Automatic Generation of Block-Recursive Codes

Nawaaz Ahmed and Keshav Pingali

Department of Computer Science,
Cornell University, Ithaca, NY 14853

Abstract. Block-recursive codes are likely to be the centerpiece of the next generation of numerical linear algebra libraries since they perform better on machines with deep memory hierarchies than existing library codes like LAPACK. In this paper, we describe compiler technology to translate iterative versions of some numerical kernels into block-recursive form. We also study the cache behaviour and performance of these compiler-generated block-recursive codes.

1 Introduction

Locality of reference is important for achieving high-performance on modern processor architectures with multiple levels of memory hierarchy. Current compiler technology attempts to address this issue by *tiling* loop-nests separately for each level of the hierarchy [3, 8, 4]. In the dense numerical linear algebra community, there is growing interest in the use of block-recursive versions of numerical kernels like matrix multiply and Cholesky factorization to address the same problem. These algorithms recursively partition the original problem into ones with smaller working sets. This recursion has the effect of blocking the data at many different levels at the same time, so the data access patterns of the resulting codes exploit locality at all levels of the memory hierarchy. Experiments by Gustavson [6] and others have shown that these algorithms achieve substantial performance improvement over the tiled versions of the codes.

In this paper, we will use as examples block-recursive codes for two algorithms: matrix multiplication and Cholesky factorization. Consider the code fragment shown in Figure 1. It is an *iterative* version of Cholesky factorization that factories a symmetric positive definite matrix `A` such that $A = L \cdot L^T$ where L is a lower triangular matrix. In the code fragment shown here, the matrix L overwrites A. A block-recursive version of the algorithm can be obtained by sub-dividing the arrays A and L into 2×2 blocks and equating terms on both sides.

$$\begin{bmatrix} A_{00} & A_{10}^T \\ A_{10} & A_{11} \end{bmatrix} = \begin{bmatrix} L_{00} & 0 \\ L_{10} & L_{11} \end{bmatrix} \begin{bmatrix} L_{00}^T & L_{10}^T \\ 0 & L_{11}^T \end{bmatrix} = \begin{bmatrix} L_{00}L_{00}^T & L_{00}L_{10}^T \\ L_{10}L_{00}^T & L_{10}L_{10}^T + L_{11}L_{11}^T \end{bmatrix}$$

$$\begin{aligned} L_{00} &= chol(A_{00}) \\ L_{10} &= A_{10} / L_{00}^T \\ L_{11} &= chol(A_{11} - L_{10}L_{10}^T) \end{aligned}$$

Here $chol(X)$ computes the Cholesky factorization of array X. The recursive version performs a Cholesky factorization of the A_{00} block, then a division on the A_{10} block, and finally performs another Cholesky factorization on the updated A_{11} block. The termination condition for the recursion can either be a single element of A (degenerating

```
for j = 1, n
    for k = 1, j-1
        for i = j, n
S1:         A(i,j) -= A(i,k) * A(j,k)
S2: A(j,j) = dsqrt(A(j,j))
    for i = j+1, n
S3:     A(i,j) = A(i,j) / A(j,j)
```

Fig. 1. Cholesky Factorization

```
for j = 1, n
  for k = 1, n
    for i = 1, n
      C(i,j) -= A(i,k) * B(k,j)
```

Fig. 2. Matrix Multiplication

to square root operation) or to a $b \times b$ block of A which can be solved by the iterative code fragment.

A recursive version of matrix multiplication $C = AB$ can also be derived in a similar manner. The iterative code is shown in Figure 2. Subdividing the arrays into 2×2 blocks results in the following :-

$$\begin{bmatrix} C_{00} & C_{01} \\ C_{10} & C_{11} \end{bmatrix} = \begin{bmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{bmatrix} \begin{bmatrix} B_{00} & B_{01} \\ B_{10} & B_{11} \end{bmatrix} = \begin{bmatrix} A_{00}B_{00} + A_{01}B_{10} & A_{00}B_{01} + A_{01}B_{11} \\ A_{10}B_{00} + A_{11}B_{10} & A_{10}B_{01} + A_{11}B_{11} \end{bmatrix}$$

In the above formulation, each multiply results in eight recursive calls acting on sub-blocks. The natural order of traversal of a space in this recursive manner is called a *block-recursive* order and is shown in Figure 4 for a two-dimensional space. Unlike the Cholesky factorization case, however, the eight recursive calls in matrix multiplation can be performed in any order as they are all independent. Another way of ordering these calls is to make sure that one of the operands is reused between adjacent calls[1]. One such ordering corresponds to traversing the sub-blocks in the *gray-code* order. A gray-code order on the set of numbers $(1 \ldots m)$ would arrange the numbers so that adjacent numbers differ by exactly 1 bit in their binary representation. A gray-code order of traversing a 2-dimensional space is shown in Figure 5. Such an order is called *space-filling*, since the order traces a complete path through all the points, always moving from one point to an adjacent point. There are other space-filling orders, and some of them are described in the references [5]. Note that lexicographic order is not a space-filling order.

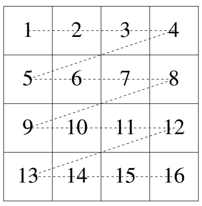


Fig. 3. Lexicographic Order

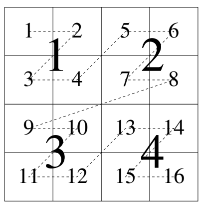


Fig. 4. Block Recursive Order

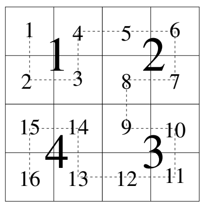


Fig. 5. Space-Filling Order

[1] Not more than one can be reused, in any case.

In this paper, we describe compiler technology that can automatically convert iterative versions of array programs into their recursive versions. In these programs, arrays are referenced via affine functions of the loop-index variables. As a result, partitioning the iterations of a loop will result in the partitioning of data as well. We exploit this insight as follows. We use affine mapping functions to map all the statement instances of the program to a space we call the *program iteration space*. This mapping effectively converts the program into a perfectly-nested loop-nest, with all statements nested in the innermost loop. We develop legality conditions under which the loops (which correspond to the dimensions of the space) can be recursively bisectioned. This corresponds to recursively bisectioning the dimensions of the program iteration space. Code is then generated to traverse the space in a block-recursive or space-filling manner, and when each point in this space is visited, the statements mapped to it are executed. This strategy effectively converts the iterative versions of codes into their recursive ones. The mapping functions that enable this conversion can be automatically derived when they exist. This approach does not require that the original program be in any specific form – any sequence of perfectly or imperfectly nested loops can be transformed in this way.

The rest of this paper is organized as follows. In Section 2, we give an overview of our approach to program transformation (details are in an associated technical report [1]). We derive legality conditions in Section 2.1 for recursively traversing the program iteration space, show how to generate recursive code in Section 3, and present experimental results in Section 4. Finally, we discuss ongoing work in Section 5.

2 The Program-Space Formulation

A program consists of statements contained within loops. All loop bounds and array access functions are assumed to be affine functions of surrounding loop indices. We will use S_1, S_2, ..., S_n to name the statements of the program in syntactic order.

A *dynamic instance* of a statement S_k refers to a particular execution of that statement for a given value of index variables i_k of the loops surrounding it, and is represented by $S_k(i_k)$. The execution order of these instances can be represented by a *statement iteration space* of $|i_k|$ dimensions, where each dynamic instance $S_k(i_k)$ is mapped to the point i_k. For the iterative Cholesky code shown in Figure 1, the statement iteration spaces for the three statements `S1`, `S2` and `S3` are $j_1 \times k_1 \times i_1$, j_2, and $j_3 \times i_3$ respectively.

The program execution order of a code fragment can be modeled in a similar manner by a *program iteration space*, defined as follows.

1. Let $\mathcal{P}$ be the Cartesian product of the individual statement iteration spaces of the statements in that program. The order in which this product is formed is the syntactic order in which the statements appear in the program. If p is the sum of the number of dimensions in all statement iteration spaces, then $\mathcal{P}$ is p-dimensional. $\mathcal{P}$ is also called the *product space* of the program.
2. Embed all statement iteration spaces $\mathcal{S}_k$ into $\mathcal{P}$ using embedding functions $\tilde{F_k}$ which satisfy the following constraints:
   (a) Each $\tilde{F_k}$ must be one-to-one.

(b) If the points in space $\mathcal{P}$ are traversed in lexicographic order, and all statement instances mapped to a point are executed in original program order when that point is visited, the program execution order is reproduced.

The program execution order can thus be modeled by the pair $(\mathcal{P}, \tilde{\mathcal{F}} = \{\tilde{F_1}, \tilde{F_2}, \ldots, \tilde{F_n}\})$. We will refer to the program execution order as the *original* execution order. For the Cholesky example, the program iteration space is a 6-dimensional space $\mathcal{P} = j_1 \times k_1 \times i_1 \times j_2 \times j_3 \times i_3$. One possible set of embedding functions $\tilde{\mathcal{F}}$ for this code is shown below –

$$\tilde{F_1}(\begin{bmatrix} j_1 \\ k_1 \\ i_1 \end{bmatrix}) = \begin{bmatrix} j_1 \\ k_1 \\ i_1 \\ j_1 \\ j_1 \\ i_1 \end{bmatrix} \quad \tilde{F_2}(\begin{bmatrix} j_2 \end{bmatrix}) = \begin{bmatrix} j_2 \\ j_2 \\ j_2 \\ j_2 \\ j_2 \\ j_2 \end{bmatrix} \quad \tilde{F_3}(\begin{bmatrix} j_3 \\ i_3 \end{bmatrix}) = \begin{bmatrix} j_3 \\ j_3 \\ i_3 \\ j_3 \\ j_3 \\ i_3 \end{bmatrix}$$

Note that all the six dimensions are not necessary for our Cholesky example. Examining the mappings shows us that the last three dimensions are redundant and the program iteration space could as well be 3-dimensional. For simplicity, we will drop the redundant dimensions when discussing the Cholesky example. The redundant dimensions can be eliminated in a systematic manner by retaining only those dimensions whose mappings are linearly independent.

In a similar manner, any other execution order of the program can be represented by an appropriate pair $(\mathcal{P}, \mathcal{F})$. Code for executing the program in this new order can be generated as follows. We traverse the entire product space lexicographically, and at each point of $\mathcal{P}$ we execute the original program with the statements protected by guards. These guards ensure that only the statement instances mapped to the current point are executed. For the Cholesky example, naive code which implements the execution order $(\mathcal{P}, \tilde{\mathcal{F}})^2$ is shown in Figure 6. In this code we traverse the entire product space (accounting for the `-inf` and `+inf`) lexicographically and at each point we execute the original code with the guards incorporating the mapping functions. This naive code can be optimized by using standard polyhedral techniques [7] to find the bounds of the outer loops and to remove the redundant loops. For example, we only need to traverse that part of the product space which has statement instances mapped to it. An optimized version of the code is shown in Figure 7. The conditionals in the innermost loop can be removed by index-set splitting the outer loops.

2.1 Traversing the Program Iteration Space

Not all execution orders $(\mathcal{P}, \mathcal{F})$ respect the semantics of the original program. An execution order must respect the *dependences* present in the original program if it is to be legal. A dependence is said to exist from instance i_s of statement S_s to instance i_d of statement S_d if both statement instances reference the same array location, at least one of them writes to that location and instance i_s occurs before instance i_d in original execution order. Since we are traversing the product space lexicographically, we require that the point that i_s is mapped to in the product space is lexicographically less than

[2] We have dropped the last three redundant dimensions for clarity.

```
for j1 = -inf to +inf
for k1 = -inf to +inf
for i1 = -inf to +inf
  for j = 1, n
    for k = 1, j-1
      for i = j, n
        if (j1==j && k1==k && i1==i)
S1:       A(i,j) -= A(i,k) * A(j,k)

    if (j1==j && k1==j && i1==j)
S2:   A(j,j) = dsqrt(A(j,j))

    for i = j+1, n
      if (j1==j && k1==j && i1==i)
S3:     A(i,j) = A(i,j) / A(j,j)
```

Fig. 6. Naive code for Cholesky

```
for j1 = 1,n
for k1 = 1,j1
for i1 = j1,n
    if (k1 < j1)
S1:    A(i1,j1) -= A(i1,k1) * A(j1,k1)

    if (k1==j1 && i1==j1)
S2:    A(j,j) = dsqrt(A(j1,j1))

    if (k1==j1 && i1 > j1)
S3:    A(i1,j1) = A(i1,1j) / A(j1,j1)
```

Fig. 7. Optimized code for Cholesky

the point that i_d is mapped to. In other words, the vector $v = F_d(i_d) - F_s(i_s)$ must be lexicographically positive for every pair (i_s, i_d) between which a dependence exists. We refer to v as the *difference vector*.

For a given legal embedding $\mathcal{F}$, the lexicographic traversal may not be the only legal order of traversing the product space. In particular, other legal orders of traversing the space can be obtained in the following ways.

1. Any order of walking the product space represented by a unimodular transformation matrix T is legal if $T \cdot v$ is lexicographically positive for every difference vector v associated with the code.
2. If the entries of all difference vectors corresponding to a set of dimensions of the product space are **non-negative**, then those dimensions can be blocked[3]. This partitions the product space into blocks with planes parallel to the axes of the dimensions. These blocks are visited in lexicographic order. When a particular block is visited, all points within that block are visited in lexicographic order as well. This order of traversal for a two-dimensional product space divided into equal-sized blocks is shown in Figure 3.
   Note that the points within each block do not need to be visited lexicographically. Any set of dimensions which can be blocked can also be *recursively blocked* i.e. each block can itself be further blocked and these inner blocks can either be traversed lexicographically or be further blocked recursively. If we choose to block the program iteration space by recursively bisecting block-dimensions then we obtain the block-recursive order shown in Figure 4.
3. If the entries of all difference vectors corresponding to a dimension of the product space are **zero**, then that dimension does not have to be traversed lexicographically — it can be traversed in any order. If a set of dimensions exhibit this property, then not only can those dimensions be blocked, but the blocks themselves do not have to be visited in a lexicographic order. In particular, the blocks of these dimensions can be traversed in a *space-filling* order. This principle can be applied recursively

[3] This is also called tiling.

within each block, to obtain space-filling orders of traversing the entire sub-space (Figure 5).

Given an execution order $(\mathcal{P}, \mathcal{F})$, and the dependences in the program, it is easy to check if the difference vectors exhibit the above properties using standard dependence analysis [9]. If we limit our embedding functions $\mathcal{F}$ to be affine functions of the loop-index variables and symbolic constants, we can determine functions which allow us to block dimensions (and hence also recursively block them) or to traverse a set of dimensions in a space-filling order. The condition that entries corresponding to a particular dimension of all difference vectors must be non-negative (for recursive-blocking) or zero (for space-filling orders) can be converted into a system of linear inequalities on the unknown coefficients of $\mathcal{F}$ by an application of *Farkas' Lemma* as discussed in [1]. If this system has solutions, then any solution satisfying the linear inequalities would give the required embedding functions. The embedding functions for the Cholesky example were determined by this technology.

3 Code Generation

Consider an execution order of a program represented by the pair $(\mathcal{P}, \mathcal{F})$. Let p represent the number of dimensions in the product-space. We wish to block the program iteration space recursively, terminating when blocks of size $B \times B \ldots \times B$ are reached.

For simplicity we will assume that redundant dimensions have been removed and that all dimensions can be blocked. We will also assume that all points in the program iteration space that have statement instances mapped to them are positive and that they are all contained in the bounding box $(1 \cdots B \times 2^{k_1}, \ldots, 1 \cdots B \times 2^{k_p})$. This can be ensured by choosing suitably large values of $k_1, \ldots k_p$.

Code to recursively traverse the product-space is shown in Figure 8. The procedure `Recurse` is parameterized with the current block to traverse given by `(lb[1]:ub[1], ..., lb[p]:ub[p])`. If the current block is not the base-block (the termination condition), `GenerateRecursiveCalls` subdivides the block into 2^p sub-blocks by bisecting each dimension and calls `Recurse` recursively in a lexicographic order[4]. If the termination condition is reached, code for the block is executed in `BlockCode`. The function `HasPoints` prevents the code from recursing into blocks that have no statement instances mapped to them. The initial call to `Recurse` is made with the lower and upper bounds set to the bounding box.

Naive code for `BlockCode(lb,ub)` is very similar to the naive code for executing the program. Instead of traversing the entire product space we only need to traverse the points in the current block lexicographically, and execute statement instances mapped to them. The redundant loops and conditionals can be hoisted out by employing polyhedral techniques.

We can identify blocks `(lb[1]:ub[1], ...,lb[p]:ub[p])` that contain points with statement instances mapped to them by creating a linear system of inequalities with variables lb_i, ub_i corresponding to each entry of `lb[1..p]`, `ub[1..p]` and variables x_i corresponding to each dimension of the product-space. Constraints are

[4] This must be changed appropriately if space-filling orders are required

```
Recurse(lb[1..p], ub[1..p])
if (HasPoints(lb,ub)) then
   if (∀i ub[i] == lb[i]+B-1) then
      BlockCode(lb)
   else
      GenerateRecursiveCalls(lb,ub,1)
   endif
endif
end
```

```
GenerateRecursiveCalls(lb[1..p], ub[1..p], q)
if (q > p)
   Recurse(lb, ub)
else
  for i = 1,p
     lb'[i] = lb[i]
     ub'[i] = (lb[i]+ub[i])/2
  GenerateRecursiveCalls(lb',ub',q+1)

  for i = 1, p
     lb'[i] = (lb[i]+ub[i])/2 + 1
     ub'[i] = ub[i]
  GenerateRecursiveCalls(lb',ub',q+1)
endif
end
```

Fig. 8. Recursive code generation

```
BlockCode(lb[1..3])
for j1 = lb[1], lb[1]+B-1
for k1 = lb[2], lb[2]+B-1
for i1 = lb[3], lb[3]+B-1
  for j = 1, n
    for k = 1, j-1
      for i = j, n
        if (j1==j && k1==k && i1==i)
S1:       A(i,j) -= A(i,k) * A(j,k)

    if (j1==j && k1==j && i1==j)
S2:   A(j,j) = dsqrt(A(j,j))

    for i = j+1, n
      if (j1==j && k1==j && i1==i)
S3:     A(i,j) = A(i,j) / A(j,j)
```

```
HasPoints(lb[1..3], ub[1..3])
if (lb[1]<=n && lb[2]<=n && lb[3]<=n
    && lb[1]<=ub[3]
    && lb[2]<=ub[1]
    && lb[2]<=ub[3])
    return true
else
    return false
end
```

Fig. 9. Recursive code for Cholesky

added to ensure that the point $(x_1, x_2, \ldots, x_p)$ has a statement instance mapped to it and that it lies within the block $(lb_1 : ub_1, \ldots, lb_p : ub_p)$. From the above system, we obtain the condition to be tested in `HasPoints(lb,ub)` by projecting out (in the Fourier-Motzkin sense) the variables x_i.

For our Cholesky example, the embedding functions shown in Section 2 allow all dimensions to be blocked. Since there are difference vectors with non-zero entries, the program iteration space cannot be walked in a space-filling manner, though it can be recursively blocked. The naive code for executing the code in each block is shown in Figure 9. As mentioned earlier, the redundant loops must be removed and the conditionals hoisted out for good performance. The part of the product-space that has statement instances mapped to it is $[j, k, i] : 1 \leq k \leq j \leq i \leq n$. This is used to obtain the condition in `HasPoints()`.

4 Experimental Results

In this section, we discuss the performance of block-recursive and space-filling codes produced using the technology described in this paper. The legality conditions discussed in Section 2.1 allow us to decide that the matrix multiply example (MMM, Figure 2) can be blocked both recursively as well as in a space-filling manner. The Cholesky code in Figure 1 can only be blocked recursively. We generated recursive code terminating in four different base-block sizes (16, 32, 64, 128) for both programs. The codes for these base-block sizes (`BlockCode(lb)`) were compiled with the "-O3 -

LNO:blocking=off" option of the SGI compiler. At this level of optimization, the SGI compiler performs tiling for registers and software-pipelining.

For each program, we ran the recursive (and if legal, the space-filling) versions of the code for a variety of matrix sizes For lack of space, we will only present results for a matrix size of 4000×4000. Results for other matrix sizes are similar. In the graphs, the results marked `Lexicographic` correspond to executing the code in `BlockCode(lb)` by visiting the base-blocks in a lexicographic manner. This has the effect of blocking (tiling) the program iteration space. We also show results of executing vendor-supplied hand-tuned implementations of matrix multiply (`BLAS`) and Cholesky (`LAPACK` [2]), for comparison. All experiments were run on an SGI R12K machine running at 300Mhz with a 32Kb primary-data cache (L1), 4Mb second-level cache (L2) and 64 TLB entries.

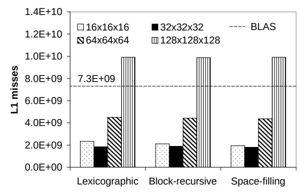


Fig. 10. MMM : L1 misses

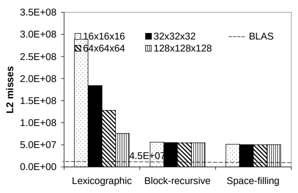


Fig. 11. MMM : L2 misses

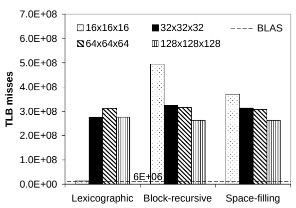


Fig. 12. MMM : TLB misses

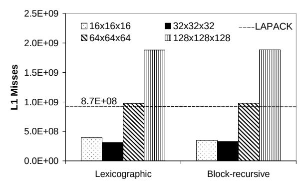


Fig. 13. CHOL : L1 misses

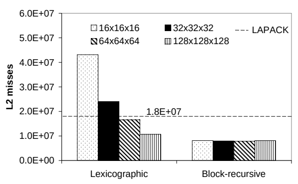


Fig. 14. CHOL : L2 misses

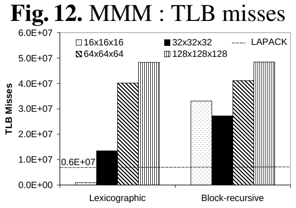
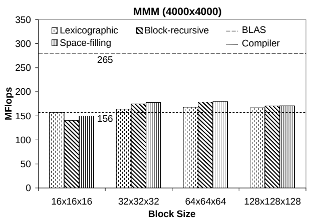


Fig. 15. CHOL : TLB misses

Figures 10 and 13 show the number of primary data cache misses for the two programs. For the larger block sizes (64, 128), the data touched by a base-block does not fit into cache (32K) and hence both the recursive and lexicographic versions suffer the same penalty. For smaller block sizes (16, 32), the data fits into cache resulting in much fewer misses. There is a small difference between the lexicographic and recursive versions for a block size of 16. The lexicographic versions have slightly more misses than the recursive versions since at this block size, the data touched by a base-block fits into less than 25% of the first-level cache. The recursive doubling effect aids the recursive-versions in using the cache more effectively. This effect is more pronounced for the second-level cache misses shown in Figures 11 and 14. The lexicographic versions for block sizes of 16 and 32 exhibit much higher miss numbers than the corresponding recursive versions since these block sizes are too small to fully utilize the 4M cache. In the recursive versions, however, even the small block sizes succeed in full utilization of the cache. These recursive versions will have a similar effect on any further levels of caches. Of the two recursive orders, the space-filling orders show slightly better cache performance for both programs.

Figures 12 and 15 show the number of TLB misses for the two programs. The R12K TLB has only 64 entries, hence large block sizes (more than 64) will exhibit high miss rates in both the lexicographic and recursive cases. Small block sizes could work well in the lexicographic case if the loop-order is chosen well. In our case, the `jki`-order is the best order for both the programs, and hence the case when block size is 16 has very few TLB misses as it uses less than 48 entries at a time. In the recursive case, the recursive doubling does cause significantly more TLB misses for small block sizes, although the recursive walks are largely immune to the effect of reordering the loops. For example, in the `jik`-order (not shown here), the 16 block size suffers a 100-fold increase in the number of TLB misses for the lexicographic case but remains roughly the same in the recursive cases.



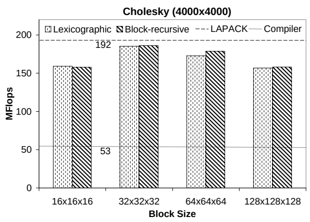


Fig. 16. Performance

Figure 16 shows the performance of the two programs in MFlops. As a sanity check, the lines marked `Compiler` show the performance obtained with compiling the original code with the "-O3" flag of the SGI compiler which attempts to tile for cache and registers and then software-pipeline the resulting code. For both programs, the recursive codes with block size of 32 are the best among all the generated code. For most block sizes, the recursive codes are better than their lexicographic counter-parts by a small percentage (2-5%). This is not the case when the block size is 16 because of the large number of TLB misses in the recursive cases for this block size.

The best recursive code generated is still substantially worse than the hand-tuned versions of the programs even though the recursive overhead is less than 1% in all cases. This difference could be due to the high number of TLB misses suffered by the recursive versions. Though the hand-tuned versions suffer higher primary cache miss rates, this does not seem to have affected performance much. This is not surprising in an out-of-order issue processor like the R12K where the latency (10 cycles) can be hidden by scheduling and software-pipelining. These misses will be more important in an in-order issue processor like the Merced.

5 Related Work and Conclusions

In this paper, we developed technology that automatically transforms iterative numerical programs into block-recursive versions. Previous work by Gustavson [10] in implementing IBM's Engineering and Scientific Subroutine Library (ESSL) has demonstrated the importance of recursive control strategies for dense linear algebra codes. Implementations of recursive algorithms for matrix multiply have also been studied by Sid Chaterjee [5].

Our experiments show that the block-recursive versions of matrix multiply and Cholesky do succeed in effectively blocking for all cache levels. Base-block sizes must be chosen with care – the data accessed in a base-block must fit into the lowest level of the cache hierarchy, the blocks must be large enough so that the recursive overhead is negligible, and the back-end compiler must be able to schedule the instructions in a base-block efficiently. Unfortunately, our experiments also show that the block-recursive algorithms do no interact well with the TLB. In spite of this, our best generated code for the two applications was still a recursive version. Better interaction with the TLB requires data copying which we have not implemented in our system.

The work in this paper can be extended in many ways. Further experiments are needed to assess the importance of block-recursive traversals for other applications like relaxation codes. Also, more experiments are needed to study the effect of non-square base-blocks. An interesting idea is to extend the recursion down to single elements, but unroll the inner recursive calls to expose operations which can then be scheduled without paying the recursion-overhead penalty. Finally, it would be interesting to study the effect of using recursive data layouts as suggested by Sid Chaterjee [5] or by copying data used by a base-block into contiguous locations as suggested by Gustavson [10].

References

1. Nawaaz Ahmed, Nikolay Mateev, and Keshav Pingali. Tiling imperfectly-nested loops. Technical Report TR2000-1782, Cornell University, Computer Science, January 2000.
2. E. Anderson, Z. Bai, C. Bischof, J. Demmel, J. Dongarra, J. Du Croz, A. Greenbaum, S. Hammarling, A. McKenney, S. Ostrouchov, and D. Sorensen, editors. *LAPACK Users' Guide. Second Edition*. SIAM, Philadelphia, 1995.
3. Steve Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Supercomputing*, 1992.
4. L. Carter, J. Ferrante, and S. Flynn Hummel. Hierarchical tiling for improved superscalar performance. In *International Parallel Processing Symposium*, April 1995.
5. S. Chatterjee, V. Jain, A. Lebeck, S. Mundhra, and M. Thottethodi. Nonlinear array layouts for hierarchical memory systems. In *International Conference on Supercomputing (ICS'99)*, June 1999.
6. F. G. Gustavson. Recursion leads to automatic variable blocking for dense linear-algebra algorithms. *IBM Journal of Research and Development*, 41(6):737–755, November 1997.
7. Wayne Kelly, William Pugh, and Evan Rosser. Code generation for multiple mappings. In *5th Symposium on the Frontiers of Massively Parallel Computation*, pages 332–341, February 1995.
8. Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *Programming Languages, Design and Implementation*. ACM SIGPLAN, June 1997.
9. William Pugh. The Omega test: A fast and practical integer programming algorithm for dependence analysis. In *Communications of the ACM*, pages 102–114, August 1992.
10. Joan McComb Ramesh C. Agarwal, Fred G. Gustavson and Stanley Schmidt. Engineering and Scientific Subroutine Library Release 3 for IBM ES/3090 Vector Multiprocessors. *IBM Systems Journal*, 28(2):345–350, 1989.