A Loop Restructuring Research Tool

Michael Wolfe

Oregon Graduate Institute
Department of Computer Science
and Engineering
19600 N.W. von Neumann Drive
Beaverton, OR 97006-1999 USA

Technical Report No. CS/E 90-014

August, 1990

A Loop Restructuring Research Tool

Michael Wolfe

Oregon Graduate Institute of Science and Technology
Department of Computer Science and Engineering
19600 NW von Neumann Drive
Beaverton, OR 97006
(503)-690-1153
mwolfe@cse.ogi.edu

A Loop Restructuring Research Tool

Abstract

Program restructuring, or more precisely, loop restructuring is often proposed as a way to automatically (or manually) improve the performance of scientific programs on high performance computer systems. Loop restructuring transformations have been proposed to detect loops that can execute in parallel or vector mode, to interchange loops to improve parallelism or memory hierarchy performance, and to regularize interprocessor communication patterns. The benchmarking process undertaken by the Perfect Club is one of identifying software transformations that will improve performance; many of these are loop restructuring transformations. We have begun a research program to explore the potential of loop restructuring transformations from three points of view: (1) What kinds of transformations can be performed? (2) What information is necessary to decide when a transformation is legal? (3) What performance improvement can be attained from each transformation? To support this research we have begun implementation of a loop restructuring tool. This paper describes the goals of the research and some experiences with the tool.

A Loop Restructuring Research Tool

1. Introduction

Many loop restructuring transformations have been proposed to increase parallelism or otherwise improve performance on advanced computer architectures. We present a table of some of these below; In each case we give a representative citation (instead of an exhaustive list); a summary of many of these transformations is available in [Wol89b].

	transformation	enhances	reference
•	vectorization	parallelism	[Sch72]
•	parallelization	parallelism	[AlK82]
•	strip mining	vectorization	[Lov77]
•	distribution	vectorization	[AlK87]
•	interchanging	parallelism	[AlK84]
•	interchanging	memory	[GJG88]
	skewing	parallelism	[Wol86b]
	tiling	memory	[AKL81]
	reversal	interchanging	[Wol82]
	alignment	parallelism	[ACK87]
	splitting	parallelism	[Ban79]

An entry of "enhances memory" means the transformation enhances the performance of memory hierarchies. Those listed above with an bullet are implemented in commercial language products (reversal was implemented in one of the TI ASC compilers [Wed75]). Other transformations have not been shown to be cost-effective, considering the cost to implement, debug and support more advanced transformations, the cost to train users how to write programs to take advantage of the new transformations (and how to understand or debug the code generated by the compiler or system), and the compile-time cost of attempting each transformation on each program. This latter point is especially touchy; the argument that spending a little more time compiling is worth the savings in execution time generally falls on deaf ears. All the good intentions of compiler and programming environment researchers and developers won't sell a slow tool to users.

Other transformations have also been defined from time to time, but have not been implemented or have been used only in research. Several research efforts (notably systolic array synthesis research [GoT88,LeK90]) have looked at the problem of mapping the index set of a nested loop computation via linear transformations into a transformed index set that satisfies certain properties, generally dealing with parallelism and interprocessor communication patterns. While linear index set transformations can also be implemented as loop restructuring transformations, the reverse is not always true; in addition, systolic array and other research into index set mapping generally assumes some idealized form of the program, such as tightly nested loops with simple data references and constant dependence vectors. It has been shown that any systolic algorithm can be written in the form of a nested loop algorithm, but that does not mean that they *should* be written in such a most primitive form (e.g., we can argue that any computer program can be written using an idealized assembler language, but that does not mean we should write all our programs in MIX [Knu73]).

We believe that loop restructuring is a powerful method to improve the performance characteristics of a program by matching the program to complex architectures. Elementary transformations can be combined in interesting ways to produce dramatic speed enhancements that take advantage of parallelism, novel memory hierarchies and high speed interprocessor connection networks. Some of the research in to loop restructuring has already found its way into daily use in the commercial world; however we have not experimented with other potentially interesting restructuring transformations. In order to allow experimentation with loop restructuring as a craft in itself (without necessarily worrying about immediate commercial applications), we have begun work on a loop restructuring research tool.

2. Basic Loop Restructuring: Loop Interchanging

The evolution of loop interchanging is interesting since it was originally conceived as a method of enhancing parallelism, but has since been used for many other performance enhancement purposes. In the early days of vectorizing compilers, users often manually interchanged loops to improve the vectorization. Vectorization typically considers only the innermost loop; in the program fragment:

```
for i = 1 to n do
  for j = 2 to m do
    a(i,j) = a(i,j-1) + b(i,j)
  endfor
endfor
```

the j loop cannot be vectorized, since a(i,j-1) will use the value assigned to a(i,j) from the previous iteration of the j loop. This is called a *data dependence relation*, and we say that the statement *depends* on itself. Of particular interest is the dependence distance, that is, the number of iterations that the dependence crosses. In this case, the dependence distance is one, since a(i,j-1) depends on the value assigned to a(i,j) from the immediately previous iteration. Users learned quickly to recognize cases such as this and to interchange these two loops:

```
for j = 2 to m do
  for i = 1 to n do
    a(i,j) = a(i,j-1) + b(i,j)
  endfor
endfor
```

After interchanging, the program has no dependence relations preventing vectorization of the inner i loop.

In the late 1970's and early 1980's, research efforts at the University of Illinois and Rice University developed compiler technology that could automatically detect when it was legal to interchange loops [AlK84,Wol78,Wol82]. This technology uses the concept of dependence distance, but with a separate dependence distance computed for each surrounding loop. In the original program fragment above, for instance, a compiler would find a *distance vector* comprising two elements, the first for the I loop and another for the J loop. The distance in the I loop is zero, while the distance in the J loop is one (as before); the distance vector is then (0,1). Sometimes, due to complicated subcripting patterns or inexact testing algorithms, a precise distance vector cannot be computed; in these cases, a compiler might find a *direction vector* which comprises the signs of the possible distances. Each element of a direction vector would be either +, 0 or -, or a combination of these. For historical reasons, these are usually called <, = and >, respectively. If the distance vector comprised all zero entries, then we call that a *loop-independent dependence;* this corresponds to a direction vector of all-= entries. Otherwise there is a non-zero entry in the distance vector; in this case we say that the loop corresponding to the first (leftmost) non-zero entry in the distance vector *carries* the dependence, and that this is a *loop carried dependence*. An loop can be *vectorized* if the data dependence graph is acyclic, and can be *parallelized* if that loop carries no dependence relations [ACK87,AlK87].

It is shown in the literature that two loops cannot be interchanged if the outer loop carries some dependence relation for which the dependence distance in the inner loop is negative. Compiler research tools then implemented loop interchanging dependence tests, and a significant (but not overwhelming) amount of additional parallelism was uncovered; at that time, commercial compilers shunned automatic interchanging, claiming it was too expensive for little or no real gain in performance.

However, it soon became obvious that interchanging loops could have a dramatic effect on the execution behavior of the program. For instance, it not only enables parallelism at a different loop level, it changes the loop limits of the inner loop (allowing longer vector operations, for instance), changes the memory strides for array accesses, and may even allow additional code floating. Now almost all commercial vectorizing and parallelizing compilers include automatic loop interchanging as a standard feature.

Loop interchanging by itself has made it into the commercial world; in combination with other transformations, it is even more powerful. Independent work on ***loop blocking*** or ***tiling*** found that proper partitioning of the iteration space (index set) of a nested loop into blocks or tiles could greatly enhance the locality of reference for data access in the inner loop levels [AKL81,GJG88]. It was soon realized that tiling could be viewed (from a compiler point of view) as nothing more than a combination of strip mining (or sectioning) and interchanging [Lov77,Wol89a]. Additionally, there are loops where neither the inner nor the outer loop can be executed in parallel; in these cases, the ***wavefront method*** or ***hyperplane method*** was proposed to allow parallel execution of a reindexed loop [Lam75]. It has been shown that the wavefront method can be viewed as nothing more than a combination of loop skewing and interchanging, again combining two elementary loop transformations to achieve a powerful result [Wol86b]. Some of these advanced applications of loop interchanging are now making their way into commercial products.

Loop interchanging is in many ways the first real loop restructuring transformation; interchanging can have a dramatic effect on the performance and behavior of a program, and the benefits far outweigh the costs associated with implementing and supporting it. Other loop restructuring transformations are not so well understood and still seem too costly to merit much consideration. We believe further research is justified into these costs and potential benefits by further experimentation into loop restructuring.

3. Advanced Loops Restructuring Examples

Some early experiments in advanced restructuring were described in a prior paper [Wol86a]. Our goals now are somewhat more ambitious. (The program examples in this section are taken from screen dumps of the research tool; the line numbers to the left correspond to source code line numbers).

First Example. Our first example is a simple three-dimensional 6-point difference equation:

```
3: for k = 2,n do
4:  for i = 2,n do
5:   for j = 2,n do
6:    a(k,i,j) = a(k,i-1,j)+a(k,i,j-1)+a(k,i,j+1)+
                 a(k,i+1,j)+a(k-1,i,j)+a(k+1,i,j)
5:   endfor
4:  endfor
3: endfor
```

The data dependence relations in this loop are shown below; the kind of dependence (anti or flow), the statements involved (all the same statement in this example), the direction and distance vector as well as the array references involved are given:

```
anti dependence:  6 --> 6  (=,=,<)  (0,0,1)  a(k,i,j+1) --> a(k,i,j)
anti dependence:  6 --> 6  (=,<,=)  (0,1,0)  a(k,i+1,j) --> a(k,i,j)
anti dependence:  6 --> 6  (<,=,=)  (1,0,0)  a(k+1,i,j) --> a(k,i,j)
flow dependence:  6 --> 6  (=,<,=)  (0,1,0)  a(k,i,j)  --> a(k,i-1,j)
flow dependence:  6 --> 6  (=,=,<)  (0,0,1)  a(k,i,j)  --> a(k,i,j-1)
flow dependence:  6 --> 6  (<,=,=)  (1,0,0)  a(k,i,j)  --> a(k-1,i,j)
```

An anti-dependence is a "use-def" ordering, while a flow-dependence is a "def-use" ordering. While there are no dependence relations preventing interchanging the loops any way we want, each loop carries two dependence relations; this means none of the loops can execute in parallel. The standard solution is to use a wavefront method; we derive the wavefront method of execution by using program transformations. First, we ***skew*** the inner loop with respect to each of the outer loops. The first step adds I to the lower and upper limits of the J loop (and corrects within the loop):

```
3: for k = 2,n do
4:  for i = 2,n do
5:   for j = 2+i,n+i do
6:    a(k,i,j-i) = a(k,i-1,j-i)+a(k,i,j-i-1)+a(k,i,j-i+1)+
                   a(k,i+1,j-i)+a(k-1,i,j-i)+a(k+1,i,j-i)
5:   endfor
4:  endfor
3: endfor
```

Forward skew loop j with respect to i

The next step adds K to the J loop limits:

```
3: for k = 2,n do
4:  for i = 2,n do
5:   for j = 2+i+k,n+i+k do
6:    a(k,i,j-k-i) = a(k,i-1,j-k-i)+a(k,i,j-k-i-1)+a(k,i,j-k-i+1)+
                     a(k,i+1,j-k-i)+a(k-1,i,j-k-i)+a(k+1,i,j-k-i)
5:   endfor
4:  endfor
3: endfor
```

Forward skew loop j with respect to k

Finally we interchange the J loop all the way outwards; notice that since the loop limits for the J loop depend on I and K, we have to adjust the limits to cover the same index space:

```
5: for j = 2+2+2,n+n+n do
3:  for k = max(2,j-(n+n)),min(n,j-(2+2)) do
4:   for i = max(2,j-(n+k)),min(n,j-(2+k)) do
6:    a(k,i,j-k-i) = a(k,i-1,j-k-i)+a(k,i,j-k-i-1)+a(k,i,j-k-i+1)+
                     a(k,i+1,j-k-i)+a(k-1,i,j-k-i)+a(k+1,i,j-k-i)
4:   endfor
3:  endfor
5: endfor
```

Interchanging loops k and j

Inspection of the new dependence graph after these transformations tells us that the outer loop, the J loop, carries all the dependence relations:

```
anti dependence:  6 --> 6  (<,=,=)  (1,0,0)  a(k,i,j-k-i+1) --> a(k,i,j-k-i)
anti dependence:  6 --> 6  (<,=,<)  (1,0,1)  a(k,i+1,j-k-i) --> a(k,i,j-k-i)
anti dependence:  6 --> 6  (<,<,=)  (1,1,0)  a(k+1,i,j-k-i) --> a(k,i,j-k-i)
flow dependence:  6 --> 6  (<,=,<)  (1,0,1)  a(k,i,j-k-i) --> a(k,i-1,j-k-i)
flow dependence:  6 --> 6  (<,=,=)  (1,0,0)  a(k,i,j-k-i) --> a(k,i,j-k-i-1)
flow dependence:  6 --> 6  (<,<,=)  (1,1,0)  a(k,i,j-k-i) --> a(k-1,i,j-k-i)
```

Thus, the two inner loops can be executed in parallel:

```
5: for j = 2+2+2,n+n+n do
3:  doall k = max(2,j-(n+n)),min(n,j-(2+2)) do
4:   doall i = max(2,j-(n+k)),min(n,j-(2+k)) do
6:    a(k,i,j-k-i) = a(k,i-1,j-k-i)+a(k,i,j-k-i-1)+a(k,i,j-k-i+1)+
                     a(k,i+1,j-k-i)+a(k-1,i,j-k-i)+a(k+1,i,j-k-i)
4:   endfor
3:  endfor
5: endfor

Parallelize loop k
```

Second Example. Our Second example problem is to generate (via loop restructuring) all six versions of Cholesky decomposition from just one of them. Cholesky decomposition is similar is the LL^T decomposition of a symmetric positive definite matrix, and uses an algorithm much like LU decomposition. The base program we use is the KIJ form:

```
3: for k = 1,n do
4:  a(k,k) = sqrt(a(k,k))
5:  for i = k+1,n do
6:   a(i,k) = a(i,k)/a(k,k)
7:   for j = k+1,i do
8:    a(i,j) = a(i,j)-a(i,k)*a(j,k)
7:   endfor
5:  endfor
3: endfor
```

As with matrix multiplication, there are six distinct forms of the Cholesky decomposition, named for the 3! possible loop orderings around the innermost assignment. We will attempt to derive the other 5 loop orderings via loop interchanging. Unlike matrix multiplication, this program has non-tightly nested loops. The standard method to deal with interchanging non-tightly nested loops is to distribute the outer loop around the inner loop, and interchange the generated tightly nested loops. However, in this program, the distribution of the K loop (for instance) is not legal. Fortunately we have already devised the data dependence tests necessary to check for the cases when non-tightly nested loop can legally be interchanged [Wol89b]. Using this technology, we can interchange the I and K loops directly, giving the IKJ form:

```
5: for i = 1,n do
3:  for k = 1,i-1 do
6:   a(i,k) = a(i,k)/a(k,k)
7:   for j = k+1,i do
8:    a(i,j) = a(i,j)-a(i,k)*a(j,k)
7:   endfor
3:  endfor
4:  a(i,i) = sqrt(a(i,i))
5: endfor
```

`Interchanging loops k and i`

Notice now the sqrt statement had to be moved from above the inner loop to below the inner loop; this was required by the modified data dependence relations after interchanging the loops. Now we would like to interchange the `K` and `J` loops; as it turns out, there are no data dependence conditions preventing this transformation. The problem here is that the `a(i,k)` assignment must be executed for `k=1:i-1`; if the two loops were interchanged, the new loop limits (according to normal loop interchange rules for triangular and trapezoidal loops) would be:

```
for j = 1,i do
 for k = 1,j-1 do
```

The index set for the `J` loop would be either `j=1:i` or `j=2:i`, neither one of these matching the required `1:i-1` limits. So we must first align these two loops to make them exactly triangular:

```
5: for i = 1,n do
3:  for k = 1,i-1 do
6:   a(i,k) = a(i,k)/a(k,k)
7:   for j = k+1-1,i-1 do
8:    a(i,j+1) = a(i,j+1)-a(i,k)*a(j+1,k)
7:   endfor
3:  endfor
4:  a(i,i) = sqrt(a(i,i))
5: endfor
```

`Bump loop j by -1`

This has no effect on the dependence relations in the loop, and now the two loops can be interchanged to produce the `IJK` form:

```
5: for i = 1,n do
7:  for j = 1,i-1 do
6:   a(i,j) = a(i,j)/a(j,j)
3:   for k = 1,j do
8:    a(i,j+1) = a(i,j+1)-a(i,k)*a(j+1,k)
3:   endfor
7:  endfor
4:  a(i,i) = sqrt(a(i,i))
5: endfor
```

Interchanging loops k and j

From this we can interchange the I and J loops, getting the JIK form:

```
7: for j = 1,n do
4:  a(j,j) = sqrt(a(j,j))
5:  for i = j+1,n do
6:   a(i,j) = a(i,j)/a(j,j)
3:   for k = 1,j do
8:    a(i,j+1) = a(i,j+1)-a(i,k)*a(j+1,k)
3:   endfor
5:  endfor
7: endfor
```

Interchanging loops i and j

Notice again how the sqrt statement had to be moved as part of the transformation. We now want to get to the JKI form; unfortunately, we cannot immediately interchange the I and K loops due to the same problem with a different index set for I and K and the a(i,j) assignment. This time, we cannot get away with a simple adjustment of the index set, so we must distribute the I loop:

```
7: for j = 1,n do
4:  a(j,j) = sqrt(a(j,j))
5:  for i = j+1,n do
6:   a(i,j) = a(i,j)/a(j,j)
5:  endfor
5:  for i = j+1,n do
3:   for k = 1,j do
8:    a(i,j+1) = a(i,j+1)-a(i,k)*a(j+1,k)
3:   endfor
5:  endfor
7: endfor
```

Distributing loop i

Now interchanging produces the JKI form:

```
7: for j = 1,n do
4:  a(j,j) = sqrt(a(j,j))
5:  for i = j+1,n do
6:   a(i,j) = a(i,j)/a(j,j)
5:  endfor
3:  for k = 1,j do
5:   for i = j+1,n do
8:    a(i,j+1) = a(i,j+1)-a(i,k)*a(j+1,k)
5:   endfor
3:  endfor
7: endfor
```

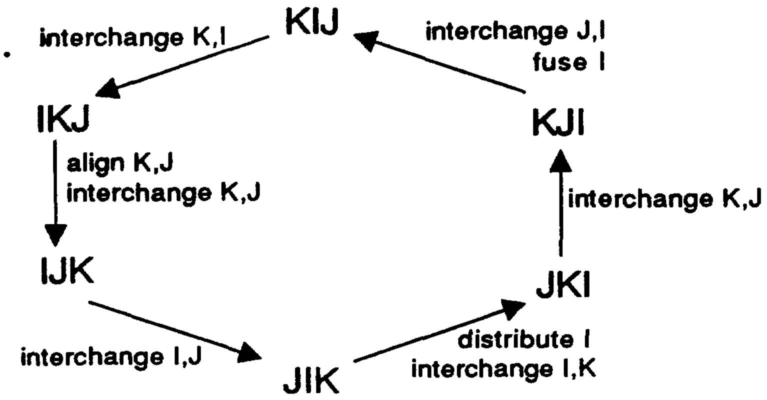
`Interchanging loops i and k`

To complete the example, we interchange the K and J loops to get the KJI form:

```
3: for k = 1,n do
4:  a(k,k) = sqrt(a(k,k))
5:  for i = k+1,n do
6:   a(i,k) = a(i,k)/a(k,k)
5:  endfor
7:  for j = k,n do
5:   for i = j+1,n do
8:    a(i,j+1) = a(i,j+1)-a(i,k)*a(j+1,k)
5:   endfor
7:  endfor
3: endfor
```

`Interchanging loops j and k`

From this final form we can interchange the I and J loops to get the KIJ form; the only difference from the final state and the initial state is that the I loop would be distributed. The restructuring sequence we followed is described by the diagram:



4. Design Goals

To further research into loop restructuring, we began the development of a restructuring research tool. The emphasis was on research, so we wanted to make it easy to add new transformations without worrying (yet) about every last detail and problem when dealing with a complex programming language which had evolved over 25 years, such as C or Fortran. We thus designed a tiny programming language comprising loops, assignments and conditionals with normal expressions involving arrays and scalars; examples of the language syntax were shown in the previous section. For this reason we call the tool Tiny.

In a compiler, a restructuring transformation has two preconditions: is the transformation legal and will the transformed program be better? The first condition is objective, and must be satisfied; most transformations require certain data dependence constraints to be satisfied, and occasionally there are other conditions having to deal with (for instance, the form of the loop limits). Some minor transformations are always legal, though they may modify the dependence graph. In order to test for these conditions, Tiny builds a data dependence graph when it parses the input program. Each variable reference in the program is linked to its dependence successors and dependence predecessors by a list of data dependence arcs. The data dependence arcs in Tiny are augmented with a direction vector and a distance vector [Wol89b]. For most of the transformations in Tiny, this abstract representation of the dependence relations is precise enough to handle the dependence constraints. However, some transformations need even more precise information, and we will show how Tiny handles these cases. Tiny now includes four different data dependence decision algorithms; we have been using Tiny to learn how to modify dependence decision algorithms to extract direction and distance vector information.

The second condition is more subjective, and certainly depends on the goal of this restructuring process (the target machine, for instance). The holy grail of our research is to find a process whereby a designer can describe (parametrically or graphically) his parallel computer system, a user can describe his algorithm, and the system will automatically find an optimal mapping of the algorithm to the computer system. Real computer systems, however, often have conflicting optimization goals (long vector operations to amortize vector startup, but short vector operations to avoid overflowing the cache memory). Just finding an appropriate set of parameters that describe the salient performance characteristics of the system is a difficult task; using these in an optimization process is yet another unsolved problem. For the time being, we have avoided this problem entirely by making the restructuring process entirely manual (menu-driven interactive).

Actually there are many issues dealing with restructured programs, whether they be automatically, manually or semi-automatically restructured. Performance prediction is useful to determine when the restructuring process is "done". Performance prediction is potentially feasible even when fully automatic restructuring is too difficult. Post-mortem performance and bottleneck analysis is also very useful. When debugging a restructured program, the user would rather be able to deal with his original program, or (at the very least) be able to see the correspondence between his original program and the restructured program being executed. This is where fully automatic restructuring compilers have an advantage, since a correct compiler will not change a correct program into an incorrect one; this allows (in principle) the original program to be debugged independently of the restructuring process. Any time the restructured program is exposed to the user, this problem creeps up. All of these issues are being ignored in Tiny for the time being. To reiterate, our goal is to concentrate on the restructuring technology itself, and see where that takes us.

Limitations. There are inherent limitations in the approach used in the development of Tiny. Since we are using "correctness-preserving" transformations, we are purposely disallowing any transformations which would change the results of the program. Our definition of "correctness-preserving" is that the data dependence relations are preserved. Note that there are perfectly valid transformations of a program that change the dependence relations, such as replacing a "sort" by a better one, that would be "correct" to an applications programmer but would not be allowed in this scheme. This is a basic limitation (some would call it a weakness) of the program transformation approach.

Another limitation is that we depend on the computed data dependence information to test for validity of the transformation. The data dependence information is only as precise as the data dependence algorithms, and they each have certain weaknesses.

The most serious weakness of Tiny is that it includes only a fixed set of program transformations. In the best of worlds, we would like to have a method that would allow a user to describe abstractly what sort of transformation he would like to perform to the program, and have the tool apply that transformation (first, of course, testing the data dependence graph for validity of the transformation). Perhaps the interface would be that the user would show a sample application of his transformation by means of "before" and "after" pictures; the tool would then have to decipher what the differences were and try to apply this change. This is an avenue for future research, if our basic program restructuring research bears any fruit.

We have found that the fixed set of transformations often is too limited for the applications we want. For instance, when first working through the 6 forms of the Cholesky decomposition (in the previous section), we knew that interchanging non-tightly nested loops would be important. We did not expect that we would need to align the loop limits to make them exactly square or triangular; we ended up adding yet another minor transformation just to handle that case. We also found that loop fusion could be useful, though we haven't implemented it yet. For the Gaussian elimination (as shown in the following section), we found that we couldn't generate all six forms without index set splitting, another minor transformation which has not been explored very well. Our worst fear is that every non-trivial example we attempt will require yet another minor transformation.

5. Restructuring Transformations

Here we describe the program restructuring transformations implemented in Tiny. Each transformation is implemented so that the dependence relations (and sometimes other conditions) are tested before program is transformed. Each transformation also modifies the dependence graph directly, rather than requiring Tiny to recompute the dependence relations. Most of these transformations are described in more detail in [Wol90].

Interchanging. The most basic restructuring transformation is loop interchanging, a well known optimization now regularly implemented in commercial compilers for parallel and vector computers. Its first applications were to exchange sequential inner loops with vectorizable outer loops; it has also proven useful for improving the performance of the memory system. The dependence test for loop interchanging (based on direction vectors) is well known, and the modified dependence distance and direction vectors are simple to derive (by interchanging elements corresponding to the loop interchange). If the inner loop limits depend on the outer loop index, interchanging will modify the loop limits; Tiny allows triangular as well as general trapezoidal loop limits [Wol86a]. Loop interchanging is its own inverse.

We have implemented two variations on loop interchanging in Tiny. The first is the ability to interchange non-tightly nested loops. We believe this is the first attempt to implement this potentially powerful transformation. The dependence test for interchanging non-tightly nested loops cannot be performed by simple inspection of the dependence direction or distance vectors. Each element in a dependence direction or distance vector corresponds to the sign or value of the difference of the iteration vector elements involved in a dependence; for instance, if there is a dependence from iteration (i_1, i_2) to iteration (j_1, j_2), the distance vector is the value of (j_1-i_1, j_2-i_2) while the distance vector represents the possible signs of these values. Normal loop interchanging is illegal if $i_1<j_1$ and $i_2>j_2$. When interchanging non-tightly nested loops, however, we can have dependence from some statement outside the inner loop at iteration (i_1) to iteration (j_1, j_2) of some statement within the inner loop. After interchanging the loops, the inner loop will be reindexed to (j_2, j_1). So, in addition to the normal dependence test for interchanging, we must look at the sign of (j_2-i_1), since this will be the new dependence distance vector element after interchanging. Since this information is not encapsulated in the distance or direction vectors, Tiny reconstructs the dependence equation and finds a new solution in these terms. A minor enhancement to one of the dependence algorithms in Tiny was necessary to handle this case. Interchanging of non-tightly nested loops can also create infeasible dependence cycles, since a loop carried dependence can change to a loop-independent dependence (and vice versa). Tiny uses the same algorithm as used in loop distribution to insure that there are no infeasible cycles and the proper placement of statements after interchanging.

The second variation is a generalized loop interchanging called *loop circulation* [Ban90]. Loop circulation corresponds to interchanging one loop outwards (or inwards) multiple times (over tightly nested loops). One advantage of loop circulation is that it requires only a single dependence test (so is a little faster). Also we believe that loop circulation will prove to be a more suitable candidate as an elementary transformation for automatically directed loop restructuring. The problem with automatic restructuring is to avoid exhaustive search; exhaustive search may be required if the tool cannot predict whether a transformation will always be good. With simple pairwise loop interchanging, a tool may not be able to tell whether interchanging two deeply nested loops will produce a better program in the end. With loop circulation, the tool can attempt to move the innermost loop all the way outwards in a single step, without visiting a lot of intermediate steps that may not in themselves be improvements in the performance.

Skewing. Loop skewing is a simple linear reindexing transformation; skewing is *always* legal (no dependence test is required), although it does modify the dependence relations. Simple skewing of an inner loop with respect to an outer loop modifies the dependence distance vector elements corresponding to the inner loop by adding the distance element corresponding to the outer loop. In particular, skewing loops may allow loop interchanging (by changing a `(1,-1)` dependence distance vector to `(1,0)`) or may allow parallelization after interchanging (by changing a `(1,0)` dependence distance vector to `(1,1)`). Loop skewing can be performed with any integer factor, and skewing by a negative factor is the inverse of positive skewing.

Reversal. Loop reversal (or negation) is simply running a loop backward. Reversal is legal if the loop *carries* no dependence relations. Reversal negates the dependence distance and direction vector elements for that loop, and is its own inverse.

Distribution. Loop distribution is useful when attempting to interchange non-tightly nested loops, or to break a loop with a forward loop-carried dependence relation into two separate parallel loops [BCK79]. Loop distribution partitions the statements of the loop into strongly-connected regions (treating inner loops like a single statement) based on the dependence graph [Tar72]. Distribution has little affect on the dependence graph (the distance and direction vectors for dependence relations between statements that get distributed into separate loops get shortened a little); its inverse transformation, loop fusion, is not yet implemented in Tiny.

Parallelization. Simple parallelization corresponds to finding loops with no loop carried dependence relations and marking them parallel [ACK87]. It is not a restructuring transformation in the sense of the previous transformations, but simply a recognition of parallelism in the program. Tiny can also attempt parallelization of each loop in the program after each transformation.

Bumping. As mentioned before, we have implemented a minor transformation that aligns the loop limits of one loop with another, which we call *bumping*. This is used to enable interchanging of non-tightly nested loops when the loop limits differ by a constant; it may also be useful when attempting loop fusion (see below). We expect that Tiny could perform bumping automatically when needed, but we want more experience before making the tests yet more complex.

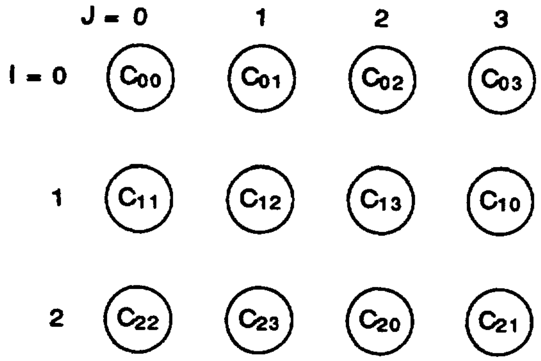
The following transformations are planned for implementation in the near future:

Fusion. Loop fusion is the inverse of loop distribution. In addition to requiring the loop limits to be identical (or the ability to align the loop limits), fusion requires another dependence test that is not simply an inspection of the direction vectors or distance vectors. Two loops can be fused only if all dependence relations from the upper loop to the lower loop *would not change sense* after fusion (would not change into loop-carried dependence relations the other way). The way to test for this is to compute a direction (or distance) vector element for the to-be-fused loop position, and test for a (>) direction (or negative distance) [Wol89b].

Rotation. Loop rotation is useful for mapping loop algorithms onto distributed memory systems [Wol89c]. Loop rotation corresponds to skewing the loop around a torus; given a loop such as:

```
for i = 0 to N-1
 for j = 0 to M-1
  computation(i,j)
 endfor
endfor
```

the picture below shows a rotated iteration space where C_{ij} corresponds to *computation*(i,j), and N=3, M=4.



The transformation changes the program to look like:

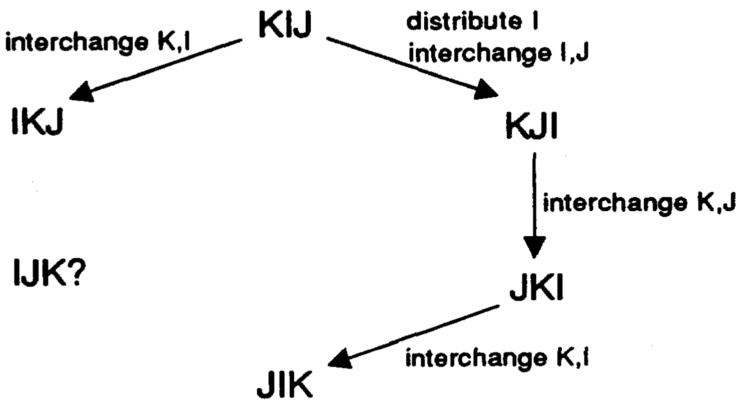
```
for i = 0 to N-1
 for j = 0 to M-1
  computation(i, (j-i) mod M)
 endfor
endfor
```

The dependence test for loop rotation requires recognition and representation of loop-carried dependence relations that correspond to a reduction operation (such as summation of a vector). This semantic information has never before been proposed in a data dependence abstraction.

Splitting. We also see the need for index set splitting; this has been proposed in the past as a way of exposing more parallelism in a loop, where the dependence relations change sense half-way through the execution of the loop [Ban79]. We see the need for splitting the index set to enable other transformations. For instance, if we use Tiny to produce the 6 versions of Gaussian Elimination (without pivoting), it turns out we get stuck in one place. Suppose the initial version of Gaussian Elimination is the KIJ form:

```
3: for k = 1,n do
4:  for i = k+1,n do
5:   a(i,k) = a(i,k)/a(k,k)
6:   for j = k+1,n do
7:    a(i,j) = a(i,j)-a(k,j)*a(i,k)
6:   endfor
4:  endfor
3: endfor
```

Using the same type of transformations as with the Cholesky decomposition, we can generate 4 of the other 5 orderings, using the transformation tree:



But when we try to go from the IKJ form to the IKJ form, we are stuck.

```
4: for i = 1+1,n do
3:  for k = 1,min(n,i-1) do
5:   a(i,k) = a(i,k)/a(k,k)
6:   for j = k+1,n do
7:    a(i,j) = a(i,j)-a(k,j)*a(i,k)
6:   endfor
3:  endfor
4: endfor
```

Interchanging loops k and i

The K and J loops cannot be interchanged immediately (due to the same index set problem as in the Cholesky decomposition), and the K loop cannot be distributed. We should be able to simplify the loop limits for K and split the J loop into two loops:

```
for i = 1,n do
 for k = 1,i-1 do
  a(i,k) = a(i,k)/a(k,k)
  for j = k+1,i-1 do
   a(i,j) = a(i,j)-a(k,j)*a(i,k)
  endfor
  for j = i,n do
   a(i,j) = a(i,j)-a(k,j)*a(i,k)
  endfor
 endfor
endfor
```

From here we can distribute the K loop and interchange each K with its inner J loop:

```
for i = 1,n do
 for j = 1,i-1 do
  for k = 1,j-1 do
   a(i,j) = a(i,j)-a(k,j)*a(i,k)
  endfor
  a(i,j) = a(i,j)/a(j,j)
 endfor
 for j = i,n do
  for k = 1,i-1 do
   a(i,j) = a(i,j)-a(k,j)*a(i,k)
  endfor
 endfor
endfor
```

This example shows that the loop restructuring method is general enough to handle this example, but our tool needs more elementary transformations in order to produce satisfactory results in nontrivial examples. We don't see the need for implementing the inverse to index set splitting (though we could be surprised).

Others. We have reason to believe that other interesting transformations will arise and be worth consideration. In almost every case, the transformation is one that is performed manually in order to optimize performance for some architecture. We formalize the transformation, discover or invent dependence conditions to allow the transformation, and implement it.

6. Summary

The Tiny program is a tool used to support research into program restructuring. Its interactive menu-driven interface allows a user to apply a sequence of loop restructuring transformations to a nested loop algorithm. Some new and advanced restructuring transformations are being implemented in Tiny, and more are planned for the near future. Currently this research is an end in itself, and details such as code generation are left as an open question.

The first program restructuring transformation is loop interchanging. When first proposed as an automatic optimization, it was considered expensive and unnecessary in most cases; now it is regularly included in commercial compilers. In fact it is not an expensive transformation, and its effects on the execution of a program reach beyond simple enhancement of the parallelism in a few programs. For instance, it can dramatically reduce the bandwidth requirements on the memory system, and can change memory strides and vector lengths. We believe that our research may identify other transformations that should (or should not) be included in programming environments and compilers for advanced architecture computer systems.

References

[AKL81] W. A. Abu-Sufah, D. J. Kuck and D. H. Lawrie, On the Performance Enhancement of Paging Systems Through Program Analysis and Transformations, *IEEE Trans. on Computers C-30,* 5 (May 1981), 341-356.

[AlK82] J. R. Allen and K. Kennedy, PFC: A Program to Convert Fortran to Parallel Form, in *Supercomputers: Design and Applications,* K. Hwang (ed.), IEEE Computer Society Press, Silver Spring, MD, 1982, 186-203.

[AlK84] J. R. Allen and K. Kennedy, Automatic Loop Interchange, in *Proc. of the SIGPLAN 84 Symposium on Compiler Construction,* New York, June 1984, 233-246.

[ACK87] R. Allen, D. Callahan and K. Kennedy, Automatic Decomposition of Scientific Programs for Parallel Execution, in *Conf. Record of the 14th Annual ACM Symp. on Principles of Programming Languages,* ACM Press, New York, 1987, 63-76.

[AlK87] J. R. Allen and K. Kennedy, Automatic Translation of Fortran Programs to Vector Form, *ACM Transactions on Programming Languages and Systems 9,* 4 (October 1987), 491-542.

[BCK79] U. Banerjee, S. Chen, D. J. Kuck and R. A. Towle, Time and Parallel Processor Bounds for Fortran-Like Loops, *IEEE Trans. on Computers C-28,* 9 (September 1979), 660-670.

[Ban79] U. Banerjee, *Speedup of Ordinary Programs,* PhD Thesis, Univ. of Illinois, October 1979. (UMI 80-08967).

[Ban90] U. Banerjee, A Theory of Loop Permutations, in *Languages and Compilers for Parallel Computing,* D. Gelernter, A. Nicolau and D. Padua (ed.), Pitman, London, 1990, 54-74.

[GJG88] D. Gannon, W. Jalby and K. Gallivan, Strategies for Cache and Local Memory Management by Global Program Transformation, *J. Parallel and Distributed Computing 5,* 5 (October 1988), 587-616, Academic Press.

[GoT88] M. B. Gokhale and T. C. Torgerson, The Symbolic Hyperplane Transformation for Recursively Defined Arrays, in *Proc. of Supercomputing 88,* IEEE Computer Society Press, Los Angeles, 1988, 207-214. Orlando, FL, November 14-18, 1988.

[Knu73] D. Knuth, *The Art of Computer Programming: Fundamental Algorithms,* Addison-Wesley, Reading, MA, 1973.

[Lam75] L. Lamport, The Hyperplane Method for an Array Computer, in *Parallel Processing: Proc. of the Sagamore Computer Conference,* vol. 24, T. Feng (ed.), Springer-Verlag, Berlin, 1975, 113-131.

[LeK90] P. Lee and Z. M. Kedem, Mapping Nested Loop Algorithms into Multidimensional Systolic Arrays, *IEEE Trans. on Parallel and Distributed Systems 1,* 1 (January 1990), 64-76.

[Lov77] D. Loveman, Program Improvement by Source-to-Source Transformation, *J. of the ACM 20,* 1 (January 1977), 121-145.

[Sch72] P. B. Schneck, Automatic Recognition of Vector and Parallel Operations in a Higher Level Language, *SIGPLAN Notices 7,* 11 (November 1972), 45-52.

[Tar72] R. Tarjan, Depth-First Search and Linear Graph Algorithms, in *SIAM J. Comput.*, vol. 1, June. 1972, 146-160.

[Wed75] D. Wedel, Fortran for the Texas Instruments ASC System, *SIGPLAN Notices 10,* 3 (March 1975), 119-132.

[Wol78] M. Wolfe, Techniques for Improving the Inherent Parallelism in Programs, UIUCDCS-R-78-929, Univ. of Illinois, July 1978.

[Wol82] M. Wolfe, *Optimizing Supercompilers for Supercomputers,* Ph.D. Thesis, Univ. of Illinois UIUCDCS-82-1105, Urbana, IL, October 1982. (UMI 83-03027).

[Wol86a] M. Wolfe, Advanced Loop Interchanging, in *Proc. of the 1986 Intl Conf. on Parallel Processing,* K. Hwang, S. M. Jacobs and E. E. Swartzlander (ed.), St. Charles, IL, August 19-22, 1986, 536-543.

[Wol86b] M. Wolfe, Loop Skewing: The Wavefront Method Revisited, *Intl J. Parallel Programming 15,* 4 (August 1986), 279-294.

[Wol89a] M. Wolfe, Iteration Space Tiling for Memory Hierarchies, in *Parallel Processing for Scientific Computing,* G. Rodrigue (ed.), Society for Industrial and Applied Mathematics, Philadelphia PA, 1989, 357-361.

[Wol89b] M. Wolfe, *Optimizing Supercompilers for Supercomputers,* Pitman Publishing, London, 1989.

[Wol89c] M. Wolfe, Loop Rotation, CS/E 89-004, Oregon Graduate Institute, Beaverton OR, May 1989.

[Wol90] M. Wolfe, Data Dependence and Program Restructuring, *Journal of Supercomputing,* 1990.