An Experimental Evaluation of Tiling and Shackling for Memory Hierarchy Management

Induprakas Kodukula, Keshav Pingali
Department of Computer Science,
Cornell University, Ithaca, NY 14853

Robert Cox, Dror Maydan
Silicon Graphics Inc.,
Mountain View, CA 94043.

Abstract

On modern computers, the performance of programs is often limited by memory latency rather than by processor cycle time. To reduce the impact of memory latency, the restructuring compiler community has developed locality-enhancing program transformations, the most well-known of which is *loop tiling*. Tiling is restricted to perfectly nested loops, but many imperfectly nested loops can be transformed into perfectly nested loops that can then be tiled. Recently, we proposed an alternative approach to locality enhancement called *data shackling*. Data shackling reasons about data traversals rather than iteration space traversals, and can be applied directly to imperfectly nested loops. We have implemented shackling in the SGI MIPSPro compiler which already has a sophisticated implementation of tiling. Our experiments on the SGI Octane workstation with dense numerical linear algebra programs show that shackled code obtains double the performance of tiled code for most of these programs, and obtains five times the performance of tiled code for some versions of Cholesky factorization. Data shackling has been integrated into the SGI MIPSPro compiler product-line.

1 Introduction

The performance of programs that touch large amounts of data is often limited by the cost of memory accesses. To reduce the impact of memory latency on program performance, the compiler community has developed an arsenal of locality-enhancement techniques that attempt to transform high-level programs into programs that exhibit good locality. A well-developed theory exists for perfectly nested loops (loop nests in which all assignment statements are contained in the innermost loop), and this theory recommends the use of linear loop transformations like permutation, followed by tiling [1, 3, 6, 9, 11, 13]. Locality-enhancement for *imperfectly nested* loops is less well-understood. Faced with imperfectly nested loops, compilers like the SGI MIPSPro try to apply other transformations like jamming, distribution and statement sinking [2, 12, 13] to convert these loops into perfectly nested loops if possible. However, there is no systematic theory for guiding this conversion, and the quality of the final tiled code may depend critically on how the conversion is done [7].

This approach to locality enhancement can be called *control-centric* program transformation because it reasons about the control structure (loop structure) of the program and modifies this control structure directly to enhance locality. A different approach to locality enhancement called *data-shackling* was proposed recently to address some of the limitations of control-centric approaches [8]. Instead of reasoning about control structures of programs, data shackling takes a *data-centric* approach by fixing an order of traversal through the data structures of the program, and determining which computations should be performed when a data item is touched. Intuitively, the compiler determines a schedule for the arrival of data at the highest level of the memory hierarchy, determines what computations should be performed when that data arrives, and generates the appropriate code.

While the data-centric compilation approach is intuitively appealing, it is not obvious that it must produce code with better locality than a conventional control-centric approach. A practical implementation has to automate key decisions (like the choice of block sizes) that were made manually in the experimental implementation of shackling described earlier [8]. Furthermore, locality enhancement is just one of many phases in a modern compiler which typically contains phases for inner-loop parallelization, software pipelining, instruction scheduling, register allocation etc. Since the benefits of increased locality are illusory if the transformed code is not felicitous for later phases of the compiler, it remains to be shown that data-centric transformations are useful in the context of a production compiler. Finally, it is interesting to compare the quality of compiler-generated code with that of handwritten code for memory hierarchies.

To address these concerns, we implemented data-shackling in the SGI MIPSPro compiler for the Octane workstation line. Existing modules in the SGI compiler were used for down-stream phases like register-tiling and instruction scheduling. The MIPSPro compiler already had a very sophisticated implementation of control-centric loop transformations for locality enhancement including linear loop transformations, tiling, loop jamming, statement sinking and loop distribution [12]. These two implementations were used to generate optimized code for the SGI Octane work-

[0]This work was supported by NSF grants CCR-9720211, EIA-9726388 and ACI-9870687.
Corresponding author: pingali@cs.cornell.edu

station for dense numerical linear algebra codes, and the performance of these two sets of compiler-generated programs was compared with that of hand-optimized LAPACK codes for these applications. To make robust comparisons, these experiments were performed with matrices of many sizes; also, several versions of Cholesky and LU factorizations were used.

2 Experimental setup

The experiments were conducted on an unloaded Octane workstation with an R10000 processor running at 195 MHz. The R10000 can perform one load/store operation and two floating point operations per cycle, giving it a theoretical peak performance of 390 MFlops. The processor has 32 logical registers and 64 physical registers. The workstation was equipped with separate first-level (L1) instruction and data caches of size 32Kb each, and a second-level (L2) unified cache of size 1MB. The L1 cache is non-blocking with a miss penalty of 10 cycles, and it is organized as a 2-way set associative cache with a line size of 32 bytes. The L2 cache is also non-blocking with a miss penalty of 70 cycles, and it is organized as a 2-way set associative cache with a line size of 128 bytes. Therefore, the four highest levels of memory hierarchy are the registers, the L1 and L2 caches and main memory.

2.1 Implementation of control-centric transformations

The performance numbers reported in this paper for the control-centric approach were obtained by using the unmodified MIPSPro compiler, and therefore reflect performance improvements from tiling for *both* levels of cache and from register optimizations. Code was compiled with the flags -O3 -n32 -mips4.

In the MIPSPro compiler, tiling is applied to one *Singly Nested Loops* (SNLs) at a time. A singly nested loop is a generalization of a perfectly nested loop in which atomic statements can appear at any nesting level; however, each loop still contains at most one other loop. The MIPSPro compiler tiles for both levels of cache, and also optimizes register use by unroll-and-jam, and then performing scalar replacement [12]. An important factor in the performance of tiled code is the size of the tile. The MIPSPro compiler uses heuristics for determining a good tile size for each loop nest [12].

2.2 Implementation of Shackling

Experiments done at SGI have shown that tiling for both levels of cache is usually not much better than tiling for the L2 cache alone for large problem sizes. This is because the miss penalty for the L1 cache is relatively small (10 cycles) and can usually be masked by ALU operations executed concurrently with servicing the miss. It is well-known that optimizing register usage is critical for performance. Therefore, shackling was implemented for the L2 cache only, and existing compiler modules were used for optimizing register usage.

The intuitive idea behind data-shackling is to make the compiler orchestrate the movement of data *directly* rather than *indirectly* as a result of loop transformations.

A *data shackle* is a three-part specification.

```
do b1 = 1 ..  ⌈(N/25)⌉
 do b2 = 1 ..  ⌈(N/25)⌉
  do I = 1 .. N
   do J = 1 .. N
    do K = 1 .. N
     if ((b1-1)*25 < I <= b1*25) &&
        ((b2-1)*25 < J <= b2*25)
          C[I,J] = C[I,J] + A[I,K] * B[K,J]
```

(a) Naive code produced by shackling C

```
do t1 = 1 ..  ⌈(N/25)⌉
 do t2 = 1 ..  ⌈(N/25)⌉
  do It = (t1-1)*25 +1 .. min(t1*25,N)
   do Jt = (t2-1)*25 +1 .. min(t2*25,N)
    do K = 1 .. N
     C[It,Jt] = C[It,Jt] + A[It,K] * B[K,Jt]
```

(b) Simplified code produced by IP tool

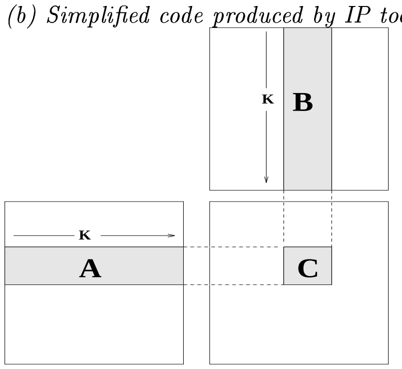


Figure 1: Code produced by shackling C in matrix-multiply

- One of the arrays in the program is divided into blocks using sets of parallel, equally spaced cutting planes.
- An order for visiting the blocks of data is determined.
- One reference to the array is selected for each statement in the program. This reference is called the data-centric reference for that statement.

Intuitively, the data shackle specifies an order in which blocks of the array are touched, and the data-centric reference is used to determine which instances of each statement are performed when a block of the array is touched — code is generated to perform all instances of that statement for which the data-centric reference touches data within the current block.

We illustrate this with matrix multiplication, assumed to be written in the usual `I,J,K` order of loops. One data shackle is obtained by dividing `C` into 2-dimensional blocks using horizontal and vertical sets of cutting planes. These blocks are visited in left-to-right, top-to-bottom order. `C(I,J)` is the only reference to this array in the assignment statement, and it is chosen to be the data-centric reference for that statement.

Figure 1(a) shows naive code generated by using this data shackle. There are two outermost loops which enumerate over the blocks of `C`. For each block, the entire input program is examined "looking for work to do". For the specified data shackle, this means that statement instances that write into the current block of `C` must be executed. This can be accomplished by inserting a suitable guard in front of the assignment statement, as shown in Figure 1(a). This code is shown only to illustrate the high-level idea of a data-shackle.

Standard integer linear programming tools can be used to produce the optimized code shown in Figure 1(b).

Within the context of a single block, iterations are done in the same order as in the original code (these are called *intra-block iterations*), but in the program as a whole, the order in which iterations are performed is different from their order in the source program. Therefore, shackling is not always legal. It is straight-forward to reduce the question of legality of a given shackle to the problem of determining whether an integer point is contained within a union of certain convex polyhedra [8], a problem which can be solved using standard technology already implemented in the MIPSPro compiler.

This shackle by itself does not produce the standard blocked matrix multiply in the literature. Shackling the `C(I,J)` reference causes the I and J loop indices to be constrained, for a given block of C, but the K index is not constrained in any way. This results in poor locality for the `A(I,K)` and `B(K,J)` references. This problem can be addressed by *composing shackles*. Intuitively, composition of shackles is the data-centric equivalent of nested loops. A single shackle specifies how the initial set of iterations is partitioned into groups, and specifies an order in which the groups of iterations are performed. However, there is no specification of how the iterations within a given group are to be performed. A second shackle can be used to refine each of these groups of iterations into finer ordered partitions. The second shackle is a refinement of the initial partition - if two iterations have already been ordered by the first ("outer") shackle, they are not reordered by the second ("inner") shackle. However, iterations assigned to the same group by the first shackle may be put into different ordered partitions. At the finest level of refinement, iterations assigned to a given composite block are performed in source program order. For matrix multiplication, we could block the `A` array, and choose the reference to `A` in the code as the data-centric reference. Composing these shackles gives us the same code as the standard blocked code for matrix multiplication.

For this example, the code produced by shackling is the same as the code produced by tiling. The appealing feature of the data-centric approach is that it makes no distinction between programs that are perfectly nested and program that are imperfectly nested, which is advantageous for programs like Cholesky factorization that have imperfectly nested loops that are not SNL's, as we show later in this paper.

2.2.1 Design decisions in implementing shackling

Implementing shackling in a production compiler raises the following questions.

1. Which is the scope of a data shackle?
2. What is the orientation of the cutting planes?
3. How are data-centric references chosen?
4. What is the separation of cutting planes (block sizes)?
5. What is the order of traversal of blocks?
6. What is the intra-block iteration order?

One approach to answering these questions is to treat them as classical optimization problems, and try to find optimal solutions given an accurate memory model. This approach is unlikely to be practical for use in a production compiler where speed of compilation is an important consideration. Therefore, the following simple heuristics have been developed to solve these problems.

1. In principle, shackling can be applied across multiple imperfectly nested loops and even across procedure boundaries by finding inter-procedural "program slices" [10] for the array blocks. To avoid complicating the implementation, the scope of transformation is restricted to a single imperfectly nested loop at a time. This is adequate for dense numerical linear algebra codes.
2. Cutting plane orientations are always parallel to the array co-ordinate axes. The use of skewed blocks often leads to inner loops with variable trip counts which is harmful for software pipelining.
3. Preference is given to data-centric references for which the access matrix [14] has the highest rank among all the references in that statement. In the presence of multiple references with highest rank, preference is given to a reference on the left hand side of the assignment statement, but otherwise the choice is arbitrary. Shackles are composed until no further benefit results from shackling [8]. For instance, in the matrix multiply example, it is recognized that after shackling `C[I,J]` and one of the other two references (say `A[I,K]`), further shackling does not have any benefit.
4. A key parameter for performance is block size. The model we use is based on the assumption that a cache has an *effective size*, such that if the amount of data in a cache is smaller than its effective size, conflict misses are relatively unimportant. The number of cache lines touched by the shackled code (called its *footprint*) is estimated as follows.
   (i) Statements that are most deeply nested in the imperfectly nested loop are determined, and in each such statement, references whose data access matrix has highest row-rank are identified. For matrix multiplication, we would select the references `C(i,j)`, `A(i,k)` and `B(k,j)`. In Cholesky factorization, we would select `A(i,j)`, `A(i,k)` and `A(j,k)` from the update step (see Figure 3). If `A(k,k)` appeared in the update step, it would not be chosen because the row rank of the data access matrix of this reference is just 1.
   (ii) For each statement considered in the previous step, selected references are partitioned into equivalence classes — two references are in the same equivalence class if they refer to the same array and their data access matrices have the same linear part (but possibly different affine parts). For example, `A(i+1,j)` and `A(i,j)` would be assigned to the same equivalence class, but `B(i,j)` would be placed in a different class. The assumption is that references in the same equivalence class will enjoy perfect group reuse.
   (iii) Equivalence classes from two different statements are then merged if (a) they refer to the same array, (b) the linear parts of the data access matrices are the same, and (c) the same data-centric reference is chosen for both the statements. The assumption is that under data-centric scheduling, instances of these statements that reference the same data will be scheduled close together so that they enjoy perfect group reuse.
   (iv) The last step computes a footprint for each equivalence class for a single instance of a composite shackle. The current implementation uses square blocks and uses the same block size for all arrays. The number of

distinct cache lines touched by a single equivalence class of references can be reduced using standard techniques to the problem of counting the number of integer solutions within a parameterized convex polyhedron. This number is estimated currently using the bounding box for the polyhedron, but more sophisticated solutions using Erhart polynomials are also possible [4]. The contributions from each equivalence class are added together and the maximum value of the block size for which the sum is less than the effective cache size of the L2 cache is computed.

5. The blocks of an array are visited in lexicographic order of the block co-ordinates. For example, for a two-dimensional array, the blocks are visited from left to right, and within a given block column, from top to bottom. If this order is not legal, reversals of the horizontal and vertical orders of traversal are tried. The analog of this reversal in the control-centric framework is loop reversal. If no legal traversal order is found, the loop nest is not shackled.
   The legality test is performed after block sizes have been determined, so nonlinear equations do not arise in this implementation. A potential drawback is that a shackle that is illegal for one choice of block size may be legal if a different block size is used. However, this appears to be unimportant in practice.
6. Statement instances that are not explicitly ordered by the shackles are performed in program order.

3 Basic Linear Algebra Subroutines

The core routines in dense numerical linear algebra are the Basic Linear Algebra Subroutines (BLAS). They can be classified as follows:

1. BLAS-3: routines for computing matrix multiplications (DGEMM)
2. BLAS-2: routines for computing matrix vector product (DGEMV).
3. BLAS-1: routines for computing the inner-product of two vectors (DDOT), and for scaling a vector and adding it to another vector (DAXPY).

Figure 2 shows the performance of hand-coded BLAS routines that are provided by SGI for Octane workstations[1]. For comparison, the performance of restructured codes generated by the SGI MIPSPro compiler from standard high-level FORTRAN programs for the BLAS is shown in the same figure (all loops are perfectly nested in these programs, and there is no difference between the shackled and tiled codes).

In matrix-multiplication, $O(n^3)$ operations are performed on $O(n^2)$ data, so there is excellent reuse of data. The hand-coded DGEMM routine obtains 300MFlops. Tiling does well as expected since the loop nest is perfectly nested, and obtains 240 MFlops. The small performance difference is due to differences in block size choices and unrolling factors. For matrix-vector product, only vector elements are reused, but spatial locality can be exploited in accessing matrix elements. Both the hand-written and compiler-generated BLAS-2 codes obtain roughly 100 MFlops for large problem sizes. Finally, inner-product and scaling of vectors has no reuse and the programs exhibit poor spatial locality if the vectors are not contiguous in memory.

[1] They were implemented by Mimi Celes at SGI.

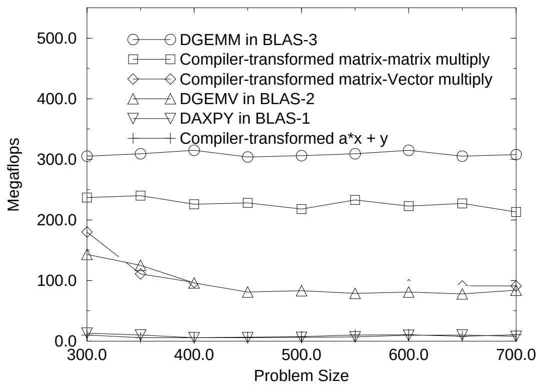


Figure 2: Performance of hand-coded and compiled BLAS routines

Both versions of DDOT and DAXPY perform at roughly 10 Mflops.

We conclude that compiler restructuring produces code competitive with handwritten code for the BLAS routines.

4 Cholesky factorization

Cholesky factorization is used to solve systems of equations of the form `Ax = b`, where `A` is a symmetric positive-definite matrix, by factorizing `A` into the product LL^T where `L` is lower-triangular, and solving the two resulting triangular systems. On the Octane, the LAPACK code runs at 260 MFlops for matrix sizes between 400 to 1200.

Like matrix multiplication, Cholesky factorization has three nested loops, but these loops are imperfectly nested. All six permutations of these three loops are legal and one of these permutations comes into two versions, giving a total of seven versions of the Cholesky program. Figures 3— 8 show pseudo-code for these versions. Figures 3(c)— 8(c) compare the performance of shackled and tiled codes for these versions of Cholesky factorization (in each of these figures, the bar on the left for each matrix size shows the performance of shackled code, while the bar on the right show the performance of tiled code). Ideally, a restructuring compiler would be able to generate the best code for Cholesky factorization from any of these versions of Cholesky factorization, just as many state-of-the-art restructuring compilers do not care which one of the six permutations of matrix multiplication is given as input. However, the theory of imperfectly nested loop transformations is not as well developed as the theory for perfectly nested loops. As we show below, the performance of code generated by the control-centric approach depends quite critically on which version of Cholesky is given as input. In principle, the data-centric approach does not care which version of Cholesky is given to it. However, since our implementation of the data-centric approach performs intra-block computations in the same order as in the input program, the performance of the shackled code does depend on which version is given as input although as we show below, the variation is less than it is for control-centric transformations.

The shackled code produced by the compiler was generated by composing two shackles. In both shackles, the array was divided into rectangular blocks (the compiler heuristic chose 70x70 blocks), and these blocks were visited in left-to-right, top-to-bottom order. In the outer shackle, the compiler chose the left-hand side reference from each assignment statement for shackling, while in the inner shackle, the compiler selected a reference from the right-hand side of each statement: `a(k,k)` for the square root statement, and `a(i,k)` for the scale and update statements. The same shackle was used for all other versions of Cholesky factorization as well.

Figures 3 and 4 show the performance of the *kij* versions of Cholesky factorization. The lower triangular portion of matrix `a` does not fit into the L2 cache when the matrix size exceeds 500, so the tiled code exhibits increasing miss ratios after this point. The fused version is an SNL, the loop structure for which the SGI compiler's locality enhancement techniques are targeted. Figures 3(b) and 4(b) show the effect of varying the block size in the shackled code. It can be seen the optimal block size is roughly 30 x 30 rather than the 70 x 70 chosen by the heuristic. With this block size, the performance of the shackled code is boosted to 240 Mflops which is very close to LAPACK performance. For the SNL in Figure 4, the SGI compiler's techniques are actually more effective than shackling in reducing L2 cache misses, although the miss ratios are very small in either case (the y-axis scales in Figures 3(d) and 4(d) are different). However, register tiling appears to interact better with the shackled code, permitting the shackled version to obtain much better overall performance. This shows the importance of register tiling for obtaining good performance, a point made by Carr and Kennedy earlier [2].

The *kij* versions update the right lower-triangular portion of the matrix row-by-row. Permuting the two update loops gives the *kji* version shown in Figure 5 that performs this update column-by-column. This version is not an SNL, so tiling is not effective. Fusing the scale loop with the outer update loop is illegal. The only way to get an SNL is to interchange the two update loops and then fuse the new outer update loop with the scale loop, generating the code of Figure 4, but this is too complicated for the MIPSPro compiler's imperfectly nested loop transformation heuristics to reason about. The performance of the baseline code (no cache or register tiling) is modestly better than that of the baseline *kij* versions because of better spatial locality in the update loops. This also explains why the shackled code performs a little better than the shackled code from the *kij* version.

Right-looking Cholesky factorization performs updates *eagerly* in the sense that the columns to the right of the current column are updated as soon as that column is computed. An alternative is to perform the updates *lazily*, which means that a column is updated only when it becomes current. This leads to the *left-looking column Cholesky factorization* code (also called the *jik* version) shown in Figure 6. This loop nest is not an SNL, but the computational work in the update loops is essentially a matrix-vector product which is performed by the MIPSPro compiler by accumulating the updates to `a(i,j)` in a register. The performance of the shackled version drops dramatically because of conflict misses when the array size is around 960. Since shackling reduces capacity misses only, it is of little benefit at this point as can be seen in Figure 6(c). Figure 6(b) shows that choosing the block size adaptively to reduce conflict misses is one solution, but our current implementation does not do that.

In the *jik* version, all the updates to an element of the current column are performed before succeeding elements are updated. Permuting the `i` and `k` loops gives the *jki* version. The MIPSPro compiler interchanges the update loops back to the *jik* version, so the performance of the baseline and tiled versions is identical to the performance of the *jik* versions. There is no difference in the performance of the shackled code either, so we do not show the graphs for this version of Cholesky.

Finally, there are two versions of Cholesky factorization called the *ijk* and *ikj* versions that process the matrix by row rather than by column. The `ijk` version performs inner-products, so it is also known as *ddot Cholesky* while the *ikj* version is rich in daxpy operations. Figure 7(c) shows that while the shackled code outperforms the tiled code, it is unable to exploit register tiling, so it performs poorly compared to the LAPACK code. The shackled code for the *ikj* version performs better, but it too exploits register tiling to a limited extent. Improving the performance of the *ikj* and ijk versions requires closer examination of the interaction of shackling with register tiling.

5 LU factorization

LU factorization is used to solve general systems of equations of the form `Ax = b` by factoring `A` into the product `LU` where `L` is a lower-triangular matrix and `U` is upper-triangular. Partial pivoting is used to increase the numerical stability of the procedure.

Figure 9 compares the performance of shackling and tiling for LU factorization with pivoting. The entire loop nest is not an SNL, and therefore cannot be tiled. However, the update loop nest can be tiled, and this has a small benefit because it permits spatial locality to be exploited.

Shackling the entire factorization code raises several interesting issues. Note the usage of scalar variables in the program - any shackling must ensure that dependences between scalar variables are preserved as well. In addition, the LU factorization code also involves conditionals with non-affine tests, so shackling must handle these as well.

Due to space constraints, only the highlights of how these issues are addressed (details can be found in [7]). If a statement defines a scalar variable that is used by another statement, the implementation tries to assign the same data-centric reference for both statements. If this is not legal, *scalar expansion* is performed to ensure that the flow dependence is respected. Scalar expansion is also necessary if different statements nested inside a non-affine conditional have different shackling references. In our implementation, scalar expansion was performed by using existing modules in the SGI MIPSPro compiler.

Using simple data-flow analysis, it can be determined automatically for the LU factorization code in Figure 9 that the scalar `m` needs to be expanded. The data shackle chosen by the compiler divides array `A` into block columns with block sizes ranging from 10 to 25 depending on the size of the problem. For the scale and update statements, the shackling references are chosen to be `a(i,k)` and `a(j,l)` respectively. For the three statements implementing the row permutations, the shackling references are `a(k,j)`, `a(k,j)` and `a(ipvt(k),j)` respectively, and for all the other statements, the shackling reference is `a(i,k)`. In this particular example, the expansion of `m` can be completely free, since `ipvt(k)` represents precisely a scalar expanded `m`; however this analysis is not currently implemented.

While the performance of the shackled code beats the performance of the tiled code, it is still slower than the LAPACK version which obtains about 200 MFlops on the Octane. This is because the LAPACK code uses domain-specific information about the commutativity of permutations and row-updates; this permits it in essence to use two-dimensional blocks rather than block columns, which results in better code. An interesting open question is how to write LU factorization with partial pivoting so that a compiler can determine this information automatically.

6 QR factorization

QR factorization performs orthogonal factorization of a matrix `A` into the product `QR` where `Q` is an orthonormal matrix and `R` is upper triangular. It is a key kernel in eigenvalue calculations. Figure 10 compares the performance of shackling and tiling on QR factorization using Householder reflections. As in the case of LU factorization with partial pivoting, the array `A` is partitioned into block columns because a two-dimensional blocking is not legal. QR is similar to LU factorization except that in this case, *array expansion* of the vector `x` is required for legality. The necessary array expansion has not yet been implemented, so we modified the standard code for QR factorization to perform array expansion. Figure 10 shows this program. The need to expand `x` raises an important profitability question - scalar expansion is usually quite cheap, however expanding `x` creates an array as large as `A` in this case. Although shackling once again outperforms tiling, the performance of the shackled code is a factor of 2 worse than that of the LAPACK code which obtains roughly 225 MFlops on this code. The LAPACK code uses domain-specific information about the associativity of matrix products to improve efficiency. It is conceivable that a compiler could exploit this information too if the input program were written in a language like MATLAB or FORTRAN-90 in which array operations are primitives.

7 Conclusions and Future Work

The experiments reported in this paper have demonstrated the effectiveness of shackling for improving locality of numerical codes, and have shown the utility of data-centric transformations in a production compiler. Shackled code obtains double the performance of tiled code for most of these programs, and obtains five times the performance of tiled code for some versions of Cholesky factorization. Shackling has been incorporated into the SGI MIPSPro compiler product-line as of January 1999.

There are a number of areas where more work is needed. First, the block size heuristic we have implemented clearly over-estimates block sizes, and must be improved. Second, conflict misses cause dramatic drops in performance for some matrix sizes. Choosing block sizes adaptively may be one solution. Coleman and McKinley have studied this problem in the context of tiling [5], and their techniques may be applicable here. Third, the interactions between cache locality-enhancement and register tiling need to be understood better. Miss ratios alone do not give a complete picture of performance since the effectiveness of register tiling is often an equally important determinant of performance (see Figure 4). Finally, for LU and QR factorizations, the lack of domain-specific information prevents the compiler from restructuring code along the lines of the LAPACK library. Coding in an array language might address some of these problems but this remains to be investigated. User directives are another option, but it is unclear what these directives should be or how they might be exploited by the compiler.

References

[1] U. Banerjee. Unimodular transformations of double loops. In *Proceedings of the Workshop on Advances in Languages and Compilers for Parallel Processing*, pages 192–219, August 1990.

[2] Steve Carr and K. Kennedy. Compiler blockability of numerical algorithms. In *Supercomputing*, 1992.

[3] Steve Carr and R. B. Lehoucq. Compiler blockability of dense matrix factorizations. Technical report, Argonne National Laboratory, Oct 1996.

[4] Phillipe Claus. Counting solutions to linear and nonlinear constraints through Erhart polynomials. In *"ACM International Conference on Supercomputing*. ACM, May 1996.

[5] Stephanie Coleman and Kathryn S. McKinley. Tile size selection using cache organization and data layout. In *ACM SIGPLAN '95 Conference on Programming Language Design and Implementation (PLDI)*. ACM Press, June 1995.

[6] Jack Dongarra and Robert Schreiber. Automatic blocking of nested loops. Technical Report UT-CS-90-108, Department of Computer Science, University of Tennessee, May 1990.

[7] I. Kodukula. *Data-centric Compilation*. PhD thesis, Cornell University, 1998.

[8] Induprakas Kodukula, Nawaaz Ahmed, and Keshav Pingali. Data-centric multi-level blocking. In *Programming Languages, Design and Implementation*. ACM SIGPLAN, Jun 1997.

[9] W. Li and K. Pingali. Access Normalization: Loop restructuring for NUMA compilers. *ACM Transactions on Computer Systems*, 1993.

[10] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, 10(4):352–357, 1984.

[11] M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. In *SIGPLAN 1991 conference on Programming Languages Design and Implementation*, Jun 1991.

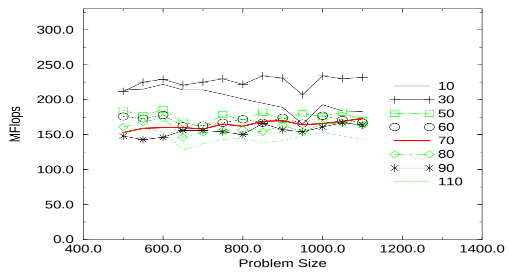
[12] Michael E. Wolf, Dror E. Maydan, and Ding-Kai Chen. Combining loop transformations considering caches and scheduling. In *MICRO 29*, pages 274–286, Silicon Graphics, Mountain View, CA, 1996.

[13] M. Wolfe. Iteration space tiling for memory hierarchies. In *Third SIAM Conference on Parallel Processing for Scientific Computing*, December 1987.

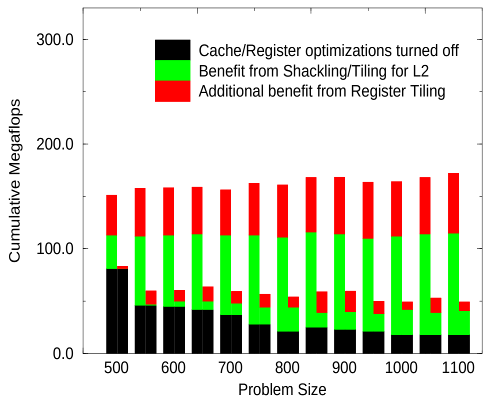
[14] M. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Publishing Company, 1995.

```
do k = 1, NMAX
  a(k,k) = dsqrt (a(k,k))
  do i = k+1, NMAX
    a(i,k) = a(i,k) / a(k,k)
  do i = k+1, NMAX
    do j = k+1, i
      a(i,j) -= a(i,k) * a(j,k)
```

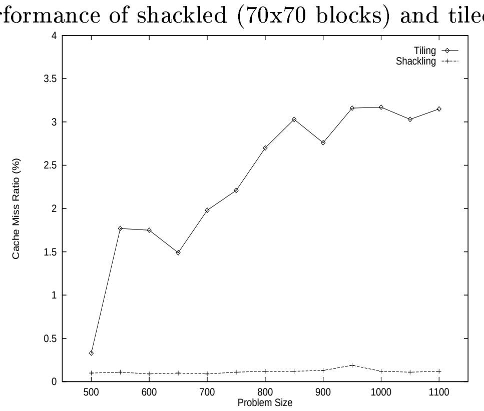
(a) Input code



(b) Performance of shackled code (varying block sizes)



(c) Performance of shackled (70x70 blocks) and tiled codes

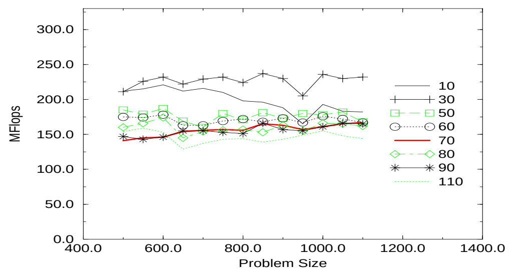


(d) L2 cache miss ratios

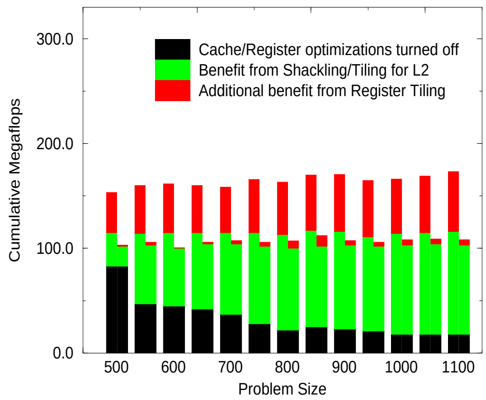
Figure 3: Cholesky Factorization: kij version (distributed)

```
do k = 1, NMAX
  a(k,k) = dsqrt (a(k,k))
  do i = k+1, NMAX
    a(i,k) = a(i,k) / a(k,k)
    do j = k+1, i
      a(i,j) -= a(i,k) * a(j,k)
```

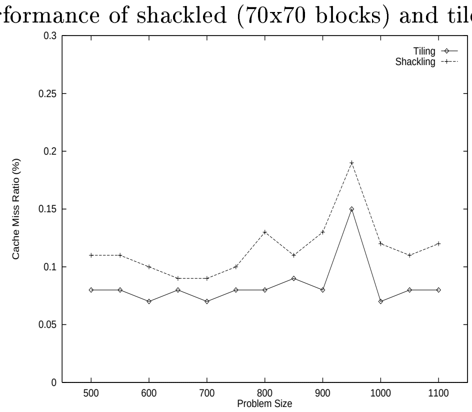
(a) Input code



(b) Performance of shackled code (varying block sizes)



(c) Performance of shackled (70x70 blocks) and tiled codes

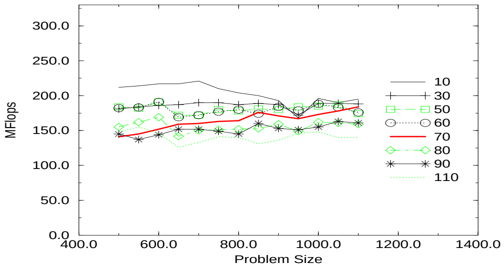


(d) L2 cache miss ratios

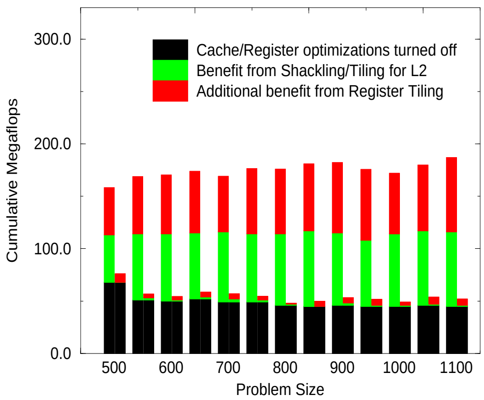
Figure 4: Cholesky Factorization: kij version (fused)

```
do k = 1, NMAX
  a(k,k) = dsqrt (a(k,k))
  do i = k+1, NMAX
    a(i,k) = a(i,k) / a(k,k)
  do j = k+1, NMAX
    do i = j, NMAX
      a(i,j) -= a(i,k) * a(j,k)
```

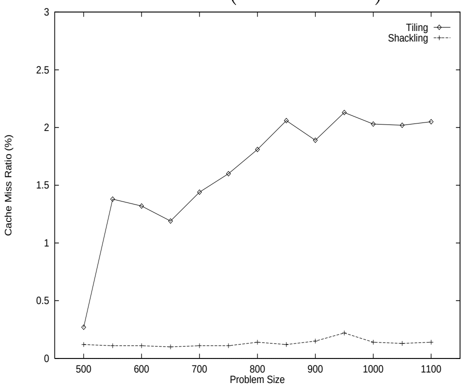
(a) Input code



(b) Performance of shackled code (varying block sizes)



(c) Performance of shackled (70x70 blocks) and tiled codes

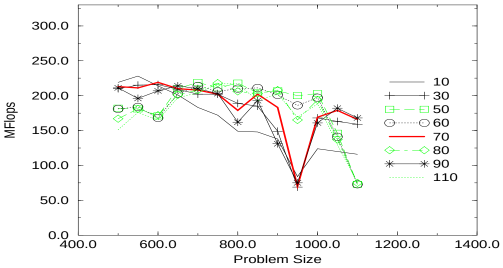


(d) L2 cache miss ratios

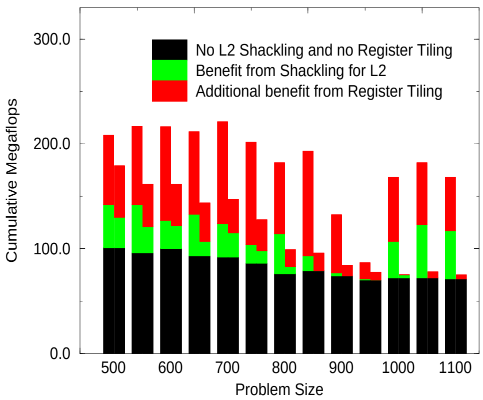
Figure 5: Cholesky Factorization: kji version

```
do j = 1, NMAX
  do i = j, NMAX
    do k = 1, j-1
      a(i,j) -= a(i,k) * a(j,k)
   a(j,j) = dsqrt (a(j,j))
   do i = j+1, NMAX
     a(i,j) = a(i,j) / a(j,j)
```

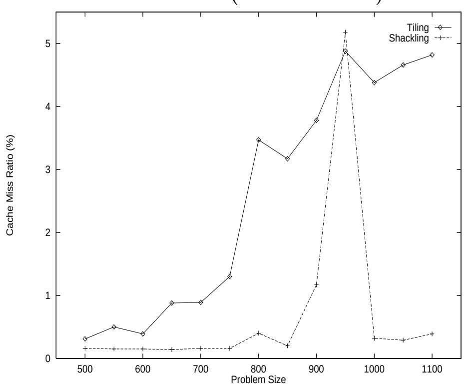
(a) Input code



(b) Performance of shackled code (varying block sizes)



(c) Performance of shackled (70x70 blocks) and tiled codes

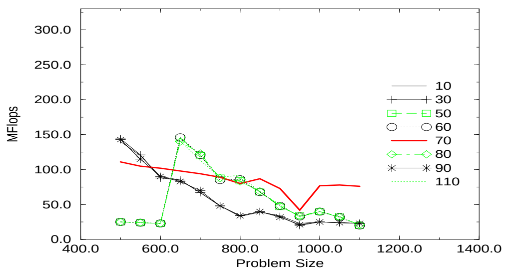


(d) L2 cache miss ratios

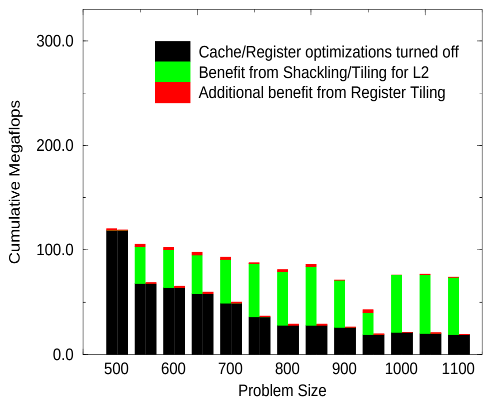
Figure 6: Cholesky Factorization: Version jik

```
do i = 1, NMAX
  do j = 1, i-1
    do k = 1, j-1
      a(i,j) -= a(i,k) * a(j,k)
    a(i,j) = a(i,j)/a(j,j)
  do k = 1, i-1
      a(i,i) -= a(i,k) * a(i,k)
  a(i,i) = dsqrt (a(i,i))
```

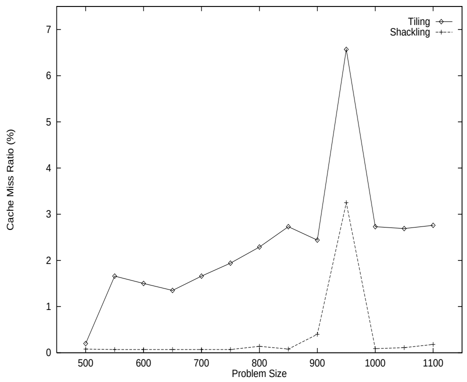
(a) Input code



(b) Performance of shackled code (varying block sizes)



(c) Performance of shackled (70x70 blocks) and tiled codes

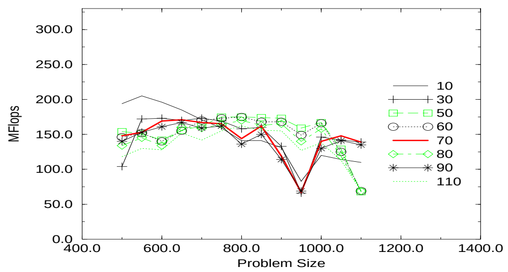


(d) L2 cache miss ratios

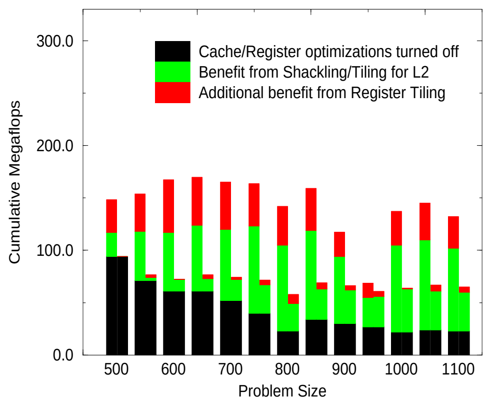
Figure 7: Cholesky Factorization: Version ijk

```
do i = 1, NMAX
  do k = 1, i-1
    a(i,k) = a(i,k) / a(k,k)
    do j = k+1, i
      a(i,j) -= a(i,k) * a(j,k)
  a(i,i) = dsqrt (a(i,i))
```

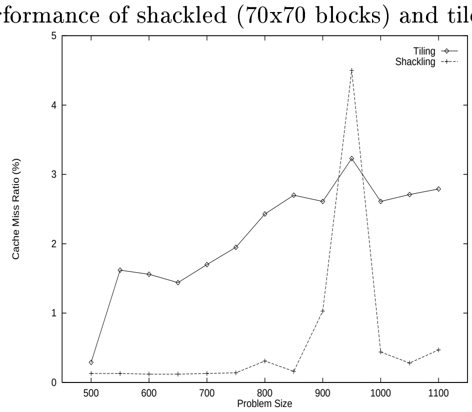
(a) Input code



(b) Performance of shackled code (varying block sizes)



(c) Performance of shackled (70x70 blocks) and tiled codes

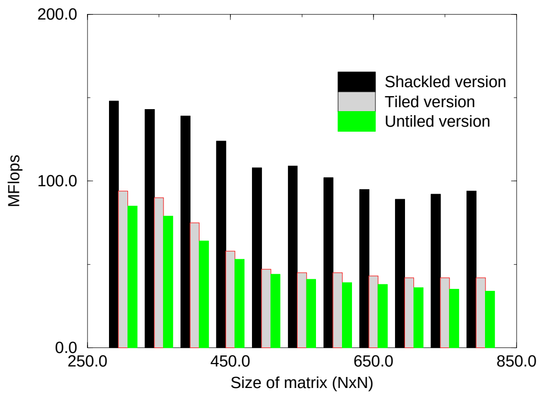


(d) L2 cache miss ratios

Figure 8: Cholesky Factorization: Version ikj

```
do k = 1, n
  temp = 0.0d0
  m = k
   //find pivot row
  do i = k, n
    d = a(i,k)
    if (ABS (d) .gt. temp)
      temp = abs(d)
      m = i
  if (m .ne. k)
    ipvt(k) = m
     //row permutation
    do j = k, n
      temp = a(k,j)
      a(k,j) = a(ipvt(k),j)
      a(ipvt(k),j) = temp
   //scale loop
  do i = k+1, n
    a(i,k) = a(i,k) / a(k,k)
   //update loops
  do j = k+1, n
    do l = k+1, n
      a(j,l) -= a(j,k) * a(k,l)
```

(a) LU factorization with Partial Pivoting

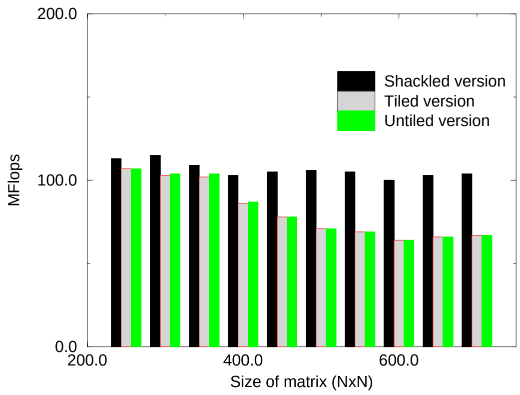


(b) Performance of shackled and tiled Codes

Figure 9: LU Factorization with Partial Pivoting

```
do i = 1, n
  norm = 0
  do j = i, n
    norm = norm + A(j,i) * A(j,i)
  norm2 = dsqrt (norm)
  asqr = A(i,i) * A(i,i)
  A(i,i) =
    dsqrt(norm-asqr+((A(i,i)-norm2)^2))
  do j = i+1, n
    A(j,i) = A(j,i) / A(i,i)
  do j = i+1, n
    x(j,i) = 0
    do k = i, n
      x(j,i) += A(k,i) * A(k,j)
  do j = i+1, n
    do k = i+1, n
      A(k,j) = A(k,j) - A(k,i) * x(j,i)
```

(a) QR factorization



(b) Performance of shackled and tiled Codes

Figure 10: QR Factorization using Householder reflections