Hyperblocking: A Data Reorganization Method to Eliminate Cache Conflicts in Tiled Loop Nests

USC–CS–98–671

Sungdo Moon and Rafael H. Saavedra

Computer Science Department
University of Southern California
Los Angeles, California, 90089-0781
E-mail: {sungdomo, saavedra}@cs.usc.edu
Hyperblocking: A Data Reorganization Method to Eliminate Cache Conflicts in Tiled Loop Nests

Sungdo Moon and Rafael H. Saavedra

Computer Science Department
University of Southern California
Los Angeles, California, 90089-0781
E-mail: {sungdomo, saavedra}@cs.usc.edu

Abstract

This paper presents hyperblocking, or hypertiling, a novel optimization technique that makes it possible to drastically eliminate the self and cross interference misses of tiled loop nests, while significantly reducing the overhead involved in copying data blocks. In hyperblocking, all arrays used in a tiled loop nest are reorganized at the beginning of the computation such that all data blocks are guaranteed to map into disjoint and contiguous cache regions. Doing this effectively eliminates the occurrence of interference misses in many dense-matrix computations. When data prefetching is used in combination with hyperblocking, it is possible to completely hide the latency of capacity misses, and given that no cache conflicts can occur, it makes it possible to increase the prefetch distance to cover larger latencies without causing any amount of cache pollution. Furthermore, the performance of hyperblocked loop nests no longer depends on input sizes and starting addresses, so it’s possible to predict the execution time of tiled loop nests for any input sizes. According to the simulation done in a couple of machines, hyperblocking significantly improved the performance of several tiled loop nests and outperformed previous approaches such as copy optimization and tile size selection.
1 Introduction

In today’s computer systems, as the gap between CPU speed and memory access time continues to widen, the effective utilization of the memory system has become a potential bottleneck in achieving high performance. Most current computers provide several levels of caches to reduce memory reference access times by making it possible to exploit locality. Scientific applications written without considering the effects of the memory hierarchy, however, cannot fully utilize caches due to their large data sets and naive access patterns. Thus, many compiler optimization techniques have been proposed that enable a better cache utilization such as loop permutation, loop tiling, loop unrolling, etc.

Tiling, or blocking [1, 5, 6, 12], is one of the most effective optimizations for improving the locality of numerical loop nests. It reduces the size of data sets referenced during the iterations of the innermost loops by strip-mining the iteration space. Thus, caches can hold the working sets of a loop nest and exploit more locality. Although effective in reducing capacity misses, which occur when the cache space is insufficient to hold all data to be reused, tiling increases the amount of interference misses (or conflict misses) that occur when too much data is mapped to the same cache set. Therefore, the performance of tiled algorithms is highly dependent on the layout and the size of arrays referenced in the loop [6]. Furthermore, when data prefetching is used, cache mapping conflicts increase the probability to evict prefetched data from the cache before it is used, and consequently degrade the effectiveness of prefetching. With prefetching, therefore, tiled loop nests can experience more cache misses than the original loop nests in many cases [8].

In this paper, we present a new optimization technique, called hyperblocking or hypertiling, that reorganizes data arrays with the goal of eliminating most cache conflicts in tiled loop nests. Hyperblocking makes it possible to drastically eliminate the interference misses of tiled loop nest, while significantly reducing the overhead involved in copying data blocks. In hyperblocking, all arrays used in a tiled loop nest are reorganized at the beginning of the computation in a way that all data blocks used concurrently are guaranteed to map into disjoint and contiguous cache regions. Doing this effectively eliminates the occurrence of interference misses in many dense matrix computations. Furthermore, after applying hyperblocking, the performance of a tiled loop is no longer dependent on the starting address and the size of input array. When data prefetching is used in combination with hyperblocking, it is possible to hide completely the latency of capacity misses, and given that no cache conflicts can occur, it makes it possible to increase the prefetch distance to cover larger latencies without causing any amount of cache pollution.

We have measured cache miss rates through trace-driven simulations and execution times on DEC 4000 Model 710 AXP system and Sun SPARCstation 20. Three dense matrix algorithms including matrix multiplication, LU, and SOR with hyperblocking applied were simulated and compared with three proposed heuristics for tile size selection. The results show that hyperblocking can effectively eliminate cache conflicts of tiled loop nests while incurring insignificant overheads in reorganizing the arrays. Compared with other techniques, hyperblocking outperformed in almost all cases and, more importantly, its performance was constant for a wide range of input size, something none of the other techniques can do.

The rest of paper is organized as follows. In Section 2, we review previous approaches and introduce an analytic cache model of tiled matrix multiplication. With this cache model, we show why previous tile selection heuristics are not successful in reducing cache misses of tiled loops. In Section 3, hyperblocking is introduced and applied to matrix multiplication as an example. Section 4 describes the theoretical framework of hyperblocking in detail. Experimental results are presented in Section 5 In Section 6, several issues of hyperblocking and future work are discussed. Section 7 concludes the paper.

2 Motivation

To improve the cache behavior of tiled loop nests, several techniques such as copy optimization and tile size selection have been proposed [2, 3, 6, 10]. In this section, we briefly review these approaches. Using the cache model presented in Section 2.2, we also show why tile selection heuristics are not effective.
2.1 Related Work

To cope with cache conflicts of tiled loops, *copy optimization* has been suggested [4, 6, 10]. In copy optimization, non-contiguous blocks of data to be reused are copied into a contiguous area in memory. With copy optimization, self interference within the data block is eliminated since each element is mapped into a different cache frame. However, the overhead of copying can be substantial and can sometimes outweigh the improvements gained by using copy optimization [10]. Cross interference between different arrays is not eliminated by copy optimization and continues to degrade performance. Furthermore, during copying, the original and the copied data can pollute the cache. Finally, when there exist multiple references to the same array and at least one of them is a write reference, coherence problems make it difficult to apply copy optimization. Since copy optimization has already been proven to be expensive due to the high overhead incurred by copying [2, 8], we will not consider it in this paper.

Another approach to deal with cache conflicts is using a proper tile size. Tile size plays an important role in making tiling profitable. Selecting the right tile size is critical for reducing not only capacity misses but also interference misses. For a given cache size, the optimal fixed-size tile all problem sizes is usually small, which utilizes only a small fraction of the cache, and results in large variations of miss rates over the range of problem sizes [6]. To eliminate this problem, several heuristics to tailor the tile size to the problem size have been proposed [6, 3, 2].

Lam, Rothberg and Wolf presented an algorithm (LRW) which chooses the largest square tile that does not suffer from self interference and it is based on computing the cache mapping pattern of each data block [6, 11]. LRW, however, usually selects small tile sizes, which tend to incur loop overheads that penalize the execution time. To solve these problems, Esseghir has proposed an algorithm (ESS) that chooses a tile size that maximizes the number of complete columns that fit in the cache [3]. Due to the tile’s large column size, ESS tends to minimize the loop overhead of the innermost loops. However, since the corresponding working set usually overflow the cache, capacity and cross interference misses tend to degrade performance. Coleman and McKinley have proposed another tile size selection algorithm (TSS) based on the data layout for given problem size, cache size and cache line size in a direct-mapped cache [2]. The algorithm chooses the rectangular tile that eliminates self interference misses and at the same time reduces cross interference misses while maximizing the size of working set.

All these three heuristics focus only on one aspect of tile selection: either the working set or the shape of tile. We show in the next subsection, to find the optimal tile size, both the working set size and the shape of a data block must be considered together in order to find the optimal tile size.

2.2 Cache Model of Tiled Matrix Multiplication

This section presents an analytic cache model for tiled matrix multiplication. It allows us to show why previous tile selection heuristics are ineffective in finding the optimal tile size. Since the tile selection heuristics eliminates self interference misses, our model only takes into account capacity and cross interference misses. In this model, we assume a direct-mapped $C$-word cache with $l$-word lines. We also assume a row-major array allocation.

```plaintext
for kk:=1 to N by Tk do
  for jj:=1 to N by Tj do
    for i:=1 to N do
      for k:=kk to min(N,kk+Tk-1) do
        tmp:=A[i,k];
        for j:=jj to min(N,jj+Tj-1) do
          C[i,j]+=tmp*B[k,j];
      endfor
  endfor
endfor
```

Figure 1: Tiled matrix multiplication.

The code for the tiled matrix multiplication of $N \times N$ matrices is given in Figure 1. The pure capacity misses of references to $A$, $B$ and $C$ assuming that the localized space includes at least 2 innermost loops are $N^3/(bT_j)$, $N^2/b$, and $N^2/(bT_k)$, respectively. The cross interference misses are estimated from the probability that the references’
footprints overlap. Since the reference to \(B\) mostly suffers from cross interference, the model considers only the cross interference misses between \(B\) and the other references. Hence, the approximated capacity and simple cross interference misses per iteration \((CCM)\) are given by:

\[
CCM = \frac{1}{b} \left( \frac{1}{T_j} + \frac{1}{T_k} + \frac{T_j + T_k}{C} \right) = \frac{(\varepsilon + 1)(\rho + 1)}{b\sqrt{C} \varepsilon \rho}
\]

where the eccentricity of a data block, \(\varepsilon\), and the inverse relative working set, \(\rho\), are defined as

\[
\varepsilon \equiv \frac{\max(T_j, T_k)}{\min(T_j, T_k)} \geq 1, \quad \rho \equiv \frac{C}{T_j \cdot T_k} \geq 1
\]

\(CCM\) has a minimum value of \(4/(b\sqrt{C})\) when \(T_j = T_k = \sqrt{C}\), that is, \(\varepsilon = \rho = 1\). Since our cache model approximates several effects, this ideal case \((CCM_{\text{ideal}})\) is unrealizable. It serves, however, as a lower limit for the number of cache misses.

Let’s define \(CCM^*\) as a ratio between \(CCM\) and \(CCM_{\text{ideal}}\), then,

\[
CCM^* = \frac{CCM}{CCM_{\text{ideal}}} = \frac{(\varepsilon + 1)(\rho + 1)}{4\sqrt{C} \varepsilon \rho} \geq 1
\]

\(CCM^*\) is a symmetric function of its two parameters: eccentricity and inverse working set. The significance of \(CCM^*\) is that represents a general result that applies to all cache and block size configurations. It also highlights the importance of the shape and size of tiles in determining capacity and cross interference misses. Furthermore, it is a convenient formula for understanding the effectiveness of tile size selection heuristics. These heuristics are based on eliminating self interference misses, so once these are removed, the only leftover misses are the capacity and cross interference misses, which are quantified by \(CCM^*\). Henceforth, \(CCM^*\) indicates that, to reduce the cache misses on tiled matrix multiplication, both the eccentricity and working set of tiles must be considered simultaneously. All previous heuristics have concentrated on minimizing only one of these two factors at the expense of the other and consequently cannot find the best tile size.

To illustrate the above discussion, Figure 2 shows the distributions of tile sizes selected by ESS, LRW, and TSS inside the \(CCM^*\) contour map. As the figure shows, ESS concentrates on finding tile sizes that maximize the number of rows fitting in the cache. It only tries to maximize the working set, hence it selects tiles having poor eccentricity. All tiles selected by ESS are located in the lower-right corner of the \(CCM^*\) contour map. The corresponding miss rates are, on the average, \(9.50 \pm 5.01\) times larger than the ideal case and become worse as the problem size grows relative to the cache size. In LRW, the largest square tile not suffering from self interference is the one selected. Thus, tiles obtained with LRW have good eccentricity (always equal to 1), so these are located on the y-axis of the \(CCM^*\) contour map. Although they exhibit good eccentricity, their working sets tend to be too small in most cases. Thus, they suffer from significant higher loop overheads and TLB misses, even though giving the best average for \(CCM^*\) \((1.36 \pm 0.97)\). By allowing rectangular tiles, TSS finds tiles with larger working sets than LRW, while maintaining reasonable eccentricities. However, it still chooses bad tiles in terms of eccentricity for a significant number of problem sizes. The average \(CCM^*\) value of tiles selected by TSS is \(2.79 \pm 3.25\).

From the cache model and the observations mentioned above, we can conclude that no single heuristic succeed in finding the best tile size that minimizes the execution time of a tiled loop nest for a wide range of problem sizes. For many problem sizes, each heuristic selects tiles sizes having either small working sets or bad eccentricities. Furthermore, in some pathological cases, where the problem size is a multiple of cache size, even the optimal tile cannot effectively eliminate cache conflicts. To get good performance from tiled loops for all problem sizes, the input arrays should be restructured before the computation. In the next section, we present hyperblocking which solves most of the problems affecting tile selection heuristics.

### 3 Hyperblocking

This section presents the general idea of hyperblocking and shows how it can be applied to matrix multiplication as an example. We start by introducing several terms used throughout this section. Let’s assume that a loop nest of depth
Reduced Cache Miss Ratio Values

<table>
<thead>
<tr>
<th>Reduced Cache Miss Ratio Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.02</td>
</tr>
<tr>
<td>14.16</td>
</tr>
<tr>
<td>13.25</td>
</tr>
<tr>
<td>11.20</td>
</tr>
<tr>
<td>10.02</td>
</tr>
<tr>
<td>8.69</td>
</tr>
<tr>
<td>7.11</td>
</tr>
<tr>
<td>5.05</td>
</tr>
<tr>
<td>4.53</td>
</tr>
<tr>
<td>3.94</td>
</tr>
<tr>
<td>3.61</td>
</tr>
<tr>
<td>3.24</td>
</tr>
<tr>
<td>2.83</td>
</tr>
<tr>
<td>2.35</td>
</tr>
<tr>
<td>1.74</td>
</tr>
<tr>
<td>1.59</td>
</tr>
<tr>
<td>1.43</td>
</tr>
<tr>
<td>1.25</td>
</tr>
<tr>
<td>1.15</td>
</tr>
<tr>
<td>1.06</td>
</tr>
<tr>
<td>1.00</td>
</tr>
</tbody>
</table>

Inverse Working Set (1/WS)

- ESS
- LRW
- TSS

Figure 2: Contour map for the reduced cache miss ratio (CCM*). Each contour line in the map represents a set of points that have the same value for CCM* which is given at the left-hand side. The origin corresponds to the ideal case. The problem size varies from 128 to 512.

$n$ is tilable and each loop bound is a linear function of the outer loop indices. Tiling maps an $n$-deep loop nest into a $2n$-deep loop nest consisting of $n$ controlling loops and $n$ tiled loops. After tiling, the new iteration space with $2n$ dimension can be viewed as an $n$-dimensional hyperspace of $n$-dimensional localized iteration spaces or iteration blocks. We define a data block for each array as a minimal block that contains a set of data elements that are accessed during the execution of one iteration block.

### 3.1 Hyperblocking Algorithm

The main objective of hyperblocking is to eliminate the interference misses within tiled loop nests. In hyperblocking, the self interference misses are eliminated by copying the elements of data blocks into contiguous areas in memory like copy optimization does but requiring little overhead. Furthermore, to eliminate the cross interference misses, hyperblocking combines the copied blocks so that the cache footprints of different arrays do not overlap with each other and blocks belonging to the same array are mapped into the same cache region. Hyperblocked loop nests usually consist of three parts: a prologue, a main body, and an epilogue.

The prologue consists of the code that constructs a hyperblocked array at run-time from the data blocks of the arrays used in the loop nest. A hyperblocked array is an array of rectangular-shaped data blocks. When data blocks are $n$-dimensional, the dimensionality of the hyperblocked array is usually $2n + 1$. The $n$ higher dimensions select a specific data block and the lower $n$ dimensions specify elements within the block. The remaining dimension distinguishes between the different arrays. Note that each data block of a hyperblocked array is a complete $n$-dimensional array, while the original data block is a fraction of an $n$-dimensional array. In other words, all the elements of the former are allocated in a contiguous memory region, while the elements of the latter are not. During copying, only the elements of data blocks that are accessed in the loop nest are copied into a corresponding data block of hyperblocked array. When the access strides are greater than 1, the original data block contains unused elements, which are not copied.
The tile sizes of hyperblocked loop are determined from the several parameters such as cache size, cache line size, the number of arrays used. To guarantee that each array’s data blocks are mapped into the same cache region, the volume of a lower \((n + 1)\)-dimensional subarray should perfectly match the cache. This ensures that data blocks from different arrays do not overlap with each other in the cache. In some cases, dummy arrays have to be inserted to satisfy this requirement. Our algorithm selects tile sizes that make the shape of data block be as close as possible to square, which are close to the ideal case in the \(CCM^*\) contour map given in Figure 2.

After constructing the hyperblocked array, the original computation is done in the main body. The main body of the hyperblocked loop is almost identical to the original tiled loop except that it makes references to the hyperblocked array instead of the original arrays. Also, loop bounds may require some adjustments for efficient subscripts of references to the hyperblocked array. In the epilogue, the elements of the hyperblocked array whose values are modified during the execution of the main body are copied back into the original arrays.

The prologue and the epilogue are the only extra run-time overhead induced by hyperblocking. Although this overhead is not significant, it can be avoided in some cases. For example, when all accesses to the modified arrays after the hyperblocked loop can be replaced by reference to the hyperblocked array, then the epilogue can be eliminated. Furthermore, in some cases, it is possible to change the layout of the arrays at compile-time and eliminate completely the run-time copying overhead.

### 3.2 Hyperblocked Matrix Multiplication

In this section, we show how hyperblocking can be applied to matrix multiplication of \(32 \times 32\) matrices.\(^1\) We assume that the cache is a 8KB, direct-mapped cache with 32 bytes lines and each array element is 8 bytes long. Thus, the cache can hold 1024 data elements and each cache line can contain four data elements.

Let’s first calculate the tile sizes. The number of accessed elements in the data block of \(A, B,\) and \(C\) assuming the loop \(i\) is also tiled, are \(T_i \times T_k, T_k \times T_j\) and \(T_i \times T_j\), respectively. Therefore, tile sizes are selected from the equation: \(T_i \times T_j + T_k \times T_j + T_i \times T_j + t_1 \times t_2 = 1024\), where \(t_1 \times t_2\) represents the size of dummy array and all tile sizes are multiple of cache line size in elements. \(T_i = T_j = T_k = t_1 = t_2 = 16\) satisfies the above equation. That is, each data block contains 256 elements and the dummy data block of 256 elements is required. Therefore, the hyperblocked array is a \(2 \times 2 \times 4\) array of \(16 \times 16\) 2-dimensional data blocks. The mapping of the hyperblocked array into a direct-mapped cache is illustrated in Figure 3. It shows how the data blocks of different arrays are mapped into disjoint contiguous cache region. However, due to the dummy array added, one fourth of the cache area cannot be utilized.

The simplified code for hyperblocked matrix multiplication is given in Figure 4. The first three loops are part of the prologue that construct the hyperblocked array and the last loop nest is the epilogue that restores the array \(C\).

---

\(^1\)Hyperblocking does not require the compile-time knowledge of problem size. We chose 32 just for simplicity.
for ii:=1 to ceil(N/Ti) do
  for kk:=1 to ceil(N/Tk) do
    CopyTile(H[ii,kk,1],A[ii*Ti,kk*Tk]);

for kk:=1 to ceil(N/Tk) do
  for jj:=1 to ceil(N/Tj) do
    CopyTile(H[kk,jj,2],B[kk*Tk,jj*Tj]);

for ii:=1 to ceil(N/Ti) do
  for jj:=1 to ceil(N/Tj) do
    CopyTile(H[ii,jj,3],C[ii*Ti,jj*Tj]);

for kk:=1 to ceil(N/Tk) do
  for jj:=1 to ceil(N/Tj) do
    for ii:=1 to ceil(N/Ti) do
      MatrixMultiplication(H[ii,jj,3],H[ii,jj,1],H[kk,jj,2]);

for ii:=1 to ceil(N/Ti) do
  for jj:=1 to ceil(N/Tj) do
    CopyTile2(C[ii*Ti,jj*Tj],H[ii,jj,3]);

**Figure 4:** The simplified code for hyperblocked matrix multiplication. *CopyTile* and *CopyTile2* copies elements between the original data blocks and the data blocks on the hyperblocked array.

### 4 Compiler Framework

In this section, the compiler framework of hyperblocking are presented. Basically, the procedure consists of four steps, which are described in the following subsections. We first start with several basic concepts that are necessary to understand the rest of this section.

#### 4.1 Basic Concepts

Let’s assume that a loop nest of depth $n$ is tilable and each loop bound is a linear function of outer loop indices. Each iteration of the loop nest is represented as a point in the $n$-dimensional iteration space, $p = (p_1, p_2, \ldots, p_n)$, where $p_i$ is the value of $i$-th loop index counting from the outermost to the innermost loop. All iterations of the loop nest constitute a polytope in the $n$-dimensional iteration space. Let $A$ be an $m$-dimensional array that is referenced inside of the loop nest and all subscripts of array references be affine functions of loop indices. An array element, $A[d_1, d_2, \ldots, d_m]$, is represented as a point, $\tilde{d} = (d_1, d_2, \ldots, d_m)$, and the entire array as a polytope in the $m$-dimensional data space. Indexing functions of an array reference represent a mapping from the iteration space to the data space. The mapping is a linear transformation, $Z^n \rightarrow Z^m : \tilde{f}(\tilde{p}) = H \cdot \tilde{p} + \tilde{c}$, where $H$ is an $m \times n$ *index matrix* and $\tilde{c}$ is an $m$-element constant offset vector.

Tiling maps an $n$-deep loop nest into a $2n$-deep loop nest which is consisted of $n$ *controlling loops* and $n$ *tiled loops*. We can view the new iteration space with $2n$ dimension after tiling as an $n$-dimensional hyperspace of $n$-dimensional *localized iteration spaces* or *iteration blocks*. During iterations of one iteration block, a set of data elements, which we call *data block*, is accessed and all data blocks have the same shape except of those corresponding to iteration bounds. Then, the mapping function from an iteration block to a data block is identical to the original one.

The shape of a data block is easily characterized from the index matrix. Let’s define a *shape vector* of an array reference, $\tilde{S}$, comprised of a set of *access vectors* which are the direction vectors of data block’s edges. Formally, a shape vector is defined as:
\[ S_1 = (a_1^*, a_2^*, \ldots, a_n^*) = \bar{E}H, \quad \text{where} \quad a_i^* = \sum_{k=1}^{m} h_{ki}e_k \]

Here, \( \bar{E} \) is an \( m \)-element vector whose element is a unit direction vector of each coordinate of the data space. \( h_{ki} \) represents an element at the \( k \)-th row and the \( i \)-th column in the index matrix and \( e_i \) represents a unit direction vector of the \( k \)-th coordinate. The \( i \)-th access vector is simply defined as a dot product between the \( i \)-th column of the index matrix and the vector \( \bar{E} \), and represents the direction of array accesses as the \( i \)-th loop index changes. It can be \( 0 \) if \( i \)-th loop index is not used as a subscript of array reference, which means that the \( i \)-th loop carries a temporal locality. Although a shape vector can be defined only with non-zero access vectors, we include all for a simplicity of the notation.

In the most cases, the number of linearly independent access vectors in a shape vector that is equivalent to the rank of an index matrix, is equal to the dimensionality of an array. In other words, the rank of an index matrix represents the dimensionality of data block and cannot be greater than \( m \). When it’s less than \( m \), only subdimension of the original array is accessed. In this case, we can treat it as a dimensionality of array since hyperblocking only uses the elements accessed within a loop nest. When the number of access vectors is larger than the rank of index matrix, the data block moves in the direction of a linearly dependent access vector as the corresponding index iterates. In this case, we will consider only \( m \) innermost linearly independent access vectors. Hence, we will assume that the number of access vectors and the rank of index matrix is equal to the dimensionality of an array.

### 4.2 Finding Data Transformation Matrices

The first step in hyperblocking is to find a necessary data transformation which improves data locality and changes the layout of data blocks to be a rectangular shape. Data locality is improved via making a set of data located along the direction of the last access vector be laid out in the direction of the last dimension of data block, which will be allocated into a contiguous area in memory. The other dimensions are also adjusted according to the order of access vectors to decrease distance between consecutively accessed data. When the access stride is greater than \( 1 \), the array is packed only with the data elements that are accessed for an effective utilization of cache lines.

Let \( \bar{S}' \) and \( H' \) be the shape vector and the index matrix of a transformed data block respectively. Then, the \( m \times m \) data transformation matrix \( D \) is calculated from the following equations:

\[ \bar{S}' = (a_1'^*, a_2'^*, \ldots, a_n'^*) = \bar{E}H', \quad \text{where} \quad a_i'^* = \begin{cases} 0 & \text{if } a_i^* = 0 \\ e_j & \text{otherwise} \end{cases} \]

\[ H' = DH, \quad \text{where} \quad h'_{ij} = \begin{cases} 1 & \text{if } a_i^* = e_j^* \\ 0 & \text{otherwise} \end{cases} \]

When \( a_i'^* = e_k \) and \( a_j'^* = e_l \), \( i = j \) implies \( k = l \) and \( i > j \) implies \( k > l \). In fact, since it’s possible to directly derive a new index matrix from the original index matrix, solving the first equation is not necessary.

Let’s consider the loop nest given in Figure 5(a) as an example. The reference \( A \) have temporal reuses on loop \( j \) and spatial reuses on loop \( i \), but the spatial locality is hardly exploited due to its large reuse distance even after tiling. Although simply interchanging loop \( i \) and \( k \) will improve the spatial locality, let’s concentrate on data transformations.

The index matrix of the reference \( A \) is

\[ \begin{bmatrix} 0 & 0 & 2 \\ 1 & 0 & 1 \end{bmatrix} \]

and its shape vector is \( (e_1^*, 0, 2e_1^* + e_2^*) \). The shape vector and the index matrix of a destined data block is \( (e_1^*, 0, e_2^*) \) and

\[ \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} \].

Hence, the data transformation matrix becomes

\[ \begin{bmatrix} -1/2 & 1 \\ 1/2 & 0 \end{bmatrix} \]

as illustrated in Figure 5(c). After the transformation, the data block only contains data elements that is used within the loop nest and they will be allocated in the order they are accessed in memory.
(a) Example code
\begin{verbatim}
for i:=1 to N do
  for j:=1 to N do
    for k:=1 to N do
      f(A[2k,i+k]);
\end{verbatim}

(b) Tiled code
\begin{verbatim}
for ii:=1 to N by Ti do
  for jj:=1 to N by Tj do
    for kk:=1 to N by Tk do
      for i:=ii to min(N,ii+Ti-1) do
        for j:=jj to min(N,jj+Tj-1) do
          for k:=kk to min(N,kk+Tk-1) do
            f(A[2k,i+k]);
\end{verbatim}

(c) Data transformation
\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{data_transformation}
\caption{Data transformation diagram.}
\end{figure}

(d) Hyperblocked code
\begin{verbatim}
for ii:=1 to ceil(N/Tk) do
  for kk:=1 to ceil(N/Tk) do
    for i:=1 to min(Ti,N-ii*Ti+1) do
      for k:=1 to min(Tk,N-kk*Tk+1) do
        H[ii,kk,i,k]:=A[2*(k*Tk+k),i*Ti+i+k*Tk+k];

for ii:=1 to ceil(N/Ti) do
  for jj:=1 to ceil(N/Tj) do
    for kk:=1 to ceil(N/Tk) do
      for i:=1 to min(Ti,N-ii*Ti+1) do
        for j:=1 to min(Tj,N-jj*Tj+1) do
          for k:=1 to min(Tk,N-kk*Tk+1) do
            f(H[ii,kk,i,k]);

for ii:=1 to ceil(N/Tk) do
  for kk:=1 to ceil(N/Tk) do
    for i:=1 to min(Ti,N-ii*Ti+1) do
      for k:=1 to min(Tk,N-kk*Tk+1) do
        A[2*(k*Tk+k),i*Ti+i+k*Tk+k] := H[ii,kk,i,k];
\end{verbatim}

Figure 5: The procedure for hyperblocking of the given example code.
4.3 Combining Arrays

After individually transforming each array according to its access pattern, a hyperblocked array is constructed by combining data blocks of one or more arrays. The arrays that have the same dimensionality and the same data block size, which is the number of elements within a transformed data block, are able to be merged into the same hyperblocked array. The data block size $E$ is simply a product of tile sizes whose corresponding access vector is not zero. That is, $E = \prod_{i=1}^{n} a_i^{T} = H'T$, where $T$ is a column matrix of $n$ tile sizes. Therefore, the tile sizes should be carefully selected to make the data blocks from the different arrays be combined.

In hyperblocking, the tile sizes are determined by considering the cache size, the data element size, and the number of arrays. Let $C$ be the cache size in elements and $l$ be the number of arrays being merged. To ensure that each array is mapped to its own region in the cache, it’s important to select tile sizes that make the sum of data block sizes be exactly matched with the cache size, i.e., $C = (l + x)E$, where $x$ is the number of dummy arrays required to satisfy the restriction. If we choose tile size that is a power of 2, then we can always find $x$ that satisfies the equation. Currently, we chooses tile sizes that make the shape of data block square or close to square, which is close to the optimal in terms of cache misses. To optimize the execution time, tile size have to be selected also considering the ratio of loop overheads and cache penalties, but in the machines we used for simulation, the improvement was minimal.

Hyperblocking combines $m$-dimensional arrays into a $(2m + 1)$-dimensional hyperblocked array\(^2\). $(l + x)$ $m$-dimensional data blocks, one from each array, constitute $(m + 1)$-dimensional array and $(m + 1)$-th dimension is used to make a distinction between the arrays. If we consider a data block as one data element, the original array can be viewed as an $m$-dimensional array of data blocks. Thus, the $m$-dimensional array of such $(m + 1)$-dimensional merged subarrays can be constructed and becomes a $(2m + 1)$-dimensional array. Then, the size of a hyperblocked array will be:

$$b_i' = \begin{cases} 
(\sum_{i=1}^{m} L_i)_{i}/(H'T)_{i} & (1 \leq i \leq m) \\
(m + l) & (i = m + 1) \\
(H'T)_{i-m-1} & (m + 2 \leq i \leq 2m + 1)
\end{cases}$$

where, $b_i'$ is a bound of the $i$-th dimension and $L$ is a column vector consisted of the total iteration number of each index. Note that $x/(l + x)$ of the total space is wasted for dummy arrays and the same amount of cache area is never utilized.

If all arrays cannot be merged into one hyperblocked array, we may construct more than one hyperblocked arrays, one for each group of arrays that have the same characteristics. In this case, we have to make sure that they don’t share the same cache area and more wasted space is required. However, we were able to combine arrays that differ in the dimensionality or the data block size through the alias facilities provided in C language and to reduce the amount of wasted memory and cache space. Therefore, if a programming language provides the ability that make it possible to access array with a different view, the restrictions on the dimensionality and the data block size are removed in most cases. Even though a high-level programming language doesn’t have such facilities, the compiler could be able to generate the same machine codes.

Multiple references with different indexing functions to the same array makes it difficult to construct a hyperblocked array. If they are uniformly generated, i.e., share the same index matrix, the data blocks of a leading reference can be used to construct a hyperblocked array, although some amount of cache interferences cannot removed. When all of them is read references, then we may maintain several copies of the same array, one for each group of uniformly generated references. When at least one of them is a write reference, we have to select one of them and only one copy is made.

4.4 Calculating New Subscripts

After the hyperblocked array is established, all the array references to the original arrays must be changed into the references to the hyperblocked array. New subscripts can be calculated from the old subscripts using modulo and integer division operations as follows:

\(^2\)when only one array is used to construct the hyperblocked array, it becomes $2m$-dimensional array.
where, $S$ is a column matrix of the original subscripts. Although using modulo and division operations make it easy to produce a set of new subscripts, it may introduce too much overhead to compute the address of array element. However, after adjusting loop bounds as explained in the next subsection, the following set of equations can be used to obtain a new set of subscripts.

$$s'_i = \begin{cases} 
    (DS)_i/(H'T)_i & (1 \leq i \leq m) \\
    id & (i = m + 1) \\
    (DS)_{i-m-1} \mod (H'T)_{i-m-1} & (m + 2 \leq i \leq 2m + 1)
\end{cases}$$

Here, both $II$ and $I$ are column matrices and respectively comprised of controlling loop indices and tiled loop indices. For example, The reference $A[2k,i+k]$ of the example loop nest is changed to $H[i/Ti,k/Tk,i\%Ti,k\%Tk]$ or $H[ii,kk,i,k]$.

### 4.5 Code Generation

A hyperblocked loop nest consists of three parts; a prologue, a main body, and an epilogue. In epilogue part, a hyperblocked array is constructed as explained in Section 4.3. The actual computation is done in the main body whose structure is similar to a tiled loop nest. The only differences are array references which were explained in Section 4.4 and loop bounds. In hyperblocked code, all loops start from 1 and their upper bounds are adjusted as follows:

$$u'_i = \begin{cases} 
    [(u_i - l_i + 1)/t_i] & (1 \leq i \leq n) \\
    t_{i-n} & (m + 1 \leq i \leq 2n)
\end{cases}$$

where $l_i$ and $u_i$ are a lower and a upper bound of the original loop. If a loop bound is a function of the upper loop indices, the changes in them should be applied before. In the epilogue part, arrays that are modified during the execution of the main body are copied back to the original arrays. The prologue and the epilogue part can be omitted in some cases. After applying hyperblocking to the example loop nest, as shown in Figure 5(d), the subscripts of hyperblocked array reference become simpler than those of the original ones.

When multiple references to the same array exists and data blocks of each reference don’t exactly match, code generation become more complex in order to generate efficient subscripts. Sometimes, we have to use expensive modulo and division operations. However, in most of the scientific codes, these references are usually uniformly generated, which make it possible to generate efficient code. Let’s consider the following loop nest:

```plaintext
for i:=1 to N do
    for j:=1 to N do
        f(A[i,j],A[i,j+1]);
```

Both references are uniformly generated and the last column of second reference’s data block overlap with the first column of the first reference’s data block. Assuming that the hyperblocked array is constructed using the data blocks of the first reference, peeling the last iteration of the innermost loop will produce the following efficient code:

```plaintext
for ii:=1 to ceil(N/Ti) do
    for jj:=1 to ceil(N/Tj) do
        for i:=1 to min(Ti,N-ii*Ti+1) do
            for j:=1 to min(Tj-1,N-jj*Tj) do
                f(H[ii,i,jj,j],H[ii,i,jj,j+1]);
                f(H[ii,i,jj,j],H[ii,i,jj+1,1]);
```

Therefore, even though multiple references to the same array are exist, if they are uniformly generated, an efficient hyperblocked code can be generated by means of loop peeling.
5 Experimental Results

In this section, we present simulation results on the effectiveness of hyperblocking. Three dense matrix algorithms including matrix multiplication(MM), LU, and SOR are used for our experiments. Each of these applications has a distinct access pattern and represents a large class of dense matrix algorithms. We have measured cache miss rates using a trace-driven simulator and measured execution times on the DEC Alpha 4000/710 and SPARCStation 20 varying the problem size from 32 to 1024. We compared the performance of hyperblocked version with that of tiled version using tile sizes selected by ESS, LRW, and TSS.

For the three algorithms used in our experiments, hyperblocking was very effective in eliminating interference misses and outperformed all three heuristics in most cases. It also provided very stable performance for all problem sizes, while the performance of tiled algorithms varied dramatically as the problem size changed. Furthermore, hyper-blocking was very effective in the presence of prefetching and able to eliminate most of cache misses. In the next two subsections, we present the simulation results in detail.

5.1 Cache Miss Rates

We measured cache miss rates of the four versions of algorithms using a trace-driven simulator with and without data prefetching. Prefetch instructions was inserted 4 loop iteration ahead using the algorithm introduced by Mowry, Lam and Gupta [7]. For all algorithms we used, The localized iteration space is larger than the 2 innermost loops, but smaller than the volume induced by 3 innermost loop. Thus, we used two prefetch schemes: one assuming that the localized iteration space includes 3 innermost loops (PF1) and the other one assuming that it includes 2 innermost loops (PF2).

Figures 6(a) and 6(b) show the miss rates of the four versions without data prefetching (NPF) and with data prefetching (PF2), respectively. A few points that are not within the range of y-axis are not shown in the figures. Our simulation results clearly showed that hyperblocking effectively eliminates cache misses and outperforms the other techniques. When data prefetching is used, the benefits of hyperblocking were more substantial. Amongst three tile size selection heuristics, no single heuristic outperformed the others for all problem sizes. On the average, ESS showed the worst cache miss rates and they increased in proportion to the problem size. Both LRW and TSS showed large variations in cache miss rates. Although LRW gave the lowest average cache miss ratio, it can be much worse in some pathological cases. For example, it can be as high as 99.59% when the problem size is 1024 in MM.

<table>
<thead>
<tr>
<th></th>
<th>ESS</th>
<th>LRW</th>
<th>TSS</th>
<th>HT</th>
</tr>
</thead>
<tbody>
<tr>
<td>loads per iteration</td>
<td>2.003 ± 0.004</td>
<td>2.000 ± 0.070</td>
<td>2.041 ± 0.051</td>
<td>2.091 ± 0.039</td>
</tr>
<tr>
<td>NPF miss rate</td>
<td>17.18 ± 6.34</td>
<td>4.39 ± 5.13</td>
<td>6.91 ± 4.62</td>
<td>1.93 ± 0.30</td>
</tr>
<tr>
<td>PF1 miss rate</td>
<td>8.85 ± 2.67</td>
<td>2.17 ± 1.49</td>
<td>4.18 ± 1.97</td>
<td>0.06 ± 0.01</td>
</tr>
<tr>
<td>PF ratio</td>
<td>8.48 ± 4.47</td>
<td>2.40 ± 9.10</td>
<td>2.77 ± 3.44</td>
<td>1.94 ± 0.51</td>
</tr>
<tr>
<td>redundant PF</td>
<td>0.62 ± 0.84</td>
<td>0.39 ± 0.91</td>
<td>0.25 ± 0.87</td>
<td>0.26 ± 0.32</td>
</tr>
<tr>
<td>canceled PF</td>
<td>0.71 ± 0.29</td>
<td>0.36 ± 0.14</td>
<td>0.44 ± 0.32</td>
<td>0.078 ± 0.06</td>
</tr>
<tr>
<td>PF2 miss rate</td>
<td>1.50 ± 1.13</td>
<td>0.75 ± 0.27</td>
<td>1.05 ± 0.55</td>
<td>n/a</td>
</tr>
<tr>
<td>PF ratio</td>
<td>21.00 ± 4.45</td>
<td>16.13 ± 5.23</td>
<td>16.13 ± 3.49</td>
<td>n/a</td>
</tr>
<tr>
<td>redundant PF</td>
<td>28.37 ± 21.8</td>
<td>78.89 ± 8.41</td>
<td>65.80 ± 17.3</td>
<td>n/a</td>
</tr>
<tr>
<td>canceled PF</td>
<td>0.97 ± 0.96</td>
<td>2.67 ± 1.25</td>
<td>1.95 ± 1.23</td>
<td>n/a</td>
</tr>
</tbody>
</table>

Table 1: Average statistics (in % except for loads per iteration) of tiled and hyperblocked matrix multiplication in 8K, 32B-line direct-mapped cache when the problem size ranges from 32 to 1024. PF ratio is the ratio of the total number of prefetches to the total number of loads. Redundant PF represents a percentage of prefetches, where the data is already in the cache or another prefetch to the same line is in-flight.Canceled PF is a percentage of prefetches that are canceled or evicted from the cache before uses due to the conflicts with other prefetches and loads.

In Table 1, simulation results of MM are summarized. It gives the average values of several interesting metrics with their standard deviations. In MM, hyperblocking is able to completely eliminate interference misses, so the leftover cache misses are the misses that occur within the prologue and the epilogue plus the capacity misses within the main body which can only be eliminated by using prefetching. Hyperblocking decrease cache miss rates by 56.04% over
Figure 6: Cache miss rates of hyperblocked and tiled matrix multiplication in 8K, 32B-line, direct-mapped cache (a) without data prefetching and (b) with data prefetching (prefetch distance = 4).
LRW which shows the best miss rates among heuristics.

When data prefetching is used, it eliminates all capacity misses within the main body and the only leftover misses are the interference misses within the prologue and the epilogue which are negligible. Prefetching improves the cache miss rates of LRW by 82.92% (50.10% in PF1), while it improved hyperblocking by 99.68% and made hyperblocking outperform LRW by 99.20% (99.71% vs. PF1). These results clearly show the effectiveness of data prefetching when cache conflicts are removed. For tiled MM, larger number of prefetches in PF2 improves miss rates of all three heuristics. However, additional prefetches don’t help much because most of them are redundant or canceled. Since, in hyperblocking, each array is mapped into a distinct cache region, the localized iteration space can be independently defined for each array, which is usually larger than that of tiled loops. This reduces the number of prefetches to cover capacity miss latency.

5.2 Execution Times

For our experiments, cache miss rates do not always represent the performance of algorithm, because each version incurs different amount of loop overhead. Therefore, we implemented four versions of each algorithm and measured execution times on real machine: DEC 4000 Model 710 AXP system having an 8KB, 32B-line, direct-mapped first-level data cache and a 4MB second-level cache and Sun SPARCstation 20 having a 16KB, 32B-line, 4-way associative first-level data cache and an 1MB second-level cache.

Throughout the experiments, we made several observations. First, on the average, hyperblocking outperformed the other techniques in most cases and showed stable performance over all problem sizes except on very small sizes. When problem size is small, the copying overheads are relatively large and these offset the benefits. As problem size increases, however, copying costs become insignificant and the benefits of hyperblocking become larger. Second, amongst the tile selection heuristics, LRW gave the best average performance except SOR in DEC 4000. The variation of performance over many problem sizes, however, was the worst due to small tile sizes selected for many problem sizes. Small tile sizes incurred high capacity misses, TLB misses and loop overhead. Finally, The performance improvement obtained by using hyperblocking in SPARCstation was different from that obtained in the DEC 4000 system. The main reason is due to the DEC compiler’s partial inability to generate efficient code. The differences in the cache and architectural parameters also affect the relative performance.

Figure 7, 9 and 10 illustrate the simulation results of MM, LU, and SOR over a range of problem sizes varied from 32 to 1024. In the figures, there exist a few points that are outside of the range of y-axis. Table 2 summarizes the average execution times per iteration in the range of problem size from 512 to 1024.

<table>
<thead>
<tr>
<th></th>
<th>ESS</th>
<th>LRW</th>
<th>TSS</th>
<th>HT</th>
</tr>
</thead>
<tbody>
<tr>
<td>MM</td>
<td>DEC</td>
<td>146.26 ± 25.36</td>
<td>108.31 ± 72.35</td>
<td>110.12 ± 26.31</td>
</tr>
<tr>
<td></td>
<td>SPARC</td>
<td>234.77 ± 16.69</td>
<td>172.77 ± 44.32</td>
<td>186.51 ± 26.08</td>
</tr>
<tr>
<td>LU</td>
<td>DEC</td>
<td>107.28 ± 9.46</td>
<td>106.65 ± 47.79</td>
<td>109.64 ± 26.65</td>
</tr>
<tr>
<td></td>
<td>SPARC</td>
<td>194.08 ± 10.66</td>
<td>179.18 ± 33.59</td>
<td>206.60 ± 63.94</td>
</tr>
<tr>
<td>SOR</td>
<td>DEC</td>
<td>149.35 ± 7.42</td>
<td>158.73 ± 54.66</td>
<td>149.51 ± 24.30</td>
</tr>
<tr>
<td></td>
<td>SPARC</td>
<td>368.82 ± 13.41</td>
<td>347.08 ± 62.69</td>
<td>359.07 ± 29.15</td>
</tr>
</tbody>
</table>

Table 2: The average execution times per iteration (in nanosec.) of four versions of MM, LU, and SOR on DEC 4000 model 710 AXP system and Sun SPARCstation 20. The execution times of hyperblocked code are broken into two parts: the execution time of main body and the overhead (the sum of the execution times of a prologue and an epilogue).

In matrix multiplication, \( T_i = T_k = T_j = 16 \) on the DEC 4000 and \( T_i = T_j = 32, T_k = 16 \) on the SPARCstation. Hyperblocking improves the average execution time of MM over LRW by 13.88% (12.16% including overhead) in DEC 4000 and 14.44% (12.33%) in SPARCstation, and its performance variation is less than 0.57% of the total execution time. The copying overhead is quite small and contributes only 1.96% of the total execution time in the DEC 4000 and 2.40% in the SPARCstation. Although small degree of cache associativity doesn’t help much in reducing cache conflicts, tiled loop can get benefits from 4-way associativity of SPARCstation’s data cache, while the performance of hyperblocked loop isn’t affected by the degree of associativity as no conflict occurs. On the other hand, in the DEC 4000, a quarter of cache area is not utilized by hyperblocked matrix multiplication while it uses the entire cache in the
Figure 7: Execution times of matrix multiplication on DEC 4000 Model 710 AXP system and Sun SPARCstation 20.
for jj:=1 to N by Tj do
for kk:=1 to N by Tk do
for i:=1 to N do
  for j:=max(i+1,jj) to min(N, jj+Tj-1) do
    if kk<=i+1<=kk+Tj-1 then
      A[j,i] /= A[i,i];
  for k:=max(i+1,kk) to min(N, kk+Tk-1) do

(a) Tiled LU

for jj:=2 to N-1+T by Tj do
for kk:=2 to N-1+T by Tk do
  for i:=max(1, jj-N+1, kk-N+1) to min(T, jj+Tj-2, kk+Tk-2) do
    for j:=max(1, jj-i) to min(N-1, jj-i+Tj-1) do
      for k:=max(1, kk-i) to min(N-1, kk-i+Tk-1) do

(b) Tiled SOR

Figure 8: The codes for tiled LU and SOR. SOR is skewed twice before applying tiling to make it tilable and after tiling to make subscripts simpler.

SPARCstation.

The tiled code for LU decomposition without pivoting is given in Figure 8(a). In LU, there are four references to the same array with different access patterns, so the same tile size have to be selected for all loops in order to make the shape of each data block identical. Because each reference may access different data block during the iterations of one iteration block, four different data blocks may share the same cache area. Therefore, cache conflicts can not be completely eliminated and they may introduce more interference. Although hyperblocking is less effective in reducing cache conflicts of LU, on the average, it outperforms LRW by 13.37% (11.54%) on the DEC 4000 and by 6.96% (4.33%) on the SPARCstation. The less performance gain in the SPARCstation is mainly because hyperblocked LU uses only a half of cache area, while tiled LU fully utilizes the entire cache.

The original SOR code requires loop skewing before applying tiling due to data dependences (Figure 8(b)). Loop skewing usually makes it difficult to apply hyperblocking because, after skewing, the data blocks move as controlling loops iterate and overlap with each other. All references of SOR access the same array with the same shape, so tile sizes that fully utilize the cache can be selected without dummy array. The average performance of hyperblocked SOR is 6.70% (10.45%) worse than that of ESS on the DEC 4000 and 5.32% (2.25%) better than that of LRW on the SPARCstation. Since the original tiled SOR does not suffer from much cache conflicts, the performance improvement on the SPARCstation is relatively small compared to that of other algorithms. The bad performance of hyperblocked SOR on the DEC 4000 is mainly because the compiler’s inability to produce an efficient code for the hyperblocked code complicated due to overlapping data blocks. However, if the compiler is able to generate as efficient hyperblocked code as tiled code, we expect hyperblocked SOR will provide good performance.

In overall, our experiments clearly shows the effectiveness of hyperblocking. Although hyperblocking can be outperformed by tiling for some problem sizes, the performance difference is very small. Furthermore, it provides a good stable performance over all problem sizes, which cannot be obtained using tile selection heuristics.
Figure 9: Execution times of LU on DEC 4000 Model 710 AXP system and Sun SPARCstation 20.
Figure 10: Execution times of SOR on DEC 4000 Model 710 AXP system and Sun SPARCstation 20.
6 Discussion

6.1 Benefits of Hyperblocking

Although the main objective of hyperblocking is eliminating interference cache misses in tile loop nests, as a result of reducing cache conflicts, it provides numerous additional benefits. Here, we summarize the advantages of hyperblocking over other techniques.

- Improved locality: Although several code transformation techniques effectively improve locality of loop nests, it cannot be fully exploited without combination with data transformations. In hyperblocking, any data transformation can be applied to data blocks to improve locality. Usually, elements of data block are rearranged in the order they are accessed, so more spatial locality can be achieved. Furthermore, with non-unit access stride and multiple-word cache line, it reduces cache area consumed by data block as well as improving cache line locality because it copies only data elements that are accessed.

- Minimal copying overheads: Copy optimization incurs too much overheads, since it have to copy same data block many times inside a loop nest. However, as hyperblocking constructs a new array before entering a loop nest, it can reuse data block of hyperblocked array instead of copying repeatedly, delivering same benefits of copy optimization. Each element is copied at most once and thus copying overheads are order of the dimension of an array, while the overheads of copy optimization generally increase in order of the depth of a loop nest which is usually larger than array dimension.

- No coherence problem: In the loop nest having multiple references to the same array, if at least one of them is a write reference like LU and SOR, using copy optimization is difficult due to the coherence problem. However, hyperblocking maintains only one copy of each array and all references use the same copy, so no coherence problem is happened while it providing same benefits of copy optimization with minimal overheads.

- Efficient data prefetching: The reduced cache conflicts directly improve the effectiveness of data prefetching because of the less probability of prefetched data being flushed. In general, data prefetch instructions are inserted according to the locality analysis to cover the latency of capacity misses. Since hyperblocking increases the dimensionality of localized space, a large number of prefetch instructions can be reduced. Furthermore, it enables a precise analysis of cache behavior and a better prefetching strategy. Reduction of prefetches could be an important factor to achieve good performance in the multiprocessor where the network bandwidth is limited.

- Larger prefetch distance: The prefetch distance is an important parameter for the effective data prefetching. Memory latency is not fully hidden if it is too small, and the data arrived too early may be flushed from the cache before they can be used if too large. Due to large variance of memory latency, using a fixed prefetch distance may result in poor performance. Thus, adaptive prefetching that adjusts prefetch distance at run-time using runtime statistics have been proposed [9]. Since hyperblocking is able to significantly reduces cache conflicts that may pollute prefetched data, a larger prefetch distance can be used to cover longer latencies without concerning early arrived data being flushed.

- Stable performance: The performance of a tiled loop nest usually varies dramatically depending on the problem size even if a tile size is tailored to it. The performance of hyperblocked code doesn’t depend on the problem size and can be precisely estimated. Therefore, hyperblocking may be a useful technique in building scientific libraries and real-time applications.

6.2 Issues and Future Work

The overheads involved in hyperblocking are copying costs and complicated address calculations. Although copying overheads generally contribute little to a total computation, they can be avoided in some situations. In general, scientific computation usually spends most of its execution time in a few core loop nests that usually have similar structures and operate on the same arrays. Thus, when an array is used in one hyperblocked array, the compiler can directly change the layout or the definition instead of copying at run-time. And, copying back the array modified within a main body to the original array is not necessary if all references to that array after hyperblocked loop nest can be
transformed to the references to the hyperblocked array. Doubled dimensionality of hyperblocked array may result in excessive overheads in locating data elements. However, we showed that the efficient address calculations could be done with minor changes in loop bounds and loop peeling in the existence of multiple references to the same array.

Hyperblocking alone is an effective technique to improve cache performance but it also enables a further optimization as well. Although a hyperblocked loop nest can effectively utilize caches and shows good performance for any problem sizes, its performance is not optimal for some problem sizes due to overheads involved in copying and address calculation. In other words, there may exists an optimal tile size that yields a better performance. As explained in Section 2.2, the optimal tile size can be found in a reasonable number of searches. Therefore, we could combine these two techniques for a further optimization of tilled loops.

7 Conclusion

This paper introduced hyperblocking, which is an effective compiler optimization technique to minimize cache conflicts of a tiled loop nest retaining benefits of copy optimization, the best tile size selection, and data transformations in one framework with minimal overhead. It is also a good programming technique to make the performance of tiled dense matrix computations be stable for all problem size. From the experiments, we showed that it effectively eliminated interference misses and outperformed the other techniques. Moreover, the predictable cache behavior of hyperblocked loop enables other optimizations such as data prefetching to take advantages of it.

References


