is $2\sqrt{2}$ times the ideal. Without self interference, the number of misses is not nearly as sensitive to the block size. For example, a block can be chosen to fill anywhere from one-fourth of the cache up to the whole cache without increasing the number of misses by more than 6% over the minimum.

Misses can be further reduced by copying the Z data as well, placing them at the end of the same contiguous block so that they do not interfere with the Y data. The number of misses is:

$$\frac{2N^3}{B} + \frac{2N^3B}{C}$$

which achieves its minimum at $B = \sqrt{C}$. The resulting cache miss numbers are twice ideal. The overhead of copying the destination row is substantially higher than that of copying the reused

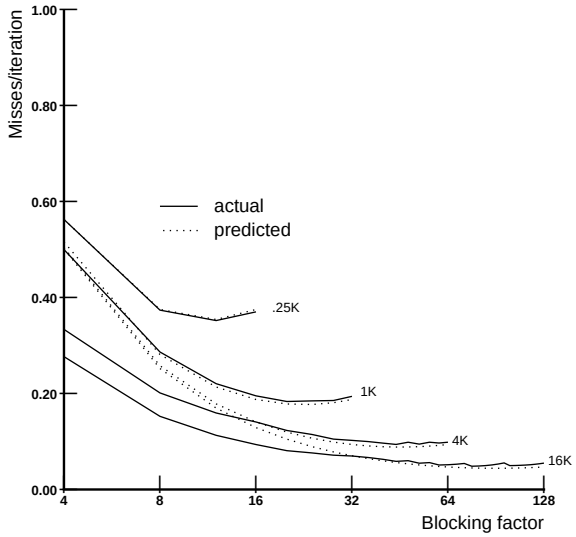


Figure 16: Actual versus predicted cache misses across different cache sizes (in words), $N = 295$.

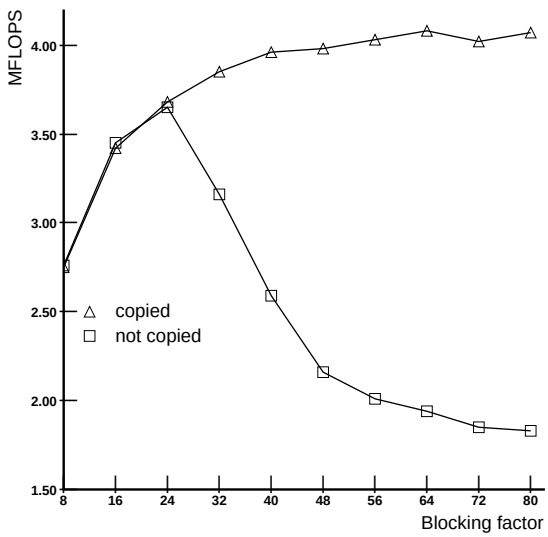


Figure 17: Performance of copying versus no copying, $N = 293$, DECstation 3100.

block. An entry from the reused block is used N times once it has been copied. An entry from the destination row, on the other hand, is reused only B times. Thus, even though the number of cache misses can be reduced by 30% by copying both the reused block and the destination row, the overhead of copying the destination row could overwhelm the benefit of fewer cache misses. In fact, our experience suggests that copying both sets of data is typically not advantageous for microprocessor systems today.

Copying can take full advantage of a cache with a longer line size. It can use at least half of the cache in each blocked loop nest, thus making the penalty due to cache line misalignment negligible. Thus, nearly all the words prefetched in the cache line will be used fully, so when copying is used with a cache line size of l, the miss rate can be reduced by a factor of l.

When the cache is set associative, we have an opportunity to reduce the minimum miss rate by eliminating not just self interference, but cross interference as well. With a set associativity a, we use $(a-1)/a$ of the cache for matrix Y and use the other $1/a$ of the cache for fetching data from X and Z. In this case, cross interference only occurs when words from rows of X and Z are mapped to the same set. Since each of the footprints of the rows of X and Z are only B/C, the probability of cross interference is small with this strategy. Thus, by choosing a blocking factor of $\sqrt{C(a-1)/a}$, we can avoid nearly all but intrinsic misses. This results in a miss rate of $\sqrt{a/(a-1)}$ times the ideal. For example, if $a = 4$, the miss rate is 1.15 of ideal, compared with 2 to 2.8 for a direct-mapped cache. Copying the Z data into a contiguous block so that Y and Z fill up $(a-1)/a$ of the cache removes the already negligible cross interference, but otherwise has little effect.

To demonstrate the effect of copying on real machines, we compare the absolute performance of a blocked matrix multiplication code with and without the copying optimization on a DECstation 3100 (Figure 17). We show the data only for the $N = 293$ case, the example which suffered the most from self interference (see Figure 2(a)). Copying allows the blocked code to deliver a consistently high level of performance for all matrix sizes.

There are several cases in which copying cannot be applied or can be applied only with difficulty. First, if the reuse factor is small, then the cost of copying can be greater than the misses saved via copying. A small reuse factor can arise because the locality within the computation is minimal, e.g. the reuse of spatial locality is a small constant. Second, the reused portion of a variable may not be exactly the same each time through a block. When the reused portion shifts, copying can be implemented with a circular buffer, with the additional addressing overhead that entails. Lastly, if a reasonably large fraction of data fits into the cache, then paying the cost of copying the data throughout may incur an unnecessary overhead.

7 Conclusions

This paper presents a comprehensive analysis of the performance of blocked code on machines with caches. While blocking has been accepted as an important optimization for scientific code, its performance on machines with caches was not well understood. By using a combination of theory and experimentation, this paper shows that blocking is effective generally for reducing the memory access latency for caches. The magnitude of the benefit, however, is highly sensitive to the problem size.

We have developed a model for understanding the cache behavior of blocked code. Through the model, we demonstrate that this cache behavior is highly dependent on the way in which a matrix interferes with itself in the cache, which in turn depends heavily on the stride of the accesses. We have derived cache miss models for four different strides: unit stride, non-unit constant stride, triangular stride, and random stride. Each of these models is validated with empirical results.

The performance of the cache is highly dependent on the problem size and the block size. The same block size can give rise to widely varying cache miss rates for very similar problem sizes. The conventional wisdom of using the entire cache, or even a fixed fraction of the cache, is incorrect. If a fixed block size is chosen, we have found that the optimal choice occupies only a small fraction of the cache, typically less than 10%. The fraction of the cache used for this optimal block size decreases as the cache

Table 2: Summary of cache miss rates, 4K word cache, as compared to ideal.

	Basic Model		Set Associativity = 4		Line Size = 4	
Ideal	$2N^3/\sqrt{C}$		$2N^3/\sqrt{C}$		$2N^3/(4\sqrt{C})$	
Method	Mean	Std. Dev.	Mean	Std. Dev.	Mean	Std. Dev.
Fixed Block Size	5.4	5.4	3.4	5.0	6.8	7.0
Best Block Size	3.4	2.4	2.0	1.1	4.4	5.2
Copy Block	2.8	0	1.2	0	2.8	0
Copy Row and Block	2.0	0	1.2	0	2.0	0

size increases. More importantly, there is a large variance in the performance obtained.

The effect of the various optimizations studied in this paper are summarized in Table 2. A multi-word line size reduces the number of cache misses but increases the memory traffic. Set associativity improves the average cache miss rate, but does not reduce the large performance variations between problem sizes. Regardless of the line size and the set associativity, it is useful to tailor the block size according to the problem size. For the non-unit constant stride case, the optimal block is the largest block which does not interfere with itself in the cache. We have presented an algorithm that determines the optimal block choice efficiently. This technique decreases the average number of cache misses as well as the variation across different problem sizes substantially. Copying is yet a better technique if it is applicable. It yields a much lower miss rate and delivers the same high performance for all matrix sizes.

Our recommendation when blocking numerical codes for a cache is to always accompany the blocking with some blocking optimization. First, use block copying if applicable. It provides the fewest cache misses and the most robust performance of the options that we have considered. If copying is not appropriate, then choose the largest block size possible that does not incur self interference within an array.

References

[1] W. Abu-Sufah, D. J. Kuck, and D. H. Lawrie. Automatic program transformations for virtual memory computers. *Proc. of the 1979 National Computer Conference*, pages 969–974, June 1979.

[2] E. Anderson and J. Dongarra. LAPACK working note 18, implementation guide for LAPACK. Technical Report CS-90-101, University of Tennessee, Apr 1990.

[3] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, June 1990.

[4] J. Dongarra, J. Du Croz, S. Hammarling, and I. Duff. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, pages 1–17, March 1990.

[5] K. Gallivan, W. Jalby, U. Meier, and A. Sameh. The impact of hierarchical memory systems on linear algebra algorithm design. Technical Report UIUCSRD 625, University of Illinios, 1987.

[6] D. Gannon and W. Jalby. The influence of memory hierarchy on algorithm organization: Programming FFTs on a vector multiprocessor. In *The Characteristics of Parallel Algorithms*. MIT Press, 1987.

[7] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformation. *Journal of Parallel and Distributed Computing*, 5:587–616, 1988.

[8] G. H. Golub and C. F. Van Loan. *Matrix Computations*. Johns Hopkins University Press, 1989.

[9] J.-W. Hong and H. T. Kung. I/O complexity: The red-blue pebble game. In *Proceedings of the Thirteenth Annual ACM Symposium on Theory of Computing*, pages 326–333. ACM SIGACT, May 1981.

[10] A. C. McKeller and E. G. Coffman. The organization of matrices and matrix operations in a paged multiprogramming environment. *CACM*, 12(3):153–165, 1969.

[11] A. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications*. PhD thesis, Rice University, May 1989.

[12] M. E. Wolf and M. S. Lam. A data locality optimizing algorithm. Submitted for publication., 1990.