Experiences with Data Dependence and Loop Restructuring in the Tiny 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

Experiences with Data Dependence and Loop Restructuring in the Tiny 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. The most commonly used loop restructuring transformation is loop interchanging, which can improve parallelism or memory hierarchy performance. 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. Our research program is exploring 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 our experiences with the second question above; in particular, we will describe somewhat disappointing results with respect to current data dependence abstractions and more advanced loop restructuring transformations.

Experiences with Data Dependence and Loop Restructuring in the Tiny 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 transformations below; in each case we give a representative citation (instead of an exhaustive list); a summary of many of these transformations is available in the monograph [Wol89].

	transformation	enhances	reference
•	vectorization	parallelism	[Sch72]
•	parallelization	parallelism	[ACK87]
•	strip mining	vectorization	[Lov77]
•	distribution	vectorization	[AlK87]
•	interchanging	parallelism	[AlK84]
•	interchanging	memory	[GJG88]
	skewing	parallelism	[Wol86]
	tiling	memory	[AKL81]
	fusion	overhead	[AKL81]
	reversal	interchanging	[Wol89]
	alignment	parallelism	[ACK87]
	splitting	parallelism	[Ban79]
	rotation	communication	[Wol90a]

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 (loop reversal, or running a loop backwards, was implemented in the TI ASC NX Fortran compiler, where it was called "loop inversion" [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. Even loop interchanging, which is included in nearly all current commercial vectorizing and parallelizing compilers and generally considered essential, was initially shunned due to its perceived cost and limited benefits.

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 general 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. Loop restructuring (in some sense) subsumes this work since linear index set transformations can always be implemented as loop restructuring transformations (some combination of loop interchanging, skewing and reversal). The converse is not true; there are loop restructuring transformations that correspond to nonlinear transformations of the index set, and so are outside the scope of that work.

Many of the interesting transformations developed for high performance computers can be formulated as a combination of multiple elementary transformations. For instance, the *wavefront method* or *hyperplane method* was proposed to allow parallel execution of a reindexed loop [Lam75] in cases where neither the inner nor outer loop could be executed in parallel. Previous work has shown that the wavefront method can be viewed as a combination of loop skewing and interchanging [Wol86]. Another example is the independent work on *loop blocking* or *tiling* which found that proper partitioning of the iteration space (index set) of a nested loop into blocks or tiles would greatly enhance the locality of reference for data access in the inner loop levels [AKL81, GJG88]. It was quickly realized that from a compiler point of view, tiling could be viewed as a combination of strip mining (or sectioning) [Lov77] and interchanging, once again combining two elementary loop transformations to achieve a powerful result.

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 loop restructuring has already found its way into daily use in the commercial world; however little experimentation has been done with other potentially interesting advanced restructuring transformations. In order to allow this 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, called Tiny.

The initial goal of the research is to develop a tool that allows a user to interactively restructure the loops in a program. The eventual goal will to be have performance metrics by which the tool will advise the user as to the predicted performance of the restructured loop vis-a-vis the original loop, or even to attempt a series of transformations automatically; while we feel these metrics are crucial to the eventual success of any restructuring process, we are initially concentrating on developing a number of loop restructuring transformations. In every case, a restructuring transformation is discovered because some human expert found it necessary to perform that transformation manually in order to improve

performance, or to enable some subsequent transformation. The automation of a transformation comprises the development of the rules to test when the transformation is legal and the mechanics of implementation.

In the Tiny tool, the legality rules are usually tests of the data dependence relations in the program [PaW86], and the mechanics of implementation include generating modified dependence relations [Wol90]. Tiny uses an abstract syntax tree data structure, where each node in the tree can have an associated list of data dependence successors; each dependence relation in the list is annotated with direction and distance vector information. These dependence abstractions were originally developed with particular transformations in mind, and have proven to be useful for many common applications. However, when looking at more advanced transformations, we have found that these abstractions are not sufficient; in fact, it is not clear if there is a single general and efficient abstraction to support all the transformations we envision.

2. Direction and Distance Vector Abstractions

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 following 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]. 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 (left-most) 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. Thus, for a doubly nested loop, a dependence with a direction vector of `(<,>)` will prevent interchanging those loops. In a triply nested loop, a dependence with a direction vector of `(=,<,>)` will prevent interchanging of the inner two loops, while a dependence with a direction vector of `(<,<,>)` will not prevent interchanging (since it is carried by the outer loop). For loop interchanging, the direction vector seemed like a natural dependence abstraction; if the direction vector is implemented as a bit-vector, with three bits per loop, then the test for loop interchanging can be implemented as an efficient bit mask test. In contrast, using the dependence distance information to test for interchanging would require more storage (one word for each loop) and a (slightly) more complicated test.

Loop skewing is designed to be used with interchanging to implement the wavefront method [Wol86]. By itself, skewing is always legal; however since the purpose of skewing is to enable interchanging to find more parallelism, it must be done with the dependence relations in mind. The canonical example for skewing is the following loop:

```
for i = 2 to n-1 do
  for j = 2 to m-1 do
    a(i,j) = 0.25*(a(i,j-1)+a(i-1,j)+a(i,j+1)+a(i+1,j))
  endfor
endfor
```

In this loop there are four dependence relations with two distinct dependence distance vectors: `(0,1)` and `(1,0)`. Since each loop carries a dependence relation, neither loop can be executed in parallel. Skewing the loop generates the program:

```
for i = 2 to n-1 do
  for j = i+2 to i+m-1 do
    a(i,j) = 0.25*(a(i,j-i-1)+a(i-1,j-i)+a(i,j-i+1)+a(i+1,j-i))
  endfor
endfor
```

by adding `i` to the limits of `j` (and correcting within the loop); the dependence relations are modified by adding the distance for the `i` loop to the distance for the `j` loop, producing the dependence distance vectors: `(0,1)` and `(1,1)`. Now, after loop interchanging:

```
for j = 2+2 to n-1+m-1 do
  for i = max(2,j-m+1) to min(n-1,j-2) do
    a(i,j) = 0.25*(a(i,j-i-1)+a(i-1,j-i)+a(i,j-i+1)+a(i+1,j-i))
  endfor
endfor
```

the dependence distance vectors are `(1,0)` and `(1,1)`; in both cases, the dependence is carried by the outer loop, so the inner loop can be executed in parallel.

In this example, the dependence distances were simple and skewing by a factor of one (adding `1×i` to the `j` loop limits) was sufficient to get to the desired result; in fact, the information in the direction vector would have been sufficient to decide that skewing would be satisfactory here. In the general case, however, dependence distance information is necessary in order to decide the skewing factor. This is a transformation that needs distance information in the general case.

3. Why Use Abstractions

One may ask "Why use dependence abstractions at all?" Consider the process of constructing a data dependence graph. For every pair of references (where one is an output reference) to the same variable (or potentially aliased variables) the compiler or programming tool must find whether there is a data dependence relation between them. Standard dataflow analysis will suffice for most scalar variables. For array references, current compiler techniques derive a set of dependence equations from the subscript functions, then use a decision algorithm to discover (a) whether there is any solution to the dependence equation at all (if not, the two references are completely independent), and (b) how to characterize the solutions to the dependence equation (with distance or direction vectors, for instance). Several decision algorithms are in common use today (such as Banerjee's Inequalities [Ban76,BCK79] and the GCD test [AlK87,Ban76,Coh73]) and recently there has been significant work in the development of new decision algorithms [Ban88,KKP90,LY90,Wal88,WoT90]. A single compiler may implement two or more of these tests, plus additional special case tests for important cases not handled by these general algorithms. By using some sort of dependence abstraction such as a direction vector, each decision algorithm can be formulated to find the potential dependence directions; each transformation can then be formulated to inspect the direction vector when testing for legality. This achieves a kind of orthagonality between the decision algorithms and the transformations, allowing new transformations or

new decision algorithms to be added without having to add or change other parts of the compiler.

An alternative would be to derive a specific dependence test for each transformation, and apply that test whenever the transformation is required; this is suggested, for instance, by the development of a specific decision algorithm for loop interchanging, as in Allen and Kennedy's paper [AlK87]. One problem with this approach is the increase in the cost of the development of the compiler as a whole. Each of the decision algorithms implemented in a compiler would then have to be specially formulated to see how it could be customized for each transformation. This might be satisfactory if only a single decision algorithm is used; our experience is that where one decision algorithm is satisfactory (in precision and speed) at certain times (perhaps when testing single-subscript, single-loop cases), it is inappropriate for other cases (i.e., testing multiple-subscript or multiple-loop), whereas a decision algorithm appropriate for the latter case would be too expensive for the former.

In comparing the two alternatives, we find several trade-offs. Using a dependence abstraction can sometimes lose information; for instance, using direction vectors exclusively loses distance information that could be very useful for some transformations. Avoiding any particular abstraction allows the development of specialized tests that might be more precise for a particular transformation (we will see an example of this in the next section). Using a dependence abstraction may require computing information that will never be used during the restructuring process; for instance, the compiler may find the full precise direction vector for each dependence relation, but may (for other reasons) never even attempt loop interchanging or any transformation that needs that information. On the other hand, using transformation-specific tests may require computing information redundantly. Interchanging an inner loop outward twice will require testing for $(=,<,>)$ directions for the first step, then for $(<,>,*)$ directions for the second step; if there is no dependence with a $(>)$ direction in that inner loop, then in fact only a single test is really needed for both steps. Using a dependence abstraction may require modifying the dependence graph after performing a transformation, or even recomputing a new dependence graph in the worst case. Using transformation-specific tests is essentially equivalent to computing a new dependence graph after each transformation, since no dependence information is carried from step to step.

Tiny uses the first approach; it eagerly computes a full dependence graph saving both direction vectors and distance vectors for each dependence relation between array references. In the cases where the dependence distance for a particular loop cannot be determined precisely, or where the distance is not a constant, Tiny stores the value zero in the distance; this is a flag that the distance is unknown and the direction vector should be inspected. If the actual distance is zero, the direction vector element for that loop will be $(=)$, so no information is lost. In the cases where the direction vector element for a particular loop cannot be determined to be either $(<)$, $(=)$ or $(>)$, Tiny stores imprecise directions, such as $(\leq)$, $(\geq)$ or $(*)$, where the latter means no information is known about the dependence direction. Determination of which loop "carries" a dependence can be done directly from the direction

vector [AlK87], and many of the transformations need only the information available in the direction and distance vectors.

4. Where the Abstractions Fail

Given that the data dependence decision algorithms in Tiny compute direction and distance vector information, we formulated the dependence tests for each transformation to use that information. Unfortunately, we found several cases where these dependence abstractions fail, or where they lack certain characteristics. The table below gives the dependence abstraction to support each elementary transformation considered, and where additional information is necessary or useful:

transformation	necessary information	additional information
vectorization	carrier loop	reductions
parallelization	carrier loop	reductions
strip mining	none	thresholds
distribution	carrier loop	
interchanging	direction vector	reductions
skewing	distance vector	
reversal	carrier loop	reductions
fusion	extended direction	
splitting		crossing threshold
non-tight interchanging	cross-direction	

Extended Directions: As has been noted in the literature [WoB87, Wol89], the data dependence test for loop fusion requires dependence direction or distance information about relative iterations in adjacent, but distinct, loops. For instance, in the loop:

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

the dependence relation from the first assignment to the second will have only a single direction (or distance) vector entry, for the single loop surrounding both statements (in this case, the direction will be (=) meaning that the dependence goes from iteration i' to iteration i'' where $i'=i''$). The test for

the legality of loop fusion must find a direction relating the j and k loops; in this case, the dependence goes from iteration j' to iteration k'' where $j'=k''-1$, or $j'>k''$. Since the extended direction is (>), loop fusion is illegal in this case. The direction vector abstraction used in Tiny does not compute this information, so implementing loop fusion requires a special decision algorithm. The problem with attempting to compute this information ahead of time is that with the possibility of loop interchanging, statement and loop reordering, and multiple fusion steps, it is not clear what loops are candidates for fusion and what are not; potentially, Tiny could attempt fusing every loop with every other, making eager dependence computation very expensive. Since fusion has not been a high priority, we did not even attempt to speed up the dependence test for fusion.

Crossing Thresholds: Index set splitting is usually used within Tiny to enable other transformations, such as interchanging non-tightly nested loops [Wol90b], and so dependence information is not needed. Sometimes index set splitting is used to break a dependence cycle. In a loop such as:

```
for i = 1 to 100 do
 a(i) = F(a(101-i))
endfor
```

there is a dependence cycle, but every dependence relation "crosses" the 51st iteration. Splitting the index set into the two pieces:

```
for i = 1 to 50 do
 a(i) = F(a(101-i))
endfor
for i = 51 to 100 do
 a(i) = F(a(101-i))
endfor
```

allows parallel execution of the iterations of both loops, though not parallel execution of the two loops with each other. The point at which to split the index set is called the *crossing threshold* [AlK87], and its computation is shown in Banerjee's thesis [Ban79]. Since most of the index set splitting is not done to break these crossing dependences, Tiny does not compute crossing thresholds.

Reduction Directions: The dependence test for vectorizing an innermost loop is that there must be no dependence cycle (after ignoring dependence relations carried by outer loops). The direction or distance vector information is necessary only to determine which dependence relations are carried by outer loops, and so can be ignored while vectorizing the innermost loop. For almost all applications, however, reduction operations can be handled differently. For instance, in the loop:

```
for i = 1 to n do
 a(i) = b(i) + c(i)
 s = s + a(i)
endfor
```

the second assignment exhibits a self-cycle of dependence, with dependence distance one or direction vector (<). From inspection of the dependence relations alone, this would appear to prevent full vectorization of the loop; looking at the assignment itself, it is obvious that the statement is a summation.

Most of the time the order of the summation is not important, and the compiler is allowed to reorder the summation in order to achieve better performance. The information necessary to determine whether vectorization of this loop is not available in the dependence graph, however.

Other transformations also should be able to treat reductions as a special case. Loop reversal, for instance, is legal if the loop carries no dependence relations; the naive direction or distance vector abstraction for a reduction, such as the loop shown above, would prevent reversing the loop. Again, if the order of the reduction does not matter, then reversing a reduction should be allowed. Similarly, parallelization is legal if the loop carries no dependence relations; some parallel computers have efficient methods to compute reduction operations, so such a loop should be considered parallelizable, even though (strictly speaking) the loop carries the reduction dependence. A more interesting case arises when a compiler is presented with a two-dimensional reduction, such as:

```
for i = 1 to N do
 for j = 1 to M do
  s = s + b(i,j)
 endfor
endfor
```

Strictly speaking, the dependence relations for this loop have direction $(=,<)$, for the dependence from iteration of the (i,j) loop to $(i,j+1)$, and direction $(<,>)$, for the dependence from iteration (i,M) to $(i+1,1)$. This latter dependence direction prevents interchanging the two loops. Again, if the order of the reduction does not matter, then interchanging should be allowed, and we would like this to be reflected in the dependence abstraction.

The problem is that the dependence abstraction lacks any semantic information about the program; only the flow of data or use of storage in the program is reflected. In Tiny, we added a new type of data dependence direction, called the *reduction direction*. With this abstraction, each dependence relation is annotated with a direction vector, where the direction vector has one element for each loop surrounding both references linked by the relation. Each direction vector element is a member of $\{<,=,>,\leq,\geq,*,R\}$. A reduction operation, such as the two-dimensional summation above, would be represented with a dependence relation with direction vector (R,R). With this information, the dependence tests for vectorization, parallelization, reversal, interchanging and other transformations can separate loop-carried dependence relations that correspond to reductions, (where reordering is allowed) and those that correspond to other dependence relations (where the order must be observed). In practice, a reduction direction corresponds to a dependence distance of one, and it seems to only make sense to have a reduction direction when all direction vector elements are either (R) or $(=)$ (i.e., no $(<,R)$ directions).

Non-Tightly Nested Loops: One of the early goals of Tiny was to be able to generate all six forms of the Cholesky decomposition algorithm (LL^T decomposition of a positive definite symmetric matrix) given one of the forms; the six forms are named after the order of the three loops around the

innermost statement, so the program below is the KIJ form:

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

In order to generate the IKJ form of the program, the non-tightly nested k and i loops must be directly interchanged; the normal process of interchanging non-tightly nested loops (distributing the outer loop, then interchanging) is prevented by a dependence cycle which disallows the loop distribution. The generated form of the program is:

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

The dependence test for interchanging (normal) tightly nested loops involves inspecting the direction or distance vector, which contains information about the relative values of indices $(k''-k', i''-i')$ for dependence from iteration (k', i') to iteration (k'', i''); in contrast, the dependence test for interchanging non-tightly nested loops involves inspecting the relative values of $(k''-k', i''-k')$ for dependence from the non-tightly nested statement at iteration (k') to a statement within the inner loop at iteration (k'', i''). Of particular importance is the fact that the *relative* iteration numbers of *different* index variables must be known to test for the legality of the transformation. This differs from the extended direction vectors needed for loop fusion; there the different loop indices being compared were for adjacent loops at the same nest level, while here the different loop indices are for nested loops. This information is not contained in the direction vector or distance vector abstraction. As with loop fusion, it is hard to know beforehand which non-tightly nested loops might be tempting to interchange, and so which cross-index directions should be precomputed. Thus, Tiny has a special decision algorithm just to compute this dependence information on demand when interchanging non-tightly nested loops is requested.

5. Summary

The Tiny program is a tool used to support research into program restructuring. We believe that our research will identify other transformations that should (or should not) be included in programming environments and compilers for advanced architecture computer systems. One aspect of this research is the discovery of methods to calculate and represent the data dependence information necessary to test for the legality of different restructuring transformations.

The examples in the previous section show some of the weaknesses of the data dependence abstractions used in Tiny; these same abstractions are also widely used in other research and commercial products. In one case, we propose extending direction vectors to include some semantic information about reduction operations; this seems sufficiently general to warrant implementation. PFC, the research parallelizing compiler developed at Rice University, finds crossing thresholds and implements the transformations necessary to take advantage of them. It is not clear whether they occur frequently enough to warrant inclusion in the general data dependence abstraction. In the other two cases, we have taken the approach that no current data dependence abstraction is both sufficiently powerful to expose the necessary information and relatively efficient (in space, computation time and access time) to warrant implementation. Tiny uses the approach that this information is computed as necessary ("on demand") when the appropriate transformation is requested. Note that in all such cases, the original data dependence graph serves as a starting point; that is, the new decision algorithm need only be applied where the more general data dependence graph already shows a dependence relation.

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.

[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.

[Ban76] U. Banerjee, Data Dependence in Ordinary Programs, UIUCDCS-R-76-837, Univ. Illinois, Dept. Computer Science, Urbana, IL, November 1976.

[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).

[Ban88] U. Banerjee, *Dependence Analysis for Supercomputing*, Kluwer Academic Publishers, Norwell, MA, 1988.

[Coh73] W. L. Cohagan, Vector Optimization for the ASC, in *Proc. of the Seventh Annual Princeton Conf. on Information Sciences and Systems*, Princeton University, Princeton, NJ, 1973, 169-174.

[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.

[KKP90] X. Kong, D. Klappholz and K. Psarris, The I Test: A New Test for Subscript Data Dependence, in *Proceedings of the 1990 International Conference on Parallel Processing*, August 1990.

[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.

[LY90] Z. Li, P. C. Yew and C. Q. Zhu, An Efficient Data Dependence Analysis for Parallelizing Compilers, *IEEE Trans. on Parallel and Distributed Systems 1*, 1 (January 1990), 26-34.

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

[PaW86] D. A. Padua and M. Wolfe, Advanced Compiler Optimizations for Supercomputers, *Comm. of the ACM 29*, 12 (December 1986), 1184-1201.

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

[Wal88] D. R. Wallace, Dependence of Multi-Dimensional Array References, in *Proc. of the 1988 International Conf. on Supercomputing*, ACM, 1988, 418-428. St. Malo, France, July 4-8, 1988.

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

[Wol86] M. Wolfe, Loop Skewing: The Wavefront Method Revisited, *Intl J. Parallel Programming 15*, 4 (August 1986), 279-294.

[WoB87] M. Wolfe and U. Banerjee, Data Dependence and Its Application to Parallel Processing, *Intl Journal of Parallel Programming 16*, 2 (April 1987), 137-178.

[Wol89] M. Wolfe, *Optimizing Supercompilers for Supercomputers*, Pitman Publishing, London, 1989.

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

[WoT90] M. Wolfe and C. Tseng, The Power Test for Data Dependence, *J. Supercomputing*, to appear, 1990.

[Wol90a] M. Wolfe, Loop Rotation, in *Languages and Compilers for Parallel Computing*, D. Gelernter, A. Nicolau and D. Padua (ed.), Pitman, London, 1990, 531-553.

[Wol90b] M. Wolfe, A Loop Restructuring Research Tool, CS/E 90-014, Oregon Graduate Institute, Beaverton OR, 1990.