Memory-Hierarchy Management

Steve Carr

CRPC-TR92222-S
September 1992

Center for Research on Parallel Computation
Rice University
6100 South Main Street
CRPC - MS 41
Houston, TX 77005

RICE UNIVERSITY

Memory-Hierarchy Management

by

Steve Carr

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Houston, Texas

February, 1993

Memory-Hierarchy Management

Steve Carr

Abstract

The trend in high-performance microprocessor design is toward increasing computational power on the chip. Microprocessors can now process dramatically more data per machine cycle than previous models. Unfortunately, memory speeds have not kept pace. The result is an imbalance between computation speed and memory speed. This imbalance is leading machine designers to use more complicated memory hierarchies. In turn, programmers are explicitly restructuring codes to perform well on particular memory systems, leading to machine-specific programs.

It is our belief that machine-specific programming is a step in the wrong direction. Compilers, not programmers, should handle machine-specific implementation details. To this end, this thesis develops and experiments with compiler algorithms that manage the memory hierarchy of a machine for floating-point intensive numerical codes. Specifically, we address the following issues:

Scalar replacement. Lack of information concerning the flow of array values in standard data-flow analysis prevents the capturing of array reuse in registers. We develop and experiment with a technique to perform scalar replacement in the presence of conditional-control flow to expose array reuse to standard data-flow algorithms.

Unroll-and-jam. Many loops require more data per cycle than can be processed by the target machine. We present and experiment with an automatic technique to apply unroll-and-jam to such loops to reduce their memory requirements.

Loop Interchange. Cache locality in programs run on advanced microprocessors is critical to performance. We develop and experiment with a technique to order loops within a nest to attain good cache locality.

Blocking. Iteration-space blocking is a technique used to attain temporal locality within cache. Although it has been applied to "simple" kernels, there has been no investigation into its applicability over a range of algorithmic styles. We show how to apply blocking to loops with trapezoidal-, rhomboidal-, and triangular-shaped iteration spaces. In addition, we show how to overcome certain complex dependence patterns.

Experiments with the above techniques have shown that integer-factor speedups on a single chip are possible. These results reveal that many numerical algorithms can be expressed in a natural, machine-independent form while retaining good memory performance through the use of compiler optimizations.

Acknowledgments

I would like to thank the members of my committee for their input and support throughout the development of this thesis. In particular, I would like to thank Ken Kennedy for his guidance and support throughout my graduate career, especially through the numerous times when I was ready to quit. I would also like to give special thanks to Keith Cooper for being willing to listen when I had ideas that needed to be heard.

From the onset of graduate school, I found that many of my fellow students provided much-needed support and insight. Preston Briggs, Uli Kremer, Mary Hall and Ervan Darnell have given feedback on the ideas developed within this thesis. Those who began graduate school with me, René Rodríguez, Elmootazbellah Elnozahy and Rebecca Parsons, have been great friends and a constant source of encouragement. The members of the ParaScope research group provided the infrastructure for the implementation of the ideas developed within this thesis. Finally, special thanks goes to Ivy Jorgensen for proofreading this thesis.

Without the encouragement of my undergraduate professors at Michigan Tech, I would have never even considered graduate school. Dave Poplawski, Steve Seidel, and Karl Ottenstein provided that encouragement and I am indebted to them for it.

As with all endeavors, financial support is needed to allow one to eat. I have been supported by IBM Corporation, the National Science Foundation and Darpa at various points in my graduate career.

Finally, I wish to thank my family. My parents have supported me with love and encouragement that has been invaluable. Most of all, I thank my wife Becky who has been the best friend that I could ever have. Her love and encouragement has done more to make my life meaningful than she could ever imagine.

Contents

Illustrations

Chapter 1

Introduction

Over the past decade, microprocessor design strategies have focused on increasing the computational power available on a single chip. These advances in power have been achieved not only through reduced cycle times, but also via architectural changes such as multiple instruction issue and pipelined floating-point functional units. The resulting microprocessors can process dramatically more data per machine cycle than previous models. Unfortunately, the performance of memory has not kept pace. The result has been an increase in the number of cycles for a memory access—a latency of 10 to 20 machine cycles is now quite common—causing an imbalance between the rate at which computations can be performed and the rate at which operands can be delivered onto the chip.

To ameliorate these problems, machine designers have turned increasingly to complex memory hierarchies. For example, the Intel i860XR has an on-chip cache memory for 8K bytes of data and there are several systems that use two levels of cache with a MIPS R3000. Still, these systems perform poorly on scientific calculations that are memory intensive and are not structured to take advantage of the target machine's memory hierarchy.

This situation has led many programmers to restructure their codes by hand to improve performance in the memory hierarchy. We believe that this is a step in the wrong direction. The user should not be writing programs that target a particular machine; instead, the task of specializing a program to a target machine should fall to the compiler. If this trend continues, an increasing fraction of the human resources available for science and engineering will be spent on conversion of high-level language programs from one machine to another—an unacceptable eventuality.

There is a long history of the use of sophisticated compiler optimizations to achieve machine independence. The Fortran I compiler included enough optimizations to make it possible for scientists to abandon machine language programming. More recently, advanced vectorization technology has made it possible to write machine-independent vector programs in a sublanguage of Fortran 77. Is it possible to achieve the same success for memory-hierarchy management on scalar processors? More precisely, can we enhance compiler technology to make it possible to express an algorithm in a natural, machine-independent form while achieving memory-hierarchy performance good enough to obviate the need for hand optimization?

This thesis shows that compiler technology can make it possible for a program expressed in a natural form to achieve high performance, even on a complex memory hierarchy. Compiler algorithms to manage the memory hierarchy automatically are developed and shown through experimentation to be effective. By adapting compiler technology developed for parallel and vector architectures, we are able to restructure scientific codes to achieve good memory performance on scalar architectures. In many cases, our techniques have been extremely effective — capable of achieving integer-factor speedups over code generated by a good optimizing compiler of conventional design. This accomplishment represents a step forward in the encouragement of machine-independent programming.

This chapter introduces the models, transformations and previous work on which our compiler strategy is based. Section 1.1 presents the models we use to understand machine and program behavior. Section 1.2 presents the transformations that we will use to improve program performance. Section 1.3 presents the previous work related to memory-hierarchy management and, finally, Section 1.4 gives a brief overview of the thesis.

1.1 Background

In this section, we lay the foundation for the application of our reordering transformations that improve the memory performance of programs. First, we describe a measure of machine and loop performance used in the application of the transformations described in the next section. Second, we present a special form of dependence graph that can be used by our transformation system to model memory usage.

1.1.1 Performance Model

We direct our research toward architectures that are pipelined and allow asynchronous execution of memory accesses and floating-point operations (*e.g.*, Intel i860 or IBM RS/6000). We assume that the target machine has a typical optimizing compiler — one that performs scalar optimizations only. In particular, we assume that it performs strength reduction, allocates registers globally (via some coloring scheme) and schedules the arithmetic pipelines [CK77, CAC$^+$81, GM86]. This makes it possible for our transformation system to restructure the loop nests while leaving the details of optimizing the loop code to the compiler.

Given these assumptions for the target machine and compiler, we will describe the notion of balance defined by Callahan, et. al, to measure the performance of program loops in relation to memory [CCK88]. This model will serve as the force behind the application of our transformations described throughout this thesis.

Machine Balance

A computer is balanced when it can operate in a steady state manner with both memory accesses and floating-point operations being performed at peak speed. To quantify this relationship, we define β_M as the rate at which data can be fetched from memory, M_M, compared to the rate at which floating-point operations can be performed, F_M:

$$\beta_M = \frac{\text{max words/cycle} = M_M}{\text{max flops/cycle} = F_M}$$

The values of M_M and F_M represent peak performance where the size of a word is the same as the precision of the floating point operations. Every machine has at least one intrinsic β_M (there may be one for single-precision floating point and one for double precision). For example, on the IBM RS/6000 $\beta_M = 1$ and on the DEC Alpha $\beta_M = 3$.

Although this measure can be used to determine if a machine is balanced between computation and data accesses, it is not our purpose to use balance in this manner. Instead, our goal is to evaluate the performance of a loop on a particular machine based upon that machine's particular balance ratio. To do this, we introduce the notion of *loop balance*.

Loop Balance

Just as machines have balance ratios, so do loops. We can define balance for a specific loop as

$$\beta_L = \frac{\text{number of memory references} = M}{\text{number of flops} = F}.$$

We assume that references to array variables are actually references to memory, while references to scalar variables involve only registers. Memory references are assigned a uniform cost under the assumption that loop interchange, software prefetching or tiling will attain cache locality [WL91, KM92, CKP91].

Comparing β_M to β_L can give us a measure of the performance of a loop running on a particular architecture. If $\beta_L = \beta_M$, the loop is balanced for the machine and will run well on that particular machine. The balance measure favors no particular machine or loop balance. It reports that out-of-balance loops will run well on similarly out-of-balance architectures and balanced loops will run well on similarly balanced architectures. In addition, it reports that performance bottlenecks occur when loop and machine balance do no match. If $\beta_L > \beta_M$, then the loop needs data at a higher rate than the memory system can provide and idle computational cycles will exist. Such a loop is said to be *memory bound* and its performance can be improved by lowering β_L. If $\beta_L < \beta_M$, then data cannot be processed as fast as it is supplied to the processor and memory bandwidth will be wasted. Such a loop is said to be *compute bound.* Compute-bound loops run at the peak floating-point rate of a machine and need not be further balanced. Floating-point operations usually cannot be removed and arbitrarily increasing the number of memory operations is pointless.

1.1.2 Dependence Graph

To aid in the computation of the number of memory references for the application of our transformations, we use a form of dependence graph that exposes reuse of values. We say that a *dependence* exists between two references if there exists a control-flow path from the first reference to the second and both references access the same memory location [Kuc78]. The dependence is

- a *true dependence* or *flow dependence* if the first reference writes to the location and the second reads from it,
- an *antidependence* if the first reference reads from the location and the second writes to it,
- an *output dependence* if both references write to the location, and
- an *input dependence* if both references read from the location.

If two references, v and w, are contained in n common loops, we can refer to separate instances of the execution of the references by an *iteration vector*. An iteration vector, denoted $\vec{i}$, is simply the values of the loop control variables of the loops containing v and w. The set of iteration vectors corresponding to all iterations of the of the loop nest is called the *iteration space*. Using iteration vectors, we can define a *distance vector*, $d = \langle d_1, d_2, \ldots, d_n \rangle$, for each consistent dependence: if v accesses location Z on iteration $\vec{i_v}$ and w accesses location Z on iteration $\vec{i_w}$, the distance vector for this dependence is $\vec{i_w} - \vec{i_v}$. Under this definition, the k^{th} component of the distance vector is equal to the number of iterations of the k^{th} loop (numbered from outermost to innermost) between accesses to Z. For example, given the following loop:

```
      DO 10 I = 1,N
        DO 10 J = 1,N
10        A(I,J) = A(I-1,J) + A(I-2,J+3)
```

the true dependence from `A(I,J)` to `A(I-1,J)` has a distance vector of $\langle 1, 0 \rangle$ and the true dependence from `A(I,J)` to `A(I-2,J+3)` has a distance vector of $\langle 2, -3 \rangle$.

The loop associated with the outermost non-zero distance vector entry is said to be the *carrier*. The distance vector value associated with the carrier loop is called the *threshold* of the dependence. If all distance vector entries are zero, the dependence is *loop independent*. In determining which dependences can be used for memory analysis, we consider only those that have a *consistent threshold* — that is, those dependences for which the threshold is constant throughout the execution of the loop [GJG87, CCK88]. For a dependence to have a consistent threshold, it must be the case that the location accessed by the dependence source on iteration i is accessed by the sink on iteration $i + c$, where c does not vary with i. Formally, if we let

$$\begin{aligned} A(f(\vec{i})) &= A(a_0 + a_1 I_1 + \cdots + a_n I_n) \\ A(g(\vec{i})) &= A(b_0 + b_1 I_1 + \cdots + b_n I_n) \end{aligned}$$

be array references where each a_i and b_i is a constant and each I_j is a loop induction variable (I_n is associated with the innermost loop), then we have the following definition.

Theorem 1.1 A dependence has a consistent threshold iff $a_i = b_i$ for each $1 \leq i \leq n$ and there exists an integer τ such that $b_n \tau = a_0 - b_0$.

Proof See Callahan, et. al [CCK88]. □

Some carried dependences will have multiple distance vector values associated with one entry. Consider the following loop.

```
      DO 10 I = 1, N
10    A(K) = A(K) + ...
```

The true dependence between the references to `A(K)` has the distances $1, 2, 3, \ldots, \texttt{N}-1$ for the entry associated with the `I`-loop. For the purposes of memory management, we will use the minimum value, 1 in this case, as the distance vector entry.

1.2 Transformations To Improve Memory Performance

Based upon the dependence graph described in the previous section, we can apply a number of transformations to improve the performance of memory-bound programs. In this section, we give a brief introduction to these transformations.

1.2.1 Scalar Replacement

In the model of balance presented in the previous section, all array references are assumed to be memory references. The principal reason for this is that the data-flow analysis used by typical scalar compilers is not powerful enough to recognize most opportunities for reuse in subscripted variables. Arrays are treated in a particularly naive fashion, if at all, making it impossible to determine when a specific element might be reused. This, however, need not be the case. In the code shown below,

```
      DO 10 I = 2, N
10      A(I) = A(I-1) + B(I)
```

the value accessed by `A(I-1)` is defined on the previous iteration of the loop by `A(I)` on all but the first iteration. Using this knowledge, obtained via dependence analysis, the flow of values between the references can be expressed with temporaries as follows.

```
       T = A(1)
       DO 10 I = 2, N
          T = T + B(I)
10        A(I) = T
```

Since global register allocation will most likely put scalar quantities in registers, we have removed the load of `A(I-1)`, resulting in a reduction in the balance of the loop from 3 to 2 [CAC$^+$81, CH84, BCKT89]. This transformation is called *scalar replacement* and in Chapter 2, we show how to apply it to loops automatically.

1.2.2 Unroll-and-Jam

Unroll-and-jam is another transformation that can be used to improve the performance of memory-bound loops [AC72, AN87, CCK88]. The transformation unrolls an outer loop and then jams the resulting inner loops back together. Using unroll-and-jam we can introduce more computation into an innermost loop body without a proportional increase in memory references. For example, the loop:

```
     DO 10 I = 1, 2*M
       DO 10 J = 1, N
10       A(I) = A(I) + B(J)
```

after unrolling becomes:

```
     DO 10 I = 1, 2*M, 2
       DO 20 J = 1, N
20       A(I) = A(I) + B(J)
       DO 10 J = 1, N
10       A(I+1) = A(I+1) + B(J)
```

and after jamming becomes:

```
     DO 10 I = 1, 2*M, 2
       DO 10 J = 1, N
         A(I) = A(I) + B(J)
10       A(I+1) = A(I+1) + B(J)
```

In the original loop, we have one floating-point operation and one memory reference after scalar replacement, giving a balance of 1. After applying unroll-and-jam, we have two floating-point operations and still only one memory reference, giving a balance of 0.5. On a machine that can perform twice as many floating-point operations as memory accesses per clock cycle, the second loop would perform better. In Chapter 3, we will show how to apply unroll-and-jam automatically to improve loop balance.

1.2.3 Loop Interchange

Not only are we concerned with the number of references to memory, but also whether the data accessed by a reference is stored in cache or main memory. Consider the following Fortran loop where arrays are stored in column-major order.

```
      DO 10 I = 1, N
        DO 10 J = 1, N
10        A = A + B(I,J)
```

References to successive elements of B by B(I,J) are a long distance apart in number of memory accesses, requiring an extremely large cache to capture the potential cache-line reuse. With the likelihood of cache misses on accesses to B, we can interchange the I- and J-loops to make the distance between successive accesses small, as shown below [AK87, Wol86a].

```
      DO 10 J = 1, N
        DO 10 I = 1, N
10        A = A + B(I,J)
```

Now, we will only have a cache miss on accesses to B once every cache line, resulting in better memory performance. In Chapter 4, we derive a compiler algorithm to apply loop interchange, when safe, to a loop nest to improve cache performance.

1.2.4 Strip-Mine-And-Interchange

Sometimes loops access more data than can be handled by a cache even after loop interchange. In these cases, the iteration space of a loop can be blocked into sections whose reuse can be captured by the cache. Strip-mine-and-interchange is a transformation that achieves this result [Wol87, Por89]. The effect is to shorten the distance between the source and sink of a dependence so that it is more likely for the datum to reside in cache when the reuse occurs. Consider the following example

```
        DO 10 J = 1,N
          DO 10 I = 1,M
  10        A(I) = A(I) + B(I,J)
```

Assuming that the value of M is much greater than the size of the cache, we would miss the opportunity to reuse the values of A on each iteration of J. To capture this reuse, we can use strip-mine-and-interchange. First, we strip mine the I-loop as shown below.

```
      DO 10 J = 1,N
        DO 10 I = 1,M,IS
          DO 10 II = I,MIN(I+IS-1,N)
10          A(II) = A(II) + B(II,J)
```

Note that this loop executes the iterations of the loop in precisely the same order as the original and its reuse properties are unchanged. However, when we interchange the stripped loop with the outer loop to give

```
      DO 10 I = 1,M,IS
        DO 10 J = 1,N
          DO 10 II = I,MIN(I+IS-1,N)
10          A(II) = A(II) + B(II,J)
```

the iterations are now executed in blocks of size IS by N. With this blocking, we can reuse IS values of A out of cache for every iteration of the J-loop if IS is less than half the size of the cache.

Together, unroll-and-jam and strip-mine-and-interchange make up a transformation technique known as *iteration-space blocking*. As discussed, the first is used to block for registers and the second for cache. In Chapter 5, we explore additional knowledge necessary to apply iteration-space blocking to many real-world algorithms.

1.3 Related Work

Much research has been done in memory-hierarchy management using the aforementioned transformations. In this section, we will review the previous work, noting the deficiencies that we intend to address.

1.3.1 Register Allocation

The most significant work in the area of register allocation has been done by Chaitin, et al., and Chow and Hennessy [CAC$^+$81, CH84]. Both groups cast the problem of register allocation into that of graph coloring on an interference graph, where the nodes represent scalar memory locations and an edge between two nodes prevents them from occupying the same register. The objective is to find a k-coloring of the interference graph, where k is the number of machine registers. Since graph coloring is NP-complete, heuristic methods must be applied. It is usually the case that a k-coloring is not possible, requiring values to be spilled from registers to satisfy physical constraints. This method has been shown to be very effective at allocating scalar variables to registers, but because information concerning the flow of array values is lacking, subscripted variables are not handled well.

Allen and Kennedy show how data dependence information can be used to recognize reuse of vector data and how that information can be applied to perform vector register allocation [AK88]. They also present two transformations, loop interchange and loop fusion, as methods to improve vector register allocation opportunities. Both of these transformations were originally designed to enhance loop parallelism. Because dependences represent the flow of values in an array, Allen and Kennedy suggest that this information could be used to recognize reuse in arrays used in scalar computations.

Callahan, Cocke and Kennedy have expanded these ideas to develop scalar replacement as shown in Section 1.2.1 [CCK88]. Their method is only applicable to loops that have no inner-loop conditional control flow; therefore, it has limited applicability. Also, their algorithm does not consider register pressure and may expose more reuse than can be handled by a particular machine's register file. If spill code must be inserted, the resulting performance degradation may negate most of the value of scalar replacement.

Callahan, Cocke and Kennedy also use unroll-and-jam to improve the effectiveness of scalar replacement and to increase the amount of low-level parallelism in the inner-loop body. Although the mechanics of unroll-and-jam are described in detail, there is no discussion of how or when to apply the transformation to a loop nest.

Aiken and Nicolau present a transformation identical to unroll-and-jam, called *loop quantization* [AN87]. However, they do not use this transformation to increase data locality, but rather to improve inner-loop parallelism. They perform a *strict* quantization, where the unrolled iterations of the original loop in the body of the unrolled loop are data-independent. This means that they do not improve the data locality in the innermost loop for true dependences; therefore, little reduction in memory references can be obtained.

1.3.2 Memory Performance Studies

Previous studies have shown how poor cache behavior can have disastrous effects on program performance. Abu-Sufah and Malony showed that the performance of the LANL BMK8A1 benchmark fell by a factor of as much as 2.26 on an Alliant FX/8 when vector sizes were too large to be maintained in cache [ASM86, GS84]. Similarly, Liu and Strother found that vector performance on the IBM 3090 fell by a factor of 1.40 when vector lengths exceeded the cache capacity [LS88]. In this second study, it was also shown that if the vector code were blocked into smaller sections that fit into cache, the optimal performance was regained. Porterfield reported that on computers with large memory latencies, many large scientific programs spent half of their execution time waiting for data to be delivered to cache [Por89].

A number of other studies have shown the effectiveness of blocking loops for cache performance. Gallivan, et al., show that on the Alliant FX/8, the blocked version of LU decomposition is nearly 8 times faster than the unblocked version, using BLAS3 and BLAS2 respectively [GJMS88, DDHH88, DDDH90]. The BLAS2 version performs a rank 1 update of the matrix while the best BLAS3 version performs a blocked rank 96 update. Also on the Alliant FX/8, Berry and Sameh have achieved speedups of as large as 9 over the standard LINPACK versions for solving tridiagonal linear systems [BS88, DBMS79] and on the Cray-2, Calahan showed that blocking LU decomposition improved the performance by a factor of nearly 6 [Cal86].

All of these studies involved tedious hand optimization to attain maximal performance. The BLAS primitives are noteworthy examples of this methodology, in which each primitive must be recoded in assembly language to get performance on each separate architecture [LHKK79]. Hand optimization is less than ideal because it takes months to code the BLAS primitives by hand, although recoding the whole program is a worse alternative. Therefore, significant motivation exists for the development of a restructuring compiler that can optimize for any memory hierarchy and relieve the programmer from the tedious task of memory-hierarchy management.

1.3.3 Memory Management

The first major work in the area of memory management by a compiler is that of Abu-Sufah on improving virtual memory performance [AS78]. In his thesis, he describes the use of a restructuring compiler to improve the virtual memory behavior of a program. Through the use of dependence analysis, Abu-Sufah is able to perform transformations on the loop structure of a program to reduce the number of physical pages required and to group accesses to each page.

The transformations that Abu-Sufah uses to improve virtual memory performance of loops are clustering (or loop distribution), loop fusion and page indexing (a combination of strip mining, interchange and distribution). Clustering is used to split the loop into components whose name spaces (or working sets) are disjoint. Loop fusion is then used to fuse components which originated in different loops but whose name spaces intersect. The page indexing transformation is used to block a loop nest so that all the references to a page are contained in one iteration over a block, thus maximizing locality. The goal is a set of loops whose working sets are disjoint; hence, the required number of physical pages is reduced without increasing the page fault rate.

Although Abu-Sufah's transformation system shows the potential for improving a program's memory behavior, his use of loop distribution can increase the amount of pipeline interlock and cause performance degradation [CCK88]. Additionally, the fact that virtual memory is fully associative rather than set associative prevents the application of his model to cache management.

Fabri's work in automatic storage optimization concentrates on extended graph-coloring techniques to manage storage overlays in memory [Fab79]. She presents the notion of array renaming (analogous to live-range splitting) to minimize the memory requirements of a program. However, the problem of storage overlays does not map to cache management since the compiler does not have explicit control over the cache.

Thabit has examined software methods to improve cache performance through the use of packing, prefetching and loop transformations [Tha81]. He shows that optimal packing of scalars for cache-line reuse is an NP-complete problem and proposes some heuristics. However, scalars are not considered to be a significant problem because of register-allocation technology. In addition, Thabit maps graph-coloring to the allocation of arrays to eliminate cache interference caused by set associativity. However, no evaluation of the effectiveness of this approach is given. In particular, he does not address the problem of an array interfering with itself. Finally, Thabit establishes the safety conditions of loop distribution and strip-mine-and-interchange.

Wolfe's memory performance work has concentrated on developing transformations to reshape loops to improve their cache performance [Wol87]. He shows how *tiling* (or *iteration-space blocking*) can be used to improve the memory performance of program loops. Wolfe also shows that his techniques for advanced loop interchange can be used to tile loops with non-rectangular iteration spaces and loops that are not perfectly nested [Wol86a]. In particular, he discusses blocking for triangular- and trapezoidal-shaped iteration spaces, but he does not present an algorithm. Instead, he illustrates the transformation by a few examples.

Irigoin and Triolet describe a new dependence abstraction, called a *dependence cone*, that can be used to block code for two levels of parallelism and two levels of memory [IT88]. The dependence cone gives more information than a distance vector by maintaining a system of linear inequalities that can be used to derive all dependences. The cone allows a larger set of perfectly nested loops to be transformed than other dependence abstractions by providing a general framework to partition an iteration space into *supernodes* (or blocks). The idea is to aggregate many loop iterations, so as to provide vector statements, parallel tasks and data reference locality. To improve memory performance, the supernodes can be chosen to maximize the amount of data locality in each supernode. Unfortunately, this technique does not work on imperfectly nested loops nor does it handle partially blockable loops, both of which occur in linear algebra codes.

Wolfe presents work that is very similar to Irigoin and Triolet's [Wol89]. He does not use the dependence cone as the dependence abstraction, but instead he uses the standard *distance vector*. Using loop skewing, loop interchange and strip mining, he can tile an iteration space into blocks which have both data locality and parallelism. Wolfe is limited by the transformations that he applies and by the restrictive nature of the subscripts handled with distance vectors, but he is capable of handling non-perfectly nested loops. The main advantage of Wolfe's techniques over Triolet's is that Wolfe's method is more straightforward to implement in a restructuring compiler, although the complexity of the algorithm is $O(d!)$, where d is the loop nesting depth.

Gannon, et al., present a technique to describe the amount of data that must be in the cache for reuse to be possible [GJG87]. They call this the *reference window*. The window for a dependence describes all of the data that is brought into the cache for the two references from the time that one datum is accessed at the source until it is used again at the sink. A family of reference windows for an array represents all of its elements that must fit into cache to capture all of the reuse. To determine if the cache is large enough to hold every window, the window sizes are summed and compared against the size of the cache. If the cache is too small, blocking transformations such as strip-mine-and-interchange can be used to decrease the size of the reference windows.

Porterfield proposes a method to determine when the data referenced in a loop does not fit entirely in cache[Por89]. He develops the idea of an *overflow iteration*, which is that iteration of a loop that brings in the data item which will cause the cache to overflow. Using this measurement, Porterfield can predict when a loop needs to be tiled to improve memory performance and to determine the size of a one dimensional tile for the loop that causes the cache to overflow.

Porterfield also presents two new transformations to block loops. The first is *peel-and-jam*, which can be used to fuse loops that have certain types of fusion preventing dependences by peeling off the offending iterations of the first loop and fusing the resulting loop bodies. The second is either a combination loop skewing, interchanging and strip-mining or loop unrolling, peeling and jamming to perform *wavefront blocking*. The key technique here is the use of non-reordering transformations (skewing, peeling and unrolling) to make it possible to block loops. Some of these non-reordering transformations will become especially important when dealing with partially blockable loops.

Porterfield also discusses the applicability of blocking transformations to twelve Fortran programs. He organizes them into three categories: transformable, semi-transformable and non-transformable. One-third of the programs in his study were transformable since the blocking transformations were directly applicable. The semi-transformable programs contained coding styles that made it difficult to transform them automatically, and in the case of the non-transformable program, partial pivoting was the culprit. Porterfield claims that a compiler cannot perform iteration-space blocking in the presence of partial pivoting, but his analysis is not extensive enough to make this claim. He does not consider increasing the "intelligence" of the compiler to improve its effectiveness.

Lam and Wolf present a framework for determining memory usage within loop nests and use that framework to apply loop interchange, loop skewing, loop reversal, tiling and unroll-and-jam [WL91, Wol86b]. Their method does not work on non-perfectly nested loops and does not encompass a technique to determine unroll-and-jam amounts automatically. Additionally, they do not necessarily derive the best block algorithm with their technique, leaving the possibility that suboptimal performance is still possible.

Lam, Rothberg and Wolf present a method to determine block sizes for block algorithms automatically [LRW91]. Their results show that typical effective block sizes use less than 10% of the cache. They suggest the use of copy optimizations to remove the effects of set associativity and allow the use of larger portions of the cache.

Kennedy and McKinley present a simplified model of cache performance that only considers inner-loop reuse [KM92]. Any reuse that occurs across an outer-loop iteration is considered to be prevented by cache interference. Using this model, they describe a method to determine the number of cache lines required by a loop if it were innermost and then they reorder the loops to use the minimum number of cache lines. This simple model is less precise than the Lam and Wolf model, but is very effective in practice. While the number of cache lines used by a loop is related to memory performance, it is not a direct measure of performance. Their work is directed toward shared-memory parallel architectures where bus contention is a real performance problem and minimizing main memory accesses is vital. On scalar processors with a higher cache-access cost, the number of cache lines accessed may not be an accurate measure of performance.

1.4 Overview

In the rest of this thesis, we will explore the ability of the compiler to optimize a program automatically for a machine's memory hierarchy. We present algorithms to apply transformations to improve loop balance and we present experiments to validate the effectiveness of the algorithms. In Chapter 2, we address scalar replacement in the presence of conditional-control flow. In Chapter 3, we discuss automatic unroll-and-jam to improve loop balance. Chapter 4 addresses loop interchange to improve cache performance and in Chapter 5, we analyze the applicability to real-world numerical subprograms of compiler blocking techniques that further optimize cache performance. Finally, we present our conclusions and future work in Chapter 6.

Chapter 2

Scalar Replacement

Although conventional compilation systems do a good job of allocating scalar variables to registers, their handling of subscripted variables leaves much to be desired. Most compilers fail to recognize even the simplest opportunities for reuse of subscripted variables. For example, in the code shown below,

```
      DO 10 I = 1, N
         DO 10 J = 1, M
10          A(I) = A(I) + B(J)
```

most compilers will not keep `A(I)` in a register in the inner loop. This happens in spite of the fact that standard optimization techniques are able to determine that the address of the subscripted variable is invariant in the inner loop. On the other hand, if the loop is rewritten as

```
       DO 20 I = 1, N
          T = A(I)
          DO 10 J = 1, M
10           T = T + B(J)
20        A(I) = T
```

even the most naive compilers allocate `T` to a register in the inner loop.

The principal reason for the problem is that the data-flow analysis used by standard compilers is not powerful enough to recognize most opportunities for reuse of subscripted variables. Subscripted variables are treated in a particularly naive fashion, if at all, making it impossible to determine when a specific element might be reused. This is particularly problematic for floating-point register allocation because most of the computational quantities held in such registers originate in subscripted arrays.

Scalar replacement is a transformation that uses dependence information to find reuse of array values and expose it by replacing the references with scalar temporaries as was done in the above example [CCK88, CCK90]. By encoding the reuse of array elements in scalar temporaries, we can give a coloring register allocator the opportunity to allocate values held in arrays to registers [CAC+81].

Although previous algorithms for scalar replacement have been shown to be effective, they have only handled loops without conditional-control flow [CCK90]. The principle reason for past deficiencies is the reliance solely upon dependence information. A dependence contains little information concerning control flow between its source and sink. It only reveals that both statements *may* be executed. In the loop,

```
      DO 10 I = 1,N
 5       IF (M(I) .LT. 0) A(I) = B(I) + C(I)
10       D(I) = A(I) + E(I)
```

the true dependence from statement `5` to statement `10` does not reveal that the definition of `A(I)` is conditional. Using only dependence information, previous scalar replacement algorithms would produce the following incorrect code.

```
      DO 10 I = 1,N
         IF (M(I) .LT. 0) THEN
 5          A0 = B(I) + C(I)
            A(I) = A0
         ENDIF
10       D(I) = A0 + E(I)
```

If the result of the predicate is false, no definition of A0 will occur, resulting in an incorrect value for A0 at statement 10. To ensure A0 has the proper value, we can insert a load of A0 from A(I) on the false branch, as shown below.

```
      DO 10 I = 1,N
         IF (M(I) .LT. 0) THEN
5           A0 = B(I) + C(I)
            A(I) = A0
         ELSE
            A0 = A(I)
         ENDIF
10       D(I) = A0 + E(I)
```

The hazard with inserting instructions is the potential to increase run-time costs. In the previous example, we have avoided the hazard. If the true branch is taken, one load of A(I) is removed. If the false branch is taken, one load of A(I) is inserted and one load is removed. It will be a requirement of our scalar replacement algorithm to prevent an increase in run-time accesses to memory.

This chapter addresses scalar replacement in the presence of forward conditional control flow. We show how to map *partial redundancy elimination* to scalar replacement in the presence of conditional control flow to ensure that memory costs will not increase along any execution path [MR79, DS88]. This chapter begins with an overview of partial redundancy elimination. Then, a detailed derivation of our algorithm for scalar replacement is given. Finally, an experiment with an implementation of this algorithm is reported.

2.1 Partial Redundancy Elimination

In the elimination of partial redundancies, the goal is to remove the latter of two identical computations that are performed on a given execution path. A computation is partially redundant when there may be paths on which both computations are performed and paths on which only the latter computation is performed. In Figure 2.1, the expression A+B is redundant along one branch of the IF and not redundant along the other. Partial redundancy elimination will remove the computation C=A+B, replacing it with an assignment, and insert a computation of A+B on the path where the expression does not appear (see Figure 2.2). Because there may be no basic block in which new computations can be inserted on a particular path, insertion is done on flow-graph edges and new basic blocks are created when necessary [DS88].

The essential property of this transformation is that it is guaranteed not to increase the number of computations performed along any path [MR79]. In mapping partial redundancy elimination to scalar replacement, references to array expression can be seen as the computations. A load or a store followed by another load from the same location represents a redundant load that can be removed. Thus, using this mapping will guarantee that the number of memory accesses in a loop will not increase. However, the mapping that we use will not guarantee that a minimal number of loads is inserted.

2.2 Algorithm

In this section, we present an algorithm for scalar replacement in the presence of forward conditional-control flow. We begin by determining which array accesses provide values for a particular array reference. Next, we link together references that share values by having them share temporary names. Finally, we generate scalar replaced code. For partially redundant array accesses, partial redundancy elimination is mapped to the elimination of partially redundant array accesses.

2.2.1 Control-Flow Analysis

Our optimization strategy focuses on loops; therefore, we can restrict control-flow analysis to loop nests only. Furthermore, it usually makes no sense to hold values across iterations of outer loops for two reasons.

1. There may be no way to determine the number of registers needed to hold all the values accessed in the innermost loop because of symbolic loop bounds.
2. Even if we know the register requirement, it is doubtful that the target machine will have enough registers.

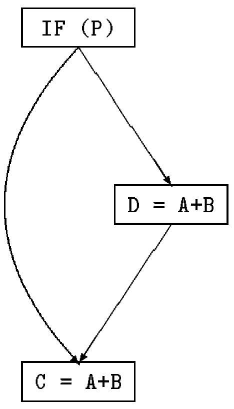

Figure 2.1 Partially Redundant Computation

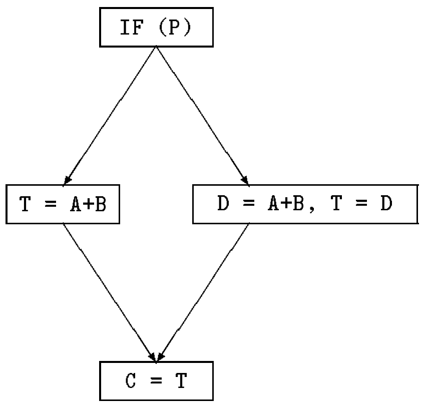

Figure 2.2 After Partial Redundancy Elimination

Hence, we need only perform control-flow analysis on each innermost loop body.

To simplify our analysis, we impose a few restrictions on the control-flow graph. First, the flow graph of the innermost loop must be reducible. Second, backward jumps are not allowed within the innermost DO-loop because they potentially create loops. Finally, multiple loop exits are prohibited. This restriction is for simplicity and can be removed with slight modification to the algorithm.

2.2.2 Availability Analysis

The first step in performing scalar replacement is to calculate *available array expressions.* Here, we will determine if the value provided by the source of a dependence is generated on every path to the sink of the dependence. We assume that enough iterations of the loop have been peeled so values can be available upon entry to the loop. Since each lexically identical array reference accesses the same memory location on a given loop iteration, we do not treat each lexically identical array reference as a separate array expression. Rather, we consider them in concert.

Array expressions most often contain references to induction variables. Therefore, their naive treatment in availability analysis is inadequate. To illustrate this, in the loop,

```
      DO 30 I = 1,N
        IF (B(I) .LT. 0.0) THEN
10        C(I) = A(I) + D(I)
        ELSE
20        C(I) = A(I-1) + D(I)
        ENDIF
30      E(I) = C(I) + A(I)
```

the value accessed by the array expression A(I) is fully available at the reference to A(I-1) in statement 20, but it is not available at statement 10 and is only partially available at statement 30. Using a completely syntactic notion of array expressions, essentially treating each lexically identical expression as a scalar, A(I) will be incorrectly reported as available at statements 10 and 30. Thus, more information is required. We must account for the fact that the value of an induction variable contained in an array expression changes on each iteration of the loop.

A convenient solution is to split the problem into loop-independent availability, denoted LIAV, where the back edge of the loop is ignored, and loop-carried availability, LCAV, where the back edge is included. Thus, an array expression is only available if it is in LIAV and there is a consistent incoming loop-independent dependence, or if it is in LCAV and there is a matching consistent incoming loop-carried dependence. The data-flow equations for availability analysis are shown below.

$$\begin{aligned}\text{LIAVIN}(\text{B}) &= \bigcap_{p\in preds\ (\text{B})} \text{LIAVOUT}(p)\\ \text{LIAVOUT}(\text{B}) &= (\text{LIAVIN}(\text{B}) - \text{LIKILL}(\text{B}))\bigcup \text{LIGEN}(\text{B})\end{aligned}$$

$$\begin{aligned}\text{LCAVIN}(\text{B}) &= \bigcap_{p\in preds\ (\text{B})} \text{LCAVOUT}(p)\\ \text{LCAVOUT}(\text{B}) &= \text{LCAVIN}(\text{B})\bigcup \text{LCGEN}(\text{B})\end{aligned}$$

For LIAV, an array expression is added to GEN when it is encountered whether it is a load or a store. At each store, the sources of all incoming inconsistent dependences are added to KILL and removed from GEN. At loads, nothing is done for KILL because a previously generated value cannot be killed. We call these sets LIGEN and LIKILL.

For LCAV, we must consider the fact that the flow of control from the source to the sink of a dependence will include at least the next iteration of the loop. Subsequent loop iterations can effect whether a value has truly been generated and not killed by the time the sink of the dependence is reached. In the loop,

```
      DO 10 I = 1,N
        IF (A(I) .GT. 0.0) THEN
          C(I) = B(I-2) + D(I)
        ELSE
          B(K) = C(I) + D(I)
        ENDIF
10      B(I) = E(I) + C(I)
```

the value generated by B(I) on iteration I=1 will be available at the reference to B(I-2) on iteration I=3 only if the false branch of the IF statement is not taken on iteration I=2. Since determining the direction of a branch at compile time is undecidable in general, we must assume that the value generated by `B(I)` will be killed by the definition of `B(K)`. In general, any definition that is the source of an inconsistent output dependence can never be in LCGEN(B), $\forall$ B. It will always be killed on the current or next iteration of the loop. Therefore, we need only compute the LCGEN and not LCKILL.

There is one special case where this definition of LCGEN will unnecessarily limit the effectiveness of scalar replacement. When a dependence threshold is 1, it may happen that the sink of the generating dependence edge occurs before the killing definition. Consider the following loop.

```
      DO 10 I = 1,N
        B(K) = B(I-1) + D(I)
10      B(I) = E(I) + C(I)
```

the value used by `B(I-1)` that is generated by `B(I)` will never be killed by `B(K)`. The solution to this limitation is to create a new set called LCAVIF1 that contains availability information only for loop-carried dependences with a threshold of 1. Control flow through the current and next iterations of the loop is included when computing this set. The data-flow equations for LCAVIF1 are identical to those of LIAV. This is because we consider control flow on the next iteration of the loop, unlike LCAV. Below are the data-flow equations for LCAVIF1.

$$\begin{aligned} \text{LCAVIF1IN(B)} &= \bigcap_{p \in preds\ (\text{B})} \text{LCAVIF1OUT}(p) \\ \text{LCAVIF1OUT(B)} &= (\text{LCAVIF1IN(B)} - \text{LIKILL(B)}) \bigcup \text{LIGEN(B)} \end{aligned}$$

Because we consider both fully and partially redundant array accesses, we need to compute partial availability in order to scalar replace references whose load is only partially redundant. As in full-availability analysis, we partition the problem into loop-independent, loop-carried and loop-carried-if-1 sets. Computation of KILL and GEN corresponds to that of availability analysis. Below are the data-flow equations for partially available array expression analysis.

$$\begin{aligned} \text{LIPAVIN(B)} &= \bigcup_{p \in preds\ (\text{B})} \text{LIPAVOUT}(p) \\ \text{LIPAVOUT(B)} &= (\text{LIPAVIN(B)} - \text{LIKILL(B)}) \bigcup \text{LIGEN(B)} \\ \text{LCPAVIN(B)} &= \bigcup_{p \in preds\ (\text{B})} \text{LCPAVOUT}(p) \\ \text{LCPAVOUT(B)} &= \text{LCPAVIN(B)} \bigcup \text{LCGEN(B)} \\ \text{LCPAVIF1IN(B)} &= \bigcup_{p \in preds\ (\text{B})} \text{LCPAVIF1OUT}(p) \\ \text{LCPAVIF1OUT(B)} &= (\text{LCPAVIF1IN(B)} - \text{LIKILL(B)}) \bigcup \text{LIGEN(B)} \end{aligned}$$

2.2.3 Reachability Analysis

Because there may be multiple lexically identical array references within a loop, we want to determine which references actually supply values that reach a sink of a dependence and which supply values that are killed before reaching such a sink. In other words, which values *reach* their potential reuses. In computing reachability, we do not treat each lexically identical array expression in concert. Rather, each reference is considered independently. Reachability information along with availability is used to select which array references provide values for scalar replacement. While reachability information is not required for correctness, it can prevent the marking of references as providing a value for scalar replacement when that value is redefined by a later identical reference. This improves the readability of the transformed code.

We partition the reachability information into three sets: one for loop-independent dependences (LIRG), one for loop-carried dependences with a threshold of 1 (LCRGIF1) and one for other loop-carried dependences (LCRG). Calculation of LIGEN and LIKILL is the same as that for availability analysis except in one respect. The source of any incoming loop-independent dependence whose sink is a definition is killed whether the threshold is consistent or inconsistent. Additionally, LIKILL is subtracted from LCRGOUT to account for consistent references that redefine a value on the current iteration of a loop. For example, in the following loop, the definition of `B(I)` kills the load from `B(I)`, therefore, only the definition reaches the reference to `B(I-1)`.

```
      DO 10 I = 1,N
        A(I) = B(I-1) + B(I)
10      B(I) = E(I) + C(I)
```

Using reachability information, the most recent access to a value can be determined. Even using LIKILL when computing LCRG will not eliminate all unreachable references. References with only outgoing consistent loop-carried output or antidependences will not be killed. This does not effect correctness, rather only readability, and can only happen when partially available references provide the value to be scalar replaced. Below are the data-flow equations used in computing array-reference reachability.

$$\begin{aligned}
\text{LIRGIN(B)} &= \bigcup_{p \in preds\ (\text{B})} \text{LIRGOUT}(p) \\
\text{LIRGOUT(B)} &= (\text{LIRGIN(B)} - \text{LIKILL(B)}) \bigcup \text{LIGEN(B)} \\
\text{LCRGOUT(B)} &= \bigcup_{p \in preds\ (\text{B})} \text{LCRGOUT(P)} \\
\text{LCRGOUT(B)} &= (\text{LCRGIN(B)} - \text{LIKILL(B)}) \bigcup \text{LCGEN(B)} \\
\text{LCRGIF1IN(B)} &= \bigcup_{p \in \text{PREDS}\ (\text{B})} \text{LCRGIF1OUT(P)} \\
\text{LCRGIF1OUT(B)} &= (\text{LCRGIF1IN(B)} - \text{LIKILL(B)}) \bigcup \text{LIGEN(B)}
\end{aligned}$$

2.2.4 Potential-Generator Selection

At this point, we have enough information to determine which array references potentially provide values to the sinks of their outgoing dependence edges. We call these references *potential generators* because they can be seen as generating the value used at the sink of some outgoing dependence. The dependences leaving a generator are called *generating dependences*. Generators are only potential at this point because we need more information to determine if scalar replacement will even be profitable. In Figure 2.3, we present the algorithm *FindPotentialGenerators* for determining potential generators.

We have two goals in choosing potential generators. The first is to insert the fewest number of loads, correlating to the maximum number of memory accesses removed. The second is to minimize register pressure, or the number of registers required to eliminate the loads. To meet the first objective, fully available expressions are given the highest priority in generator selection. To meet the second, loop-independent fully available generators are preferred because they require the fewest number of registers. If no loop-independent generator exists, loop-carried fully available generators are considered next. If there are multiple such generators, the one that requires the fewest registers (the one with the smallest threshold) is chosen.

If there are no fully available generators, partially available array expressions are next considered as generators. Partially available generators do not guarantee a reduction in the number of memory accesses because memory loads need to be inserted on paths along which a value is needed but not generated. However, we can guarantee that there will not be an increase in the number of memory loads by only inserting load instructions if they are guaranteed to have a corresponding load removal on any execution path [MR79]. Without this guarantee, we may increase the number of memory accesses at execution time, resulting in a performance degradation.

The best choice for a partially available generator would be one that is loop-independent. Although there may be a "more available" loop-carried generator, register pressure is kept to a minimum and scalar replacement will be applied if we chose a loop-independent generator. If there are no loop-independent partially available array expressions, then the next choice would be a loop-carried partially available array expression with a generating dependence having the largest threshold of any incoming potential generating dependence. Although this contradicts the goal of keeping the register pressure to a minimum, we increase the probability that there will be a use of the value on every path by increasing the window size for potential uses of the generated value. We have chosen to sacrifice register pressure for potential savings in memory accesses.

Finally, when propagating data-flow sets through a basic block to determine availability or reachability at a particular point, information is not always incrementally updated. For loop-independent information, we update a data-flow set with GEN and KILL information as we encounter statements. However, the same is not true for loop-carried and loop-carried-if-1 information. GEN information is not used to update any loop-carried data-flow set. Loop-carried information must propagate around the loop to be valid and a loop-carried data-flow set at the entry to a basic block already contains this information. KILL information is only incrementally updated for loop-carried-if-1 sets since flow through the second iteration of a loop after a value is generated is considered.

```
procedure FindPotentialGenerators($G$)

Input: $G = (V, E)$, the dependence graph
Defs: $\text{B}_v$ = basic block containing $v$

  for each $v \in V$ do
    if $\exists e \in E | e = (w, v), w \in \text{LIAV}(\text{B}_v), w \in \text{LIRG}(\text{B}_v), d_n(e) = 0$ then
      mark all such $w$'s as $v$'s potential LIAV generator
    else
      if $\exists e \in E | e = (w, v), w \in \text{LCAVIF1}(\text{B}_v), w \in \text{LCRGIF1}(\text{B}_v)$,
        $w \in \text{LCAVIF1OUT}(\text{EXIT}), w \in \text{LCRGIF1OUT}(\text{EXIT}), d_n(e) = 1$ then
        mark all such $w$'s as $v$'s potential LCAV generator
    else
      if $\exists e \in E | e = (w, v), w \in \text{LCAVOUT}(\text{EXIT}), w \in \text{LCRGOUT}(\text{EXIT})$,
        $d_n(e) > 0, \min_{f \in E_v}(d_n(f)) = d_n(e)$ then
        mark all such $w$'s as $v$'s potential LCAV generator
    else
      if $\exists e \in E | e = (w, v), w \in \text{LIPAV}(\text{B}_v), w \in \text{LIRG}(\text{B}_v), d_n(e) = 0$ then
        mark all such $w$'s as $v$'s potential LIPAV generator
    else
      if $\exists e \in E | e = (w, v), w \in \text{LCPAVOUT}(\text{EXIT}), w \in \text{LCRGOUT}(\text{EXIT})$,
        $d_n(e) > 0), \max_{f \in E_v}(d_n(f)) = d_n(e)$ then
        mark all such $w$'s as $v$'s potential LCPAV generator
    else
      if $\exists e \in E | e = (w, v), w \in \text{LCPAVIF1}(\text{B}_v), w \in \text{LCRGIF1}(\text{B}_v)$,
        $w \in \text{LCPAVIF1OUT}(\text{EXIT}), w \in \text{LCRGIF1OUT}(\text{EXIT}), d_n(e) = 1$ then
        mark all such $w$'s as $v$'s potential LCPAV generator
    else
      $v$ has no generator
```

Figure 2.3 FindPotentialGenerators

In the next portion of the scalar replacement algorithm, we will ensure that the value needed at a reference to remove its load is fully available. This will involve insertion of memory loads for references whose generator is partially available. We guarantee through our partial redundancy elimination mapping that we will not increase the number of memory accesses at run time. However, we do not guarantee a minimal insertion of memory accesses.

2.2.5 Anticipability Analysis

After determining potential generators, we need to locate the paths along which loads need to be inserted to make partially available generators fully available. Loads need to be inserted on paths along which a value is needed but not generated. We have already encapsulated value generation in availability information. Now, we encapsulate value need with *anticipability*. The value generated by an array expression, v, is anticipated by an array expression w if there is a true or input edge $v \longrightarrow w$ and v is w's potential generator.

```
      DO 6 I = 1,N
        IF (A(I) .GT. 0.0)
5         B(I) = C(I) + D(I)
        ELSE
          F(I) = C(I) + D(I)
        ENDIF
6       C(I) = E(I) + B(I)
```

In the above example, the value generated by the definition of `B(I)` in statement `5` is anticipated at the use of `B(I)` in statement `6`.

As in availability analysis, we consider each lexically identical array expression in concert. We split the problem, but this time into only two partitions: one for loop-independent generators, LIAN, and one for loop-carried generators, LCAN. We do not consider the back edge of the loop during analysis for either partition. For LIAN, the reason is obvious. For LCAN, we only want to know that a value is anticipated on all paths through the loop. This is to ensure that we do not increase the number of memory accesses in the loop. In each partition, an array expression is added to GEN at an array reference if it is a potential generator for that reference. For members of LIAN, array expressions are killed at the point where they are defined by a consistent or inconsistent reference. For LCAN, only inconsistent definitions kill anticipation because consistent definitions do not define the value being anticipated on the current iteration. For example, in the loop

```
    DO 1 I = 1,N
      A(I) = B(I) + D(I)
1     B(I) = A(I-1) + C(I)
```

the definition of `A(I)` does not redefine a particular value anticipated by `A(I-1)`. The value is generated by `A(I)` and then never redefined because of the iteration change.

$$\begin{aligned}
\text{LIANOUT(B)} &= \textstyle\bigcap_{s \in succs\ (\text{B})} \text{LIANIN}(s)\\
\text{LIANIN(B)} &= (\text{LIANOUT(B)} - \text{LIKILL(B)}) \textstyle\bigcup \text{LIGEN(B)}\\
\text{LCANOUT(B)} &= \textstyle\bigcap_{s \in succs\ (\text{B})} \text{LCANIN}(s)\\
\text{LCANIN(B)} &= \text{LCANOUT(B) - LCKILL(B)} \textstyle\bigcup \text{LCGEN(B)}
\end{aligned}$$

2.2.6 Dependence-Graph Marking

Once anticipability information has been computed, the dependence graph can be marked so that only dependences to be scalar replaced are left unmarked. The other edges no longer matter because their participation in value flow has already been considered. Figure 2.4 shows the algorithm *MarkDependenceGraph*.

At a given array reference, we mark any incoming true or input edge that is inconsistent and any incoming true or input edge that has a symbolic threshold or has a threshold greater than that of the dependence edge from the potential generator. Inconsistent and symbolic edges are not amenable to scalar replacement because it is impossible to determine the number of registers needed to expose potential reuse at compile time. When a reference has a consistent generator, all edges with threshold less than or equal to the generating threshold are left in the graph. This is to facilitate the consistent register naming discussed in subsequent

```
Procedure MarkDependenceGraph($G$)

Input: $G = (V, E)$, the dependence graph

  for each $v \in V$
    if $v$ has no generator then
      mark $v$'s incoming true and input edges
    else if $v$'s generator is inconsistent or symbolic $d_n$ then
        mark $v$'s incoming true and input edges
        $v$ no longer has a generator
    else if $v$'s generator is LCPAV and $v \notin$ LCANTIN(ENTRY)
        mark $v$'s incoming true and input edges
        $v$ no longer has a generator
    else
      $\tau_v$ = threshold of edge from $v$'s generator
      mark $v$'s incoming true and input edges with $d_n > \tau_v$
      mark $v$'s incoming edges whose source does not reach $v$ or
        whose source is not partially available at $v$
    mark $v$'s incoming inconsistent and symbolic edges
```

Figure 2.4 MarkDependenceGraph

sections. It ensures that any reference occurring between the source and sink of an unmarked dependence that can provide the value at the sink will be connected to the dependence sink. Finally, any edge from a loop-carried partially available generator that is not anticipated at the entry to the loop is removed because there will not be a dependence sink on every path.

2.2.7 Name Partitioning

At this point, the unmarked dependence graph represents the flow of values for references to be scalar replaced. We have determined which references provide values that can be scalar replaced. Now, we move on to linking together the references that share values. Consider the following loop.

```
      DO 6 I = 1,N
5        B(I) = B(I-1) + D(I)
6        C(I) = B(I-1) + E(K)
```

After performing the analysis discussed so far, the generator for the reference to `B(I-1)` in statement `6` would be the load of `B(I-1)` in statement `5`. However, the generator for this reference is the definition of `B(I)` in statement `5` making `B(I-1)` in statement `5` an intermediate point in the flow of the value rather than the actual generator for `B(I-1)` in statement `6`. These references need to be considered together when generating temporary names because they address the same memory locations.

The nodes of the dependence graph can be partitioned into groups by dependence edges (see Figure 2.5) to make sure that temporary names for all references that participate in the flow of values through a memory location are consistent. Any two nodes connected by an unmarked dependence edge after graph marking belong in the same partition. Partitioning is accomplished by performing a traversal of the dependence graph, following unmarked true and input dependences only since these dependences represent value flow. Partitioning will tie together all references that access a particular memory location and represent reuse of a value in that location.

After name partitioning is completed, we can determine the number of temporary variables (or registers) that are necessary to perform scalar replacement on each of the partitions. To calculate register requirements, we split the references (or partitions) into two groups: variant and invariant. Variant references contain the innermost-loop induction variable within their subscript expression. Invariant references do not contain the

```
Procedure GenerateNamePartitions($G$, $P$)

Input: $G = (V, E)$, the dependence graph
       $P =$ the set of name partitions

  $\forall v \in V$ mark $v$ as unvisited
  $i = 0$
  for each $v \in V$
    if $v$ is unvisited then
      put $v$ in $P_i$
      Partition($v$, $P_i$)
      $i = i + 1$
  for each $p \in P$ do
    FindGenerator($p$, $\gamma_p$)
    if $p$ is invariant then
      $r_p = 1$
    else
      $r_p = \max_{g \in \gamma_p}$(CalcNumberOfRegisters($g$) + 1)
  enddo
end

Procedure Partition($v, p$)
  mark $v$ as visited
  add $v$ to $p$
  for each $(e = (v, w) \vee (w, v)) \in E$
    if $e$ is true or input and unmarked and $w$ not visited then
      Partition($w, p$)
end

Procedure FindGenerator($p$, $\gamma$)
  for each $v \in p$ that is a potential generator
    if ($v$ is invariant $\wedge$ ($v$ is a store $\vee$
        $v$ has no incoming unmarked loop-independent edge)) $\vee$
        ($v$ is variant $\wedge$ $v$ has no unmarked incoming true or
        input dependences from another $w \in p$) then
      $\gamma = \gamma \cup v$
end

Procedure CalcNumberOfRegisters(v)
  dist = 0
  for each $e = (v, w) \in E$
    if $e$ is true or input and unmarked $\wedge$ $w$ not visited $\wedge$
        $P(w) = P(v)$ then
      dist = max(dist,$d_n(e)$+ CalcNumberOfRegisters($w$))
  return(dist)
end
```

Figure 2.5 GenerateNamePartitons

innermost-loop induction variable. In the previous example, the reference to `E(K)` is invariant with respect to the `I`-loop while all other array references are variant.

For variant references, we begin by finding all references within each partition that are first to access a value on a given path from the loop entry. We call these references *partition generators*. A variant partition generator will already have been selected as a potential generator in Section 2.2.4 and will not have incoming unmarked dependences from another reference within its partition. The set of partition generators within one partition is called the generator set, γ_p. Next, the maximum over all dependence distances from a potential generator to a reference within this partition is calculated. Even if dependence edges between the generator and a reference have been removed from the graph, we can compute the distance by finding the length of a chain, excluding cycles, to that reference from the generator. Letting τ_p be the maximum distance, each partition requires $r_p = \tau_p + 1$ registers or temporary variables: one register for the value generated on the current iteration and one register for each of the τ_p values that were generated previously and need to flow to the last sink. In the previous example, `B(I)` is the oldest reference and is the generator of the value used at both references to `B(I-1)`. The threshold of the partition is 1. This requires 2 registers because there are two values generated by `B(I)` that are live after executing statement `5` and before executing statement `6`.

For invariant references, we can use one register for scalar replacement because each reference accesses the same value on every iteration. In order for an invariant potential generator to be a partition generator, it must be a definition or the first load of a value along a path through the loop.

2.2.8 Register-Pressure Moderation

Unfortunately, experiments have shown that exposing all of the reuse possible with scalar replacement may result in a performance degradation [CCK90]. Register spilling can completely counteract any savings from scalar replacement. The problem is that specialized information is necessary to recover the original code to prevent excessive spilling. As a result, we need to generate scalar temporaries in such a way as to minimize register pressure, in effect doing part of the register allocator's job.

Our goal is to allocate temporary variables so that we can eliminate the most memory accesses given the available number of machine registers. This can be approximated using a greedy algorithm. In this approximation scheme, the partitions are ordered in decreasing order based upon the ratio of benefit to cost, or in this case, the ratio of memory accesses saved to registers required. At each step, the algorithm chooses the first partition that fits into the remaining registers. This method requires $O(n \log n)$ time to sort the ratios and $O(n)$ time to select the partitions. Hence, the total running time is $O(n \log n)$.

To compute the benefit of scalar replacing a partition, we begin by computing the probability that each basic block will be executed. The first basic block in the loop is assigned a probability of execution of 1. Each outgoing edge from the basic block is given a proportional probability of execution. In the case of two outgoing edges, each edge is given a 50% probability of execution. For the remaining blocks, this procedure is repeated, except that the probability of execution of a remaining block is the sum of the probabilities of its incoming edges. Next, we compute the probability that the generator for a partition is available at the entrance to and exit from a basic block. The probability upon entry is the sum of the probabilities at the exit of a block's predecessors weighted by the number of incoming edges. Upon exit from a block, the probability is 1 if the generator is available, 0 if it is not available and the entry probability otherwise. After computing each of these probabilities, the benefit for each reference within a partition can be computed by multiplying the execution probability for the basic block that contains the reference by the availability probability of the references generator. Figure 2.6 gives the complete algorithm for computing benefit. As an example of benefit, in the loop,

```
   DO 1 I = 1,N
      IF (M(I) .LT. 0) THEN
         A(I) = B(I) + C(I)
      ENDIF
 1    D(I) = A(I) + E(I)
```

the load of `A(I)` in statement `1` has a benefit of 0.5.

Unfortunately, the greedy approximation can produce suboptimal results. To see this, consider the following example, in which each partition is represented by a pair (n, m), where n is the number of registers needed and m is the number of memory accesses eliminated. Assume that we have a machine with 6 registers

```
Procedure CalculateBenefit($P$, $FG$)

Input: $P$ = set of reference partitions
       $FG$ = flow graph
Defs: $A_e^p(\text{B})$ = probability $\gamma_p$ is available on entry to B
      $A_x^p(\text{B})$ = probability $\gamma_p$ is available on exit from B
      $b_p$ = benefit for partition $p$
      $E_{FG}$ = edges in flow graph
      $\mathcal{P}(\text{B})$ = probability block B will be executed

 $\mathcal{P}(\text{ENTRY}) = 1$
 $n =\#$ outgoing forward edges of ENTRY
 weight each outgoing forward edge of ENTRY at $\frac{\mathcal{P}(\text{ENTRY})}{n}$
 for the remaining basic blocks B$\in FG$ in reverse
     depth-first order
   let $\mathcal{P}(\text{B})$= sum of all incoming edge weights
   $n =\#$ outgoing forward edges of B
   weight each outgoing forward edge at $\frac{\mathcal{P}(\text{B})}{n}$
 for each $p \in P$
   for each basic block B$\in FG$ in reverse depth-first order
     $n =\#$ incoming forward edges of B
     for each incoming forward edge of B, $e = (\text{C},\text{B})$
       $A_e^p(\text{B}) = A_e^p(\text{B}) + \frac{A_x^p(\text{C})}{n}$
     if $\gamma_p \in$ LIAVOUT(B) $\bigcup$ LCAVOUT(B) then
       $A_x^p(\text{B}) = 1$
     else if $\gamma_p \in$ LIPAVOUT(B) $\bigcup$ LCPAVOUT(B) then
       $A_x^p(\text{B}) = A_e^p(\text{B})$
     else
       $A_x^p(\text{B}) = 0$
   for each $v \in p$
     if $\gamma_p$ is LCPAV or LCPAVIF1 then
       $b_p = b_p + A_x^p(\text{EXIT}) \times \mathcal{P}(\text{B}_v)$
     else if $\gamma_p$ is LIPAV
       $b_p = b_p + A_e^p(\text{B}) \times \mathcal{P}(\text{B}_v)$
     else
       $b_p = b_p + \mathcal{P}(\text{B}_v)$
end
```

Figure 2.6 CalculateBenefit

and that we are generating temporaries for a loop that has the following generators: $(4, 8), (3, 5), (3, 5), (2, 1)$. The greedy algorithm would first choose $(4, 8)$ and then $(2, 1)$, resulting in the elimination of nine accesses instead of the optimal ten.

To get a possibly better allocation, we can model register-pressure moderation as a knapsack problem, where the number of scalar temporaries required for a partition is the size of the object to be put in the knapsack, and the size of the register file is the knapsack size. Using dynamic programming, an optimal solution to the knapsack problem can be found in $O(kn)$ time, where k is the number of registers available for allocation and n is the number of generators. Hence, for a specific machine, we get a linear time bound. However, with the greedy method, we have a running time that is independent of machine architecture, making it more practical for use in a general tool. The greedy approximation of the knapsack problem is also provably no more than two times worse than the optimal solution [GJ79]. However, our experiments suggest that in practice the greedy algorithm performs as well as the knapsack algorithm.

After determining which generators will be fully scalar replaced, there may still be a few registers available. Those partitions that were eliminated from consideration can be examined to see if partial allocation is possible. In each eliminated partition whose generator is not LCPAV, we allocate references whose distance from γ_p is less than the number of remaining registers. All references within p that do not fit this criterium are removed from p. This step is performed on each partition, if possible, while registers remain unused. Finally, since we have possibly removed references from a partition, anticipability analysis for potential generators must be redone.

To illustrate partial allocation, assume that in the following loop there is one register available.

```
      DO 10 I = 1,N
         A(I) = ...
         IF (B(I) .GT. 0.0) THEN
            ... = A(I)
         ENDIF
10       ... = A(I-1)
```

Here, full allocation is not possible, but there is a loop-independent dependence between the `A(I)`'s. In partial allocation, `A(I-1)` is removed from the partition allowing scalar replacement to be performed. The algorithm in Figure 2.7 gives the complete register-pressure moderation algorithm, including partial allocation.

Although this method of partial allocation may still leave possible reuses not scalar replaced, experience suggests this rarely, if ever, happens. One possible solution is to consider dependences from intermediate points within a partition when looking for potential reuse.

2.2.9 Reference Replacement

At this point, we have determined which references will be scalar replaced. We now move into the code generation phase of the algorithm. Here, we will replace array references with temporary variables and ensure that the temporaries contain the proper value at a given point in the loop.

After we have determined which partitions will be scalar replaced, we replace the array references within each partition. This algorithm is shown in Figure 2.8. First, for each variant partition p, we create the temporary variables $T_p^0, T_p^1, \ldots, T_p^{r_p-1}$, where T_p^i represents the value generated by $g \in \gamma_p$ i iterations earlier, where g is the first generator to access the value used throughout the partition. Each reference within the partition is replaced with the temporary that coincides with its distance from g. For invariant partitions, each reference is replaced with T_p^0. If a replaced reference $v \in \gamma_p$ is a memory load, then a statement of the form $T_p^i = v$ is inserted before the generating statement. Requiring that the load must be in γ_p for load insertion ensures that a load that is a potential generator but also has a potential generator itself will not have a load inserted. The value for the potential generator not in γ_p will already be provided by its potential generator. If the v is a store, then a statement of the form $v = T_p^i$ is inserted after the generating statement. The latter assignment is unnecessary if it has an outgoing loop-independent edge to definition that is always executed and it has no outgoing inconsistent true dependences. We could get better results by performing availability and anticipability analysis exclusively for definitions to determine if a value is always redefined.

```
Procedure ModerateRegisterPressure($P$, $G$)

Input: $P$ = set of reference partitions
       $G = (V, E)$, the dependence graph

Defs: $b_p$ = benefit for a partition $p$
      $r_p$ = register required by a partition $p$

  CalculateBenefit($P$, $G$)
  $\mathcal{R}$ = registers available
  $\mathcal{H}$ = sort of $P$ on ratio of $\frac{b_p}{r_p}$ in decreasing order
  for $i = 1$ to $|\mathcal{H}|$ do
    if $r_{\mathcal{H}_i} < \mathcal{R}$ then
      allocate($\mathcal{H}_i$)
      $\mathcal{R} = \mathcal{R} - r_{\mathcal{H}_i}$
    else
      $\mathcal{S} = \mathcal{S} \cup \mathcal{H}_i$
      $P = P - \mathcal{H}_i$
    endif
  enddo
  if $\mathcal{R} > 0$ then
    $i = 1$
    while $i < |\mathcal{S}|$ and $\mathcal{R} > 0$ do
      while $|\mathcal{S}_i| > 0$ and $\mathcal{R} > 0$ do
        $\mathcal{S}_i = \{v | v \in \mathcal{S}_i, d_n(\hat{e}) < \mathcal{R}, \hat{e} = (\gamma_{\mathcal{S}_i}, v)\}$
        allocate($\mathcal{S}_i$)
        $\mathcal{R} = \mathcal{R} - r_{\mathcal{S}_i}$
        $P = P \cup \mathcal{S}_i$
      enddo
    enddo
    redo anticipability analysis
  endif
end
```

Figure 2.7 ModerateRegisterPressure

```
Procedure ReplaceReferences($P$)

Input: $P$ = set of reference partitions

  for each $p \in P$ do
    let $g = v \in \gamma_p$ with no incoming marked or unmarked
      edge from another $w \in \gamma_p$ or $v$ is invariant
    for each $v \in P$ do
      if $v$ is invariant then $d = 0$
      else $d$ = distance from $g$ to $v$
      replace $v$ with $T_p^d$
      if $v \in \gamma_p$ and $v$ is a load then
        insert "$T_p^d = v$" before $v$'s statement
      if $v$ is a store $\wedge$ ($\exists e = (v, w) | e$ is true and inconsistent $\vee$
          $\forall e = (v, w)$ where $e$ is output and $\tau_e = 0$,
          $w$ is not always executed) then
        insert "$v = T_p^d$" after $v$'s statement
end
```

Figure 2.8 ReplaceReferences

The effect of reference replacement will be illustrated on the following loop nest.

```
      DO 3 I = 1, 100
 1       IF (M(I) .LT. 0) E(I) = C(I)
 2       A(I) = C(I) + D(I)
 3       B(K) = B(K) + A(I-1)
```

The reuse-generating dependences are:

1. A loop-independent input dependence from `C(I)` in statement 1 to `C(I)` in statement 2 (threshold 0), and
2. A true dependence from `A(I)` to `A(I-1)` (threshold 1).
3. A true dependence from `B(K)` to `B(K)` in statement 2 (threshold 1).

By our method, the generator `C(I)` in statement 1 needs only one temporary, `T10`. Here, we are using the first numeric digit to indicate the number of the partition and the second to represent the distance from the generator. The generator `B(K)` in statement 1 needs one temporary, `T20`, since it is invariant, and the generator `A(I)` needs two temporaries, `T30` and `T31`. When we apply the reference replacement procedure to the example loop, we generate the following code.

```
      DO 3 I = 1, 100
 1       IF (M(I) .LT. 0) THEN
            T10  = C(I)
            E(I) = T10
         ENDIF
         T30  = T10 + D(I)
 2       A(I) = T30
         T20  = T20 + T31
 3       B(K) = T20
```

The value for `T31` is not generated in this example. We will discuss its generation in Sections 2.2.11 and 2.2.13.

2.2.10 Statement-Insertion Analysis

After we replace the array reference with scalar temporaries, we need to insert loads for partially available generators. Given a reference that has a partially available potential generator, we need to insert a statement at the highest point on a path from the loop entrance to the reference that anticipates the generator where the generator is not partially available and is anticipated. By performing statement-insertion analysis on potential generators, we guarantee that every reference's anticipated value will be fully available. Here, we handle each individual reference, whereas name partitioning linked together those references that share values. This philosophy will not necessarily introduce a minimum number of newly inserted loads, but there will not be an increase in the number of run-time loads. The place for insertion of loads for partially available generators can be determined using Drechsler and Stadel's formulation for partial redundancy elimination, as shown below [DS88].

$$\begin{aligned}
\text{PPIN}(\text{B}) &= \text{ANTIN}(\text{B}) \bigcap \text{PAVIN}(\text{B}) \bigcap (\text{ANTLOC}(\text{B}) \bigcup (\text{TRANSP}(\text{B}) \bigcap \text{PPOUT}(\text{B})) \\
\text{PPOUT}(\text{B}) &= \begin{cases} \text{FALSE if B is the loop exit} \\ \bigcap_{s \in \text{SUCC}(\text{B})} \text{PPIN}(s) \end{cases} \\
\text{INSERT}(\text{B}) &= \text{PPOUT}(\text{B}) \bigcap \neg\text{AVOUT}(\text{B}) \bigcap (\neg\text{PPIN}(\text{B}) \bigcup \neg\text{TRANSP}(\text{B})) \\
\text{INSERT}_{(\text{A},\text{B})} &= \text{PPIN}(\text{B}) \bigcap \neg\text{AVOUT}(\text{A}) \bigcap \neg\text{PPOUT}(\text{A})
\end{aligned}$$

Here, PPIN(B) denotes placement of a statement is possible at the entry to a block and PPOUT(B) denotes placement of a statement is possible at the exit from a block. INSERT(B) determines which loads need to be inserted at the bottom of block B. $\text{INSERT}_{(\text{A},\text{B})}$ is defined for each edge in the control-flow graph and determines which loads are inserted on the edge from A to B. TRANSP(B) is true for some array expression if it is not defined by a consistent or inconsistent definition in the block B. ANTLOC(B) is the same as GEN(B) for anticipability information. Three problems of the above form are solved: one for LIPAV generators, one for LCPAV generator and one for LCPAVIF1 generators. Additionally, any reference to loop-carried ANTIN information refers to the entry block.

If $\text{INSERT}_{(\text{A},\text{B})}$ is true for some potential generator g, then we insert a load on the edge (A,B) of the form $T_p^d = g$, where T_p^d is the temporary name associated with g. If INSERT(B) is true for some potential generator g, then a statement of identical form is inserted at the end of block B. Finally, if $\text{INSERT}_{(\text{A},\text{B})}$ is true $\forall \text{A} \in \text{PRED}(\text{B})$, then loads can be collapsed into the beginning of block B. The algorithm for inserting statements is shown in Figure 2.9.

If we perform statement insertion on our example loop, we get the following results.

```
      DO 3 I = 1, 100
1        IF (M(I) .LT. 0) THEN
            T10  = C(I)
            E(I) = T10
         ELSE
            T10  = C(I)
         ENDIF
         T30  = T10 + D(I)
2        A(I) = T30
         T20  = T20 + T31
3        B(K) = T20
```

Again, the generation of T31 is left for Sections 2.2.11 and 2.2.13.

2.2.11 Register Copying

Next, we need to ensure that the values held in the temporary variables are correct across loop iterations. The value held in T_p^i needs to move one iteration further away from its generator, T_p^0, on each subsequent loop iteration. Since i is the number of loop iterations the value is from the generator, the variable T_p^{i+1} needs to take on the value of T_p^i at the end of the loop body in preparation for the iteration change. The algorithm in Figure 2.10 effects the following shift of values for each partition, p.

$$\begin{aligned}
T_p^{r_p-1} &= T_p^{r_p-2} \\
T_p^{r_p-2} &= T_p^{r_p-3} \\
&\dots \\
T_p^1 &= T_p^0
\end{aligned}$$

```
Procedure InsertStatements($FG$)

Input: $FG$ = flow graph

  perform insert analysis
  for each flow edge $e \in FG$
    for each $v \in$ INSERT$_e$
      if $v$ is invariant then d = 0
      else $d$ = distance from the oldest generator in $P(v)$ to $v$
      insert statement "$T^d_{P(v)} = v$" on $e$
  for each basic block B $\in FG$
    for each $v \in$ INSERT(B)
      if $v$ is invariant then d = 0
      else $d$ = distance from the oldest generator in $P(v)$ to $v$
      insert statement "$T^d_{P(v)} = v$" at the end of B
    for each $v$ such that INSERT$_{(A,B)}(v)$ is true
        $\forall$A $\in$ PREDS(B)
      collapse loads of $v$ into beginning of B
end
```

Figure 2.9 InsertStatements

After inserting register copies, our example code becomes:

```
      DO 3 I = 1, 100
1        IF (M(I) .LT. 0) THEN
            T10  = C(I)
            E(I) = T10
         ELSE
            T10  = C(I)
         ENDIF
         T30  = T10 + D(I)
2        A(I) = T30
         T20  = T20 + T31
         B(K) = T20
3        T31  = T30
```

2.2.12 Code Motion

It may be the case that the assignment to or a load from a generator may be moved entirely out of the innermost loop. This is possible when the reference to the generator is invariant with respect to the innermost loop. In the example above, `B(K)` does not change with each iteration of the `I`-loop; therefore, its value can be kept in a register during the entire execution of the loop and stored back into `B(K)` after the loop exit.

```
      DO 3 I = 1, 100
1        IF (M(I) .LT. 0) THEN
            T10  = C(I)
            E(I) = T10
         ELSE
            T10  = C(I)
         ENDIF
         T30  = T10 + D(I)
2        A(I) = T30
         T20  = T20 + T31
3        T31  = T30
      B(K) = T20
```

The algorithm for this type of code motion is shown in Figure 2.11.

```
Procedure InsertRegisterCopies(G,P,x)

Input: $G = (V, E)$, the dependence graph
       $P$ = set of reference partitions
       $x$ = peeled iteration number or
            $\infty$ for a loop body

  for each $p \in P$
    for $i = \min(r_p - 1, x)$ to 1 do
      insert "$T_p^i = T_p^{i-1}$" at the end of loop body in $G$
end
```

Figure 2.10 InsertRegisterCopies

When inconsistent dependences leave an invariant array reference that is a store, the generating store for that variable cannot be moved outside of the innermost loop. Consider the following example.

```
      DO 10 J = 1, N
   10    A(I) = A(I) + A(J)
```

The true dependence from `A(I)` to `A(J)` is not consistent. If the value of `A(I)` were stored into `A(I)` outside of the loop, then the value of `A(J)` would be wrong whenever `I=J` and `I` $>$ `1`.

2.2.13 Initialization

To ensure that the temporary variables contain the correct values upon entry to the loop, it is peeled using the algorithm in Figure 2.12. We peel $\max(r_{p_1}, \ldots, r_{p_n}) - 1$ iterations from the beginning of the loop, replacing the members of a variant partition p for peeled iteration k with their original array reference, substituting the iteration value for the induction variable, only if $j \geq k$ for a temporary T_p^j. For invariant partitions, we only replace non-partition generators' temporaries on the first peeled iteration. Additionally, we let $r_p = 2$ for each invariant partition when calculating the number of peeled iterations. This ensures that invariant partitions will be initialized correctly. Finally, at the end of each peeled iteration, the appropriate number of register transfers is inserted.

When this transformation is applied to our example, we get the following code.

```
IF (M(1) .LT. 0) THEN
   T10  = C(1)
   E(1) = T10
ELSE
   T10  = C(1)
ENDIF
T30  = T10 + D(1)
A(1) = T30
T20  = B(K) + A(0)
T31  = T30
    ...
LOOP BODY
```

2.2.14 Register Subsumption

In our example loop, we have eliminated three loads and one store from each iteration of the loop, at the cost of three register-to-register transfers in each iteration. Fortunately, inserted transfer instructions can be eliminated if we unroll the scalar replaced loop using the algorithm in Figure 2.13. If we have the partitions $p_0, p_1, \ldots, p_n$, we can remove the transfers by unrolling $\text{lcm}(r_{p_0}, r_{p_1}, \ldots, r_{p_n}) - 1$ times. In the kth unrolled

```
Procedure CodeMotion($P$)

Input: $P$ = set of reference partitions

  for each $p \in P$
    for each $v \in \gamma_p$
      if $v$ is a store $\wedge v \in$ LCANTIN(ENTRY) then
        if $\exists e = (v, w) \in E \wedge v = w \wedge$
            $\forall e = (v, z) \in E$, $e$ is consistent then
          move the store into $v$ after the loop
      else if $\exists e = (v, w) \in E$ such that $e$ is consistent $\wedge$
          $v = w \wedge$ $\forall e = (u, v) \in E$, if $e$ is true it is consistent
          move the load into $v$ before the loop
end
```

Figure 2.11 CodeMotion

```
Procedure Initialize(P,G)

Input: $P$ = set of reference partitions
        $G = (V, E)$, the dependence graph

  $x = \max(r_{p_1}, \ldots, r_{p_2}) - 1$
  for $k = 1$ to $x$
    $G'$ =peel of the $k$th iteration of $G$
    for each $v \in V'$
      if $v = T_p^j \wedge (v$ is variant $\wedge j \geq k) \vee$
          ($v$ is invariant $\wedge v \notin \gamma_{P(v)} \wedge j + 1 \geq k \wedge$
          $v$'s generator is loop carried) then
        replace $v$ with its original array reference
        replace the inner loop induction variable with
          its $k$th iteration
      endif
    InsertRegisterCopies($G'$, $P$, $k$)
end
```

Figure 2.12 Initialize

body, the temporary variable T_p^j is replaced with the variable $T_p^{\mathrm{mod}(j-k,r_p)}$ where $\mathrm{mod}(y,x) = y - \lfloor \frac{y}{x} \rfloor x$. Essentially we capture the permutation of values by circulating the register names within the unrolled iterations.

The final result of scalar replacement on our example is shown in Figure 2.14 (the pre-loop to capture the extra iterations and the initialization code are not shown).

2.3 Experiment

We have implemented a source-to-source translator in the Parascope programming environment, a programming system for Fortran, that uses the dependence analyzer from PFC. The translator replaces subscripted variables with scalars using the described algorithm. The experimental design is illustrated in Figure 2.15. In this scheme, Parascope serves as a preprocessor, rewriting Fortran programs to improve register allocation. Both the original and transformed versions of the program are then compiled and run using the standard product compiler for the target machine.

For our test machine, we chose the IBM RS/6000 model 540 because it had a good compiler and a large number of floating-point registers (32). In fact, the IBM XLF compiler performs scalar replacement for those references that do not require dependence analysis. Many fully available loop-independent cases and invariant cases are handled. Therefore, the results described here only reflect the cases that required dependence analysis. Essentially, we show the results of performing scalar replacement on loop-carried dependences and in the presence of inconsistent dependences.

Livermore Loops. We tested scalar replacement on a number of the Livermore Loops. Some of the kernels did not contain opportunities for our algorithm; therefore, we do not show their results. In the table below, we show the performance gain attained by our transformation system.

Loop	Iterations	Original	Transformed	Speedup
1	10000	3.40s	2.54s	1.34
5	10000	3.05s	1.36s	2.24
6	10000	3.82s	2.82s	1.35
7	5000	3.94s	2.02s	1.95
8	10000	3.38s	3.07s	1.10
11	10000	4.52s	1.69s	2.67
12	10000	1.70s	1.42s	1.20
13	5000	3.25s	3.01s	1.08
18	1000	2.62s	2.54s	1.03
20	500	2.99s	2.90s	1.03
23	5000	2.68s	2.35s	1.14

Some of the interesting results include the performances of loops 5 and 11, which compute first order linear recurrences. Livermore Loop 5 is shown below.

```
   DO 5 I = 2,N
5     X(I) = Z(I) * (Y(I) - X(I-1))
```

Here, scalar replacement not only removes one memory access, but also improves pipeline performance. The store to `X(I)` will no longer cause the load of `X(I-1)` on the next iteration to block. Loop 11 presents a similar situation.

Loop 6, shown below, is also an interesting case because it involves an invariant array reference that requires dependence analysis to detect.

```
      DO 6 I= 2,N
        DO 6 K= 1,I-1
6         W(I)= W(I)+B(I,K)*W(I-K)
```

A compiler that recognizes loop invariant addresses to get this case, such as the IBM compiler, fails because of the load of `W(I-K)`. Through the use of dependence analysis, we are able to prove that there is no dependence between `W(I)` and `W(I-K)` that is carried by the innermost loop, allowing code motion. Because of this additional information, we are able to get a speedup of 1.35.

```
Procedure Subsume(G,P)

Input: $P$ = set of reference partitions
       $G = (V, E)$, the dependence graph

  $x$ =lcm$(r_{p_0}, \ldots, r_{p_n}) - 1$
  unroll $G'$ $x$ times
  for $G_k$ = each of the $x$ new loop bodies
    for each $v \in V_k$
      if $v = T_p^j$ then
        replace $v$ with $T_p^{\mathrm{mod}(j-k, r_p)}$
end
```

Figure 2.13 Subsume

```
  DO 3 I = 2, 100,2
1    IF (M(I) .LT. 0) THEN
        T10    = C(I)
        E(I)   = T10
     ELSE
        T10    = C(I)
     ENDIF
     T30  = T10 + D(I)
2    A(I) = T30
     T20    = T20  + T31
     IF (M(I+1) .LT. 0) THEN
        T10    = C(I+1)
        E(I+1)   = T10
     ELSE
        T20    = C(I+1)
     ENDIF
     T31  = T10 + D(I+1)
     A(I+1) = T31
3    T20    = T20  + T30
```

Figure 2.14 Example After Scalar Replacement

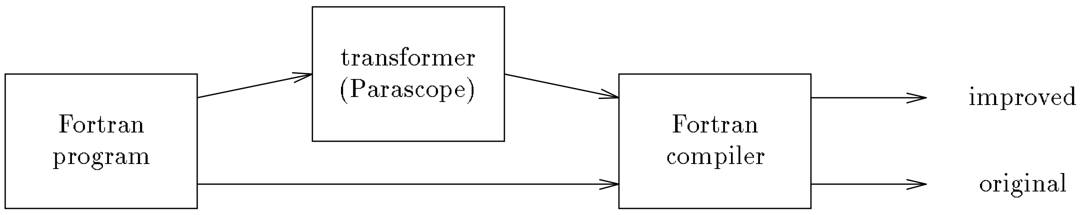

Figure 2.15 Experimental design.

Linear Algebra Kernels. We also tested scalar replacement on both the point and block versions of LU decomposition with and without partial pivoting. In the table below, we show the results.

Kernel	Original	Transformed	Speedup
LU Decomp	6.76s	6.09s	1.11
Block LU	6.39s	4.40s	1.45
LU w/ Pivot	7.01s	6.35s	1.10
Block LU w/ Pivot	6.84s	4.81s	1.42

Each of these kernels contains invariant array references that require dependence analysis to detect. The speedup achieved on the block algorithms is higher because an invariant load and store are removed rather than just a load as in the point algorithms.

Applications. To complete our study we ran a number of Fortran applications through our translator. We chose programs from Spec, Perfect, RiCEPS and local sources. Of those programs that belong to the benchmark suites, but are not included in the experiment, 5 failed to be successfully analyzed by PFC, 1 failed to compile on the RS6000 and 10 contained no opportunities for our algorithms. Table 3 contains a short description of each application.

Suite	Application	Description
SPEC	Matrix300	Matrix Multiplication
	Tomcatv	Mesh Generation
Perfect	Adm	Pseudospectral Air Pollution
	Arc2d	2d Fluid-Flow Solver
	Flo52	Transonic Inviscid Flow
RiCEPS	Onedim	Time-Independent Schrödinger Equation
	Shal	Weather Prediction
	Simple	2d Hydrodynamics
	Sphot	Particle Transport
	Wave	Electromagnetic Particle Simulation
Local	CoOpt	Oil Exploration
	Seval	B-Spline Evaluation
	Sor	Successive Over-Relaxation

The results of performing scalar replacement on these applications is reported in the following table. Any application not listed observed a speedup of 1.00.

Suite	Program	Original	Transformed	Speedup
Perfect	Adm	236.84s	228.84s	1.03
	Arc2d	410.13s	407.57s	1.01
	Flo52	66.32s	63.83s	1.04
RiCEPS	Shal	302.03s	290.42s	1.04
	Simple	963.20s	934.13s	1.03
	Sphot	3.85s	3.78s	1.02
	Wave	445.94s	431.11s	1.03
Local	CoOpt	122.88s	120.44s	1.02
	Seval	0.62s	0.56s	1.11
	Sor	1.83s	1.26s	1.46

The applications Sor and Seval performed the best because we were able to optimize their respective computationally intensive loop. Each had one loop which comprised almost the entire running time of the program. For the program Simple, one loop comprised approximately 50% of the program execution time, but the carried dependence could not be exposed due to a lack of registers. In fact, without the register-pressure minimization algorithm of Section 2.2.8, program performance deteriorated. Sphot's improvement was gained by performing scalar replacement on one partition in one loop. This particular partition contained a loop-independent partially available generator that required our extension to handle control flow.

The IBM RS/6000 has a load penalty of only 1 cycle. On processors with larger load penalties, such as the DEC Alpha, we would expect to see a larger performance gain through scalar replacement. Additionally, the problem sizes for benchmark are typically small. On larger problem sizes, we expect to see larger performance gains due to a higher percentage of time being spent inside of loops.

2.4 Summary

In this chapter, we have presented an algorithm to perform scalar replacement in the presence of forward conditional-control flow. By mapping partial redundancy elimination to scalar replacement, we are able to ensure that we will not increase the run-time memory costs of a loop. We have applied our algorithm to a number of kernels and whole applications and shown that integer-factor speedups over good optimizing compilers are possible on kernels.

Chapter 3

Unroll-And-Jam

Because applying scalar replacement alone may still leave a loop memory bound, we can sometimes apply unroll-and-jam before scalar replacement to allow a larger reduction in loop balance. For example, recall from Chapter 1 that given the loop

```
      DO 10 I = 1, 2*M
        DO 10 J = 1, N
10        A(I) = A(I) + B(J)
```

with a balance of 1, unroll-and-jam of the `I`-loop by 1 produces the loop

```
      DO 10 I = 1, 2*M, 2
        DO 10 J = 1, N
          A(I) = A(I) + B(J)
10        A(I+1) = A(I+1) + B(J)
```

with a balance of 0.5. On a machine that can perform 2 flops per load for a balance of 0.5, the transformed loop would execute faster because it would be balanced.

Although unroll-and-jam has been studied extensively, it has not been shown how to tailor unroll-and-jam to specific loops run on specific architectures. In the past, unroll amounts have been determined experimentally and specified with a compile-time parameter [CCK90]. However, the best choice for unroll amounts varies between loops and architectures. Therefore, in this chapter, we derive a method to chose unroll amounts automatically in order to balance program loops with respect to a specific target architecture. Unroll-and-jam is tailored to a specific machine based on a few parameters of the architecture, such as effective number of registers and machine balance. The result is a machine-independent transformation system in the sense that it can be retargeted to new processors by changing parameters.

This chapter begins with an overview of the safety conditions for unroll-and-jam. Then, we present a method for updating the dependence graph so that scalar replacement can take advantage of the new opportunities created by unroll-and-jam. Next, we derive an automatic method to balance program loops with respect to a particular architecture and, finally, we report on performance improvements achieved by an experimental implementation of this algorithm.

3.1 Safety

Certain dependences prevent or limit the amount of unrolling that can be done and still allow jamming to be safe. Consider the following loop.

```
     DO 10 I = 1, 2*M
        DO 10 J = 1,N
10         A(I,J) = A(I-1,J+1)
```

A true dependence with a distance vector of $\langle 1, -1 \rangle$ goes from `A(I,J)` to `A(I-1,J+1)`. Unrolling the outer loop once creates a loop-independent true dependence that becomes an antidependence in the reverse direction after loop jamming. This dependence prevents jamming because the order of the store to and load from location would be reversed as shown in the unrolled code below.

```
      DO 10 I = 1, 2*M,2
         DO 10 J = 1,N
            A(I,J) = A(I-1,J+1)
 10         A(I+1,J) = A(I,J+1)
```

Location `A(3,2)` is defined on iteration $\langle 3,2\rangle$ and read on iteration $\langle 4,1\rangle$ in the original loop. In the transformed loop, the location is read on iteration $\langle 3,1\rangle$ and defined on iteration $\langle 3,2\rangle$, reversing the access order and changing the semantics of the loop. With a semantics that is only defined on correct programs (as in Fortran), we say that unroll-and-jam is safe if the flow of values is not changed. The following theorem summarizes this property.

Theorem 3.1 Let e be a true, output or antidependence carried at level k with distance vector

$$d = \langle 0,\ldots,0,d_k,0,\ldots,0,d_j,\ldots\rangle,$$

where $d_j < 0$ and all components between the kth and jth are 0. Then the loop at level k can be unrolled at most $d_k - 1$ time before a dependence is generated that prevents fusion of the inner $n-k$ loops.

Proof See Callahan, et. al [CCK88]. □

Essentially, unrolling more than $d_k - 1$ will introduce a dependence with a negative entry in the outermost position after fusion. Since negative thresholds are not defined, the dependence direction must be reversed. Dependence direction reversal makes unroll-and-jam unsafe. Additionally, when non-`DO` statements appear at levels other than the innermost level, loop distribution must be safe in order for unroll-and-jam to be safe. Essentially, no recurrence involving data or control dependences can be violated [AK87].

3.2 Dependence Copying

To allow scalar replacement to take advantage of the new opportunities for reuse created by unroll-and-jam, we must update the dependence graph to reflect these changes. Below is a method to compute the updated dependence graph after loop unrolling for consistent dependences that contain only one loop induction variable in each subscript position and are not invariant with respect to the unrolled loop. [CCK88].

If we unroll the mth loop in a nest L by a factor of k, then the updated dependence graph $G^{'} = (V^{'},E^{'})$ for $L^{'}$ can be computed from the dependence graph $G = (V,E)$ for L by the following rules:

1. For each $v \in V$, there are $k+1$ nodes $v_0,\ldots,v_k$ in $V^{'}$. These correspond to the original reference and its k copies.
2. For each edge $e = \langle v,w\rangle \in E$, with distance vector $d(e)$, there are $k+1$ edges $e_0,\ldots,e_k$ where $e_j = \langle v_j,w_i\rangle$, v_j is the jth copy of v, w_i is the ith copy of w and

$$\begin{aligned} i &= (j + d_m(e)) \bmod (k+1) \\ d_m(e_j) &= \begin{cases} \lfloor \frac{d_m(e)}{k+1} \rfloor & \text{if } i \geq j \\ \lfloor \frac{d_m(e)}{k+1} \rfloor + 1 & \text{if } i < j \end{cases} \end{aligned}$$

Below dependence copying is described for invariant references. To aid in the explanation, the following terms are defined.

1. If references are invariant with respect to a particular loop, we call that loop *reference invariant.*
2. If a specific reference, v, is invariant with respect to a loop, we call that loop *v-invariant.*

The behavior of invariant references differs from that of variant references when unroll-and-jam is applied. Given an invariant reference v, the distance vector value of an incoming edge corresponding to a v-invariant loop represents multiple values rather than just one value. Therefore, we first apply the previous formula to the minimum distance. Then, we create a copy of the original edge for each new loop body between the source and sink corresponding the original references and insert loop-independent dependences from the references within a statement to all identical references following them.

To extend dependence copying to unroll-and-jam, we have the following rules using an outermost- to innermost-loop application ordering. If

- we unroll loop k
- we have a new distance vector, $d(e) = \langle d_1, d_2, \ldots, d_n\rangle$
- $d_k(e) = 0$
- $\forall m < k, d_m(e) = 0$

then the next inner non-zero entry, $d_p(e), p > k$, becomes the threshold of e and p becomes the carrier of e. If $\forall m, d_m(e) = 0$, then e is loop independent. If $\exists m \leq k$ such that $d_m(e) > 0$ then e remains carried by loop m with the same threshold.

The safety rules of unroll-and-jam prevent the exposure of a negative distance vector entry in the outermost non-zero position for true, output and antidependences. However, the same is not true for input dependences. The order in which reads are performed does not change the semantics of the loop. Therefore, if the outermost entry of an input dependence distance vector becomes negative during unroll-and-jam, we change the direction of the dependence and negate the entries in the vector.

To demonstrate dependence copying, consider Figure 3.1. The original edge from `A(I,J)` to `A(I,J-2)` has a distance vector of $\langle 2, 0\rangle$ before unroll-and-jam. After unroll-and-jam the edge from `A(I,J)` now goes to `A(I,J)` in statement `10` and has a vector of $\langle 0, 0\rangle$. An edge from `A(I,J+1)` to `A(I,J-2)` with a vector of $\langle 1, 0\rangle$ and an edge from `A(I,J+2)` to `A(I,J-1)` with a vector of $\langle 1, 0\rangle$ have also been created. For the first edge, the first formula for $d_m(e_j)$ applies. For the latter two edges, the second formula applies.

3.3 Improving Balance with Unroll-and-Jam

As stated earlier, we would like to convert loops that are bound by main-memory access time into loops that are balanced. Since unroll-and-jam can introduce more floating-point operations without a proportional increase in memory accesses, it has the potential to improve loop balance. Although unroll-and-jam improves balance, using it carelessly can be counterproductive. Transformed loops that spill floating-point registers or whose object code is larger than the instruction cache may suffer performance degradation. Since the size of the object code is compiler dependent, we are forced to assume that the instruction cache is large enough to hold the unrolled loop body. By doing so, we can remain relatively independent of the details of a particular software system. This assumption proved valid on the IBM RS/6000 and leaves us with the following objectives for unroll-and-jam.

1. Balance a loop with a particular architecture.
2. Control register pressure.

```
      DO 10 J = 1,N             |       DO 10 J = 1,N,3
        DO 10 I = 1,N           |         DO 10 I = 1,N
10        A(I,J) = A(I,J-2)     |           A(I,J)   = A(I,J-2)
                                |           A(I,J+1) = A(I,J-1)
                                |  10       A(I,J+2) = A(I,J)
```

Figure 3.1 Distance Vectors Before and After Unroll-and-Jam

Expressing these goals as an integer optimization problem, we have

objective function: $\min |\beta_L - \beta_M|$
constraint: # floating-point registers required $\leq$ register-set size

where the variables in the problem are the unroll amounts for each of the loops in a loop nest. For the solution to the objective function, we prefer a slightly negative difference over a slightly positive difference.*

For each loop nest within a program, we model its possible transformation as a problem of this form. Solving it will give us the unroll amounts to balance the loop nest as much as possible. Using the following definitions throughout our derivation, we will show how to construct and efficiently solve a balance-optimization problem at compile time.

V	$=$	array references in an innermost loop
V_r	$=$	members of V that are memory reads
V_w	$=$	members of V that are memory writes
X_i	$=$	number of times the ith outermost loop in L is unrolled + 1
		(This is the number of loop bodies created by unrolling loop i)
F	$=$	number of floating-point operations after unroll-and-jam
M	$=$	number of memory references after unroll-and-jam

For the purposes of this derivation, we will assume that each loop nest is perfectly nested. We will show how to handle non-perfectly nested loops in Section 3.4.6. Additionally, loops with conditional-control flow in the innermost loop body will not be candidates for unroll-and-jam. Inserting additional control flow within an innermost loop can have disastrous effects upon performance.†

3.3.1 Computing Transformed-Loop Balance.

Since the value of β_m is a compile-time constant defined by the target architecture, we need only create the formula for β_L at compile time to create the objective function. To create β_L, we compute the number of floating-point operations, F, and memory references, M, in one innermost-loop iteration. F is simply the number of floating-point operations in the original loop, f, multiplied by the number of loop bodies created by unroll-and-jam, giving

$$F = f \times \prod_{1 \leq i < n} X_i,$$

where n is the number of loops in a nest. M requires a detailed analysis of the dependence graph. Additionally, in computing M we will assume an infinite register set. The register-pressure constraint in the optimization problem will ensure that any assumption about a memory reference being removed will be true.

To simplify the derivation of our formulation for memory cycles, we initially assume that at most one incoming or outgoing edge is incident upon each $v \in V$ and that all subscript positions contain at most one induction variable. We will remove these restrictions later. Additionally, we partition the reference set of the loop into sets that exhibit different memory behavior when unroll-and-jam is applied. These sets are listed below.

1. $V^{\emptyset}$ = references without an incoming consistent dependence.
2. V_r^C = memory reads that have a loop-carried or loop-independent incoming consistent dependence, but are not invariant with respect to any loop.

*On massively parallel systems, it may be advantageous to have $\beta_L << \beta_M$ to reduce network contention.
†Experiments have shown this to be true on the IBM RS/6000.

3. V_r^I = memory reads that are invariant with respect to a loop.
4. V_w^C = memory writes that have a loop-carried or loop-independent incoming consistent dependence, but are not invariant with respect to any loop.
5. V_w^I = memory writes that are invariant with respect to a loop.

Using this partitioning, we compute the cost associated with each partition, $\{M^\emptyset, M_r^C, M_r^I, M_w^C, M_w^I\}$, to get the total memory cost,

$$M = M^\emptyset + M_r^C + M_r^I + M_w^C + M_w^I.$$

For the first partition, $M^\emptyset$, each copy of the loop body produced by unroll-and-jam contains one memory reference associated with each original reference. This value can be expressed as follows.

$$M^\emptyset = \sum_{v \in V^\emptyset} \Big(\prod_{1 \leq i < n} X_i \Big).$$

For example, in Figure 3.2, unrolling the outer two loops by 1 creates 3 additional copies from the reference to `A(I,J,K)` that will be references to memory.

For each $v \in V_r^C$, unroll-and-jam may create dependences useful in scalar replacement for some of the new references created. In order for a reference, v, to be scalar replaced, its incoming dependence, e_v, must be converted into an innermost-loop-carried or a loop-independent dependence, which we call an *innermost* dependence edge. In terms of the distance vector associated with an edge, $d(e_v) = \langle d_1, d_2, \ldots, d_n \rangle$, the edge is made innermost when it becomes $d(e'_v) = \langle 0, 0, \ldots, 0, d_n \rangle$. After unroll-and-jam, only some of the references will be the sink of an innermost edge. It is our goal to quantify this value. For example, in Figure 3.3, the first copy of `A(I,J-2)`, the reference to `A(I,J-1)`, is the sink of a dependence from `A(I,J+3)` that is not innermost, but the second and third copies, `A(I,J)` and `A(I,J+1)`, are the sinks of innermost edges (in this case, ones that are loop independent) and will be removed by scalar replacement.

To quantify the behavior of V_r^C, we first compute the total number of references before scalar replacement as in $M^\emptyset$,

$$\sum_{v \in V_r^C} \Big(\prod_{1 \leq i < n} X_i \Big).$$

Next, we determine which of the new references will be removed by scalar replacement by quantifying the number of references that will be the sink of an innermost dependence after unroll-and-jam. To simplify the derivation, we consider unrolling only loop i, where $\forall e$ and $\forall k \neq i, n$ we have $d_k(e) = 0$. This limits the dependences to be carried by loop i or to be innermost already. Later, we will extend the formulation to arbitrary loop nests and distance vectors.

Recall from our dependence-copying formulas in Section 3.2 that $d_i(e_v) < X_i$, for some $v \in V_r^C$, must be true in order to create a distance vector with a 0 in the ith position, resulting in an innermost dependence edge. However, depending upon which of the two distance formulas is applied, the edge may still have a 1 in the ith position. In order for the formula that allows an entry to become 0, $d_i(e'_v) = \lfloor \frac{d_i(e_v)}{X_i} \rfloor$, to be applied, the edge created by unroll-and-jam, $e'_v = \langle w_m, v_n \rangle$, must have $n \geq m$ true. If $n < m$, then the applicable formula, $d_i(e'_v) = \lfloor \frac{d_i(e_v)}{X_i} \rfloor + 1$, prevents an entry from becoming 0. In the following theorem, we show that $n \geq m$ is true for $X_i - d_i(e_v)$ of the new dependence edges if $d_i(e_v) < X_i$. This shows that $X_i - d_i(e_v)$ new references can be removed by scalar replacement.

```
           DO 10 K = 1,N            |    DO 10 K = 1,N,2
             DO 10 J = 1,N          |      DO 10 J = 1,N,2
               DO 10 I = 1,N        |        DO 10 I = 1,N
    10           A(I,J,K) = ...     |          A(I,J,K)     = ...
                                    |          A(I,J,K+1)   = ...
                                    |          A(I,J+1,K)   = ...
                                    | 10       A(I,J+1,K+1) = ...
```

Figure 3.2 $V^\emptyset$ Before and After Unroll-and-Jam

Theorem 3.2

Given $0 \leq m, d_i(e) < X_i$ and $n = (m + d_i(e)) \bmod X_i$ for each new edge $e'_v = \langle w_m, v_n \rangle$ created from $e_v = \langle w_0, v_0 \rangle$ by unroll-and-jam, then

$$n \geq m \iff m < X_i - d_i(e).$$

Proof

($\Rightarrow$)

Given $n \geq m$, assume
$m \geq X_i - d_i(e)$.
Since
$0 \leq m, d_i(e) < X_i$
we have
$n = (m + d_i(e)) \bmod X_i$
$\quad = m + d_i(e) - X_i$
giving
$m + d_i(e) - X_i \geq m$.
Since
$d_i(e) < X_i$,
the inequality is violated, leaving
$m < X_i - d_i(e)$.

($\Leftarrow$)

Given
$m < X_i - d_i(e)$ and $m, d_i(e) \geq 0$,
then
$n = (m + d_i(e)) \bmod X_i$
$\quad = m + d_i(e)$
Since
$m, d_i(e) \geq 0$
we have
$m + d_i(e) \geq m$.
giving
$n \geq m$.

$\square$

In the case where $X_i \leq d_i(e_v)$, no edge can be made innermost, resulting in a general formula of $(X_i - d_i(e_v))^+$ edges with $d_i(e'_v) = 0$, where x^+ is the positive part of x.

For an arbitrary distance vector, if any outer entry is left greater than 0 after unroll-and-jam, the new edge will not be innermost. Additionally, once a dependence edge becomes innermost, unrolling any loop additionally creates a copy of that edge. For example, in Figure 3.3, the load from `A(I,J+1)` has an incoming

```
      DO 10 J = 1,N            |       DO 10 J = 1,N,3
        DO 10 I = 1,N          |         DO 10 I = 1,N
10        A(I,J) = A(1,J-2)    |           A(I,J)   = A(I,J-2)
                               |           A(I,J+1) = A(I,J-1)
                               |           A(I,J+2) = A(I,J)
                               |   10      A(I,J+3) = A(I,J+1)
```

Figure 3.3 V_r^C Before and After Unroll-and-Jam innermost edge that is a copy of the incoming edge for the load from `A(I,J)`. The edge into `A(I,J+1)` was created by additional unrolling of the `J`-loop after `A(I,J)`'s edge was created. Quantifying these ideas, we have

$$\prod_{1 \le i < n} (X_i - d_i(e_v))^+$$

references that can be scalar replaced. Subtracting this value from the total number of references before scalar replacement and after unroll-and-jam gives the value for M_r^C. Therefore,

$$M_r^C = \sum_{v \in V_r^C} \Big(\prod_{1 \le i < n} X_i - \prod_{1 \le i < n} (X_i - d_i(e_v))^+ \Big).$$

In Figure 3.3, $X_1 = 4$ and $d_1(e) = 2$ for the reference to `A(I,J-2)`. Our formula correctly predicts that 2 memory references will be removed. Note that if $X_1 \le 2$ were true, no memory references would have been removed because no distance vector would have a 0 in the first entry.

For references that are invariant with respect to a loop, memory behavior is different from variant references. If $v \in V_r^I$ is invariant with respect to loop L_i, $i < n$, unrolling L_i will not introduce more references to memory because each unrolled copy of v will access the same memory location as v. This is because the induction variable for L_i does not appear in the subscript expression of v. Unrolling any loop that is not v-invariant will introduce memory references. Finally, if v is invariant with respect to the innermost loop, no memory references will be left after scalar replacement no matter which outer loops are unrolled. In Figure 3.4, unrolling the `K`-loop introduces memory references, while unrolling the `J`-loop does not. Using these properties, we have

$$M_r^I = \sum_{v \in V_r^I} \Big(\prod_{1 \le i < n} \omega(e_v, i, n) \Big)$$

where

```
$\omega(e,i,n) \Leftarrow$ if $e$ is invariant wrt loop $n$ then
                  return 0
              else if $e$ is invariant wrt loop $i$ then
                  return 1
              else
                  return $X_i$
```

Members of V_w^C behave exactly as those in V_r^C except that scalar replacement will remove stores only when their incoming dependence is loop independent. Thus, for M_w^C we have

$$M_w^C = \sum_{v \in V_w^C} \zeta(v, 1, n, n)$$

```
      DO 10 K = 1,N               |        DO 10 K = 1,N,2
        DO 10 J = 1,N             |          DO 10 J = 1,N,2
          DO 10 I = 1,N           |            DO 10 I = 1,N
10          ... = A(I,K)          |              ... = A(I,K)
                                  |              ... = A(I,K+1)
                                  |              ... = A(I,K)
                                  |   10         ... = A(I,K+1)
```

Figure 3.4 V_r^I Before and After Unroll-and-Jam where

$$\begin{array}{rl} \zeta(v,j,k,n) = & \text{if } d_n(e_v) = 0 \text{ and } e_v \text{ is an output edge then} \\ & \quad \text{return } \left(\prod_{j \le i < k} X_i - \prod_{j \le i < k} (X_i - d_i(e_v))^+ \right) \\ & \text{else} \\ & \quad \text{return } \prod_{j \le i < k} X_i \end{array}$$

Finally, M^I_w is defined exactly as M^I_r.

Now that we have obtained the formulation for loop balance in relation to unroll-and-jam, we can use it to prove that unroll-and-jam will never increase loop balance. Consider the following formula for β_L when unrolling only loop i.

$$\beta_L = \frac{\sum_{v \in V^\emptyset} X_i + \sum_{v \in V^C_r} X_i - (X_i - d_i(e_v))^+ + \sum_{v \in V^I_r \cup V^I_w} \omega(e_v, i, n) + \sum_{v \in V^C_w} \zeta(v, i, i+1, n)}{f \times X_i}$$

If we examine the vertices and edges in each of the above vertex sets to determine the simplified formula for M_L, we obtain the following, where $\forall j, a_j \ge 0$.

$$\begin{array}{rcl} M^\emptyset & = & a_0 X_i \\ M^C_r & = & a_1 X_i + a_2 \\ M^I_r & = & a_3 X_i + a_4 \\ M^C_w & = & a_5 X_i + a_6 \\ M^I_w & = & a_7 X_i + a_8 \end{array}$$

Combining these terms we get

$$\beta_L = \frac{(c_0 X_i + c_1)}{f X_i}$$

where $c_0, c_1 \ge 0$. Note that the values of c_0 and c_1 may vary with the value of X_i. As X_i increases, c_0 may decrease and c_1 may increase because the value of $(X_i - d_i(e_v))^+$ may become positive for some $v \in V^C_r$. In the theorem below, we show that β_L will not increase as X_i increases, showing that unrolling one loop will not increase loop balance.

Theorem 3.3

The function for β_L is nonincreasing in the i dimension.

Proof

For the original loop, $X_i = 1$ and

$\beta_L = \frac{(c_0 + c_1)}{f}$

If we unroll-and-jam the i-loop $n-1$ times, $n > 1$, then $X_i = n$ giving

$\beta'_L = \frac{(c'_0 n + c_1 + c_2)}{fn}$

where c_2 is the sum of all $d_i(e_v)$ where $(X_i - d_i(e_v))^+$ became positive with the increase in X_i. Since c_0 will decrease or remain the same with the change in X_i, $c'_0 \le c_0$. The difference between c_0 and c'_0 is the number of edges where $(X_i - d_i(e_v))^+$ becomes positive with the change in X_i. Given that $X_i = n$, the maximum value of any $d_i(e_v)$ added into c_2 is $n-1$ (any greater value would not make $(X_i - d_i(e_v))^+$ positive). Therefore, the maximum value of β'_L is

$\beta'_L = \frac{(c'_0 n + c_1 + (c_0 - c'_0)(n-1))}{fn}$

To show that β_L is nonincreasing, we must show that $\beta_L \geq \beta'_L$ or that the unrolled loop's balance is no greater than the original balance.

$$\begin{array}{rcl} \beta_L & \geq & \beta'_L \\ \frac{n\beta_L}{n} & \geq & \beta'_L \\ \frac{n(c_0+c_1)}{nf} & \geq & \frac{(c'_0 n+(c_0-c'_0)(n-1)+c_1)}{fn} \\ nc_0 + nc_1 & \geq & nc'_0 + nc_0 - nc'_0 + c'_0 - c_0 + c_1 \\ (n-1)c_1 & \geq & c'_0 - c_0 \end{array}$$

Since $n > 1, c_1 \geq 0$ and $c'_0 \leq c_0$, the inequality holds and β_L is nonincreasing. □

Given that we unroll in an outermost- to innermost-loop order, unrolling loop 1 produces a new loop with balance β^1_L such that $\beta^1_L \leq \beta_L$ by Theorem 3.3. After unrolling loop i, assume we have $\forall j < i, \beta^i_L \leq \beta^j_L$. If we next unroll loop $i+1$ to produce β^{i+1}_L, then again by Theorem 3.3, $\beta^{i+1}_L \leq \beta^i_L$. Therefore, unrolling multiple loops will not increase loop balance.

An Example. As an example of a formula for loop balance, consider the following loop.

```
      DO 10 J = 1,N
         DO 10 I = 1,N
10          A(I,J) = A(I,J-1) + B(I)
```

We have `A(I,J)` $\in V^\emptyset$, `A(I,J-1)` $\in V^C_r$ with an incoming distance vector of $\langle 1, 0\rangle$, and `B(I)` $\in V^I_r$. This gives

$$\beta_L = \frac{X_1+X_1-(X_1-1)^+ +1}{X_1}$$

3.3.2 Estimating Register Pressure

Now that we have the formula for computing loop balance given a set of unroll amounts, we must create the formula for register pressure. To compute the number of registers required by an unrolled loop, we need to determine which references can be removed by scalar replacement and how many registers each of those references need. We will refer to this quantity as R. As in computing M, we consider the partitioned sets of V, $\{V^\emptyset, V^C_r, V^I_r\}$, but we do not consider V_w because its register requirements are included in the computation of V_r. Associated with each of the partitions of V is the number of registers required by that set, $\{R^\emptyset, R^C_r, R^I_r\}$, giving

$$R = R^\emptyset + R^C_r + R^I_r.$$

Since each member of $V^\emptyset$ has no incoming dependence, we cannot use registers to capture reuse. However, members of $V^\emptyset$ still require a register to hold a value during expression evaluation. To compute the number of registers required for those memory references not scalar replaced and for those temporaries needed during the evaluation of expressions, we use the tree labeling technique of Sethi and Ullman [SU70]. Using this technique, the number of registers required for each expression in the original loop body is computed and the maximum over all expressions is taken as the value for $R^\emptyset$.

For variant references whose incoming edge is carried by some loop, $v \in V^C_r$, we need to determine which (or how many) references will be removed by scalar replacement after unroll-and-jam and how many registers are required for those references. The former value is derived in the computation of M^C_r and is shown below.

$$\sum_{v\in V^C_r} \Big(\prod_{1\leq i<n} (X_i - d_i(e_v))^+ \Big)$$

The latter value, $d_n(e_v)+1$, comes from scalar replacement, where we assume that all registers interfere. Any value that flows across a loop-carried dependence interferes with all other values and predicting the effects of scheduling before optimization to handle loop-independent interference is beyond the scope of this work. Therefore,

$$R^C_r = \sum_{v\in V^C_r} \Big(\prod_{1\leq i<n} (X_i - d_i(e_v))^+ \Big) \times (d_n(e_v)+1).$$

References that are invariant with respect to an outer loop require registers only if a corresponding reference-invariant loop is unrolled. However, references that are invariant with respect to the innermost loop always require registers. For each $v \in V_r^I$, unrolling loops that are not v-invariant creates references that possibly require additional registers. In Figure 3.4, the reference to `A(I,K)` requires no registers unless the `J`-loop is unrolled and since `K` is unrolled once, two registers are required. Formulating this behavior, we have

$$R_r^I = \sum_{v \in V_r^I} (\prod_{1 \leq i < n} \alpha(e_v, i, n))$$

where

```
$\alpha(e, i, n) \Leftarrow$ if $e$ is invariant wrt loop $i$ then
        if $(\exists X_j | X_j > 1 \wedge e$ is invariant wrt loop $j)$ or
            $(e$ is invariant wrt loop $n)$ then
          return 1
        else
          return 0
      else
        return $X_i$
```

An Example. As an example of a formula for computing register pressure, consider the following loop.

```
      DO 10 J = 1,N
         DO 10 I = 1,N
 10         A(I,J) = A(I,J-1) + B(J)
```

We have `A(I,J)` $\in V^{\emptyset}$, `A(I,J-1)` $\in V_r^C$ with an incoming distance vector of $\langle 1, 0\rangle$ and `B(J)` $\in V_r^I$. This gives

$$R = \quad 1 + (X_1 - 1)^+ + X_1.$$

3.4 Applying Unroll-and-Jam in a Compiler

In this section, we will discuss how to take the previous optimization problem and solve it practically for real program loop nests. In general, integer-programming problems are NP-complete, but we can make simplifying assumptions for our optimization problem that will make its solution tractable. First, we show how to choose the loops to which unroll-and-jam will be applied and then we show how to choose the unroll amounts for those loops.

3.4.1 Picking Loops to Unroll

Applying unroll-and-jam to all of the loops in an arbitrary loop nest can make the solution to our optimization problem extremely complex. To simplify the solution procedure, we will only apply unroll-and-jam to a subset of the loop nest. Since experience suggests that most loop nests have a nesting depth of 3 or less, we can limit unroll-and-jam to 1 or 2 loops in a given nest.

Our heuristic for determining the subset of loops for unroll-and-jam is to pick those loops whose unrolling is likely to result in the fewest memory references in the unrolled loop. In particular, we unroll the loops that carry the most dependences that will be innermost after unroll-and-jam is applied. To find these loops, we examine each distance vector to determine if it can be made innermost for each pair of loops in the nest. Given a distance vector $d(e) = \langle d_1, \ldots, d_i, \ldots, d_j, \ldots, d_n\rangle$ where $d_i, d_j \geq 0$, unrolling loops i and j can make e innermost if $\forall k, k \neq i, j, n$ we have $d_k(e) = 0$. If we unroll only one loop, i, then e can be made innermost if $\forall k, k \neq i, n$ we have $d_k(e) = 0$. The algorithm for choosing loops is shown in Figure 3.5.

```
procedure PickLoops($V$)

Input: $V$ = array references in a loop

  for each $v \in V$
    if $v$ is a memory read or $d_n(e_v) = 0$ then
      for $i = 1$ to $n - 1$
        for each $j \geq i | \forall k, k \neq i, j, n$ we have $d_k(e) = 0$
          count($i, j$)++
  if unrolling two loops
    unroll loops $i, j|$ count($i, j$) is the maximum
  else
    unroll loop $i|$ count($i, i$) is the maximum
end
```

Figure 3.5 PickLoops

3.4.2 Picking Unroll Amounts

In the following discussion of our problem solution, we consider unrolling only one loop to bring clarity to the solution procedure. Later, we will discuss how to extend the solution to the problem for two loops. Given this restriction, we can reduce the optimization formula to the following, where $\delta = (X_i - d_i(e))^+$ and ϱ is the effective size of the floating-point register set for the target architecture.

$$\min \left| \frac{\sum_{v \in V^\emptyset} X_i + \sum_{v \in V_r^C} (X_i - \delta) + \sum_{v \in V_r^I \cup V_w^I} \omega(e_v, i, n) + \sum_{v \in V_w^C} \zeta(e_v, i, i+1, n)}{f \times X_i} - \beta_m \right|$$

$$R^\emptyset + X_i \times \sum_{v \in V_r^C} (\delta \times (d_n(e_v) + 1)) + \sum_{v \in V_r^I} \alpha(e_v, i, n) \leq \varrho$$

$$X_i \geq 1$$

We begin the solution procedure by examining the vertices and edges of the dependence graph to accumulate the coefficient for X_i and any constant value as was shown earlier. Essentially, we solve the summations to get a linear function of X_i. However, the $^+$-function requires us to keep an array of coefficients and constants, one set for each value of d_i, that will be used depending upon the value of X_i. In practice, most distances are 0, 1 or 2, allowing us to keep 4 sets of values: one for each common distance and one for the remaining distances. If $d_i > 2$ is true for some edge, our results may be a bit imprecise if $3 \leq X_i < d_i$. However, experimentation discovered no instances of this situation.

Given a set of coefficients, we can search the solution space, while checking register pressure, to determine the best unroll factor. Since most dependence distances are 0, 1 or 2, unrolling more than ϱ will most likely increase the register pressure beyond the physical limits of the target machine. Therefore, by Theorem 3.3, we can find the solution to the optimization problem in the solution space in $O(\log \varrho)$ time.

To extend unroll-and-jam to two loops, we can create a simplified problem from the original by setting $n = 2$. A formula for two particular loops, i and j, can be constructed by examining the dependence distance vectors as in the case for one loop. By Theorem 3.3, if we hold either X_i or X_j constant, the function for β_L is nonincreasing, which gives a two dimensional solution space with each row and column sorted in decreasing order, lending itself to an $O(\varrho)$ intelligent search. In general, a solution for unroll-and-jam of k loops, $k > 1$, can be found in $O(\varrho^{k-1})$ time.

3.4.3 Removing Interlock

After unroll-and-jam, a loop may contain pipeline interlock, leaving idle computational cycles. To remove these cycles, we use the technique of Callahan, et. al, to estimate the amount of interlock and then unroll one loop to create more parallelism by introducing enough copies of the inner loop recurrence [CCK88]. Essentially, the length of the longest recurrence carried by the innermost loop is calculated and then unroll-and-jam is applied to an outer loop until there are more floating-point operations than the recurrence length.

3.4.4 Multiple Edges

In general, we cannot assume that there will be only one edge entering or leaving each node in the dependence graph since multiple incident edges are possible and likely. One possible solution is to treat each edge separately as if it were the only incident edge. Unfortunately, this is inadequate because many edges may correspond to the flow of the same set of values. In the loop,

```
      DO 10 J = 1,N
         DO 10 I = 1,N
10          B(I,J) = A(I-1,J) + A(I-1,J) + A(I,J)
```

there is an input dependence from `A(I,J)` to each `A(I-1,J)` with distance vector $\langle 0,1\rangle$ and a loop-independent dependence between the `A(I-1,J)`'s. Considering each edge separately would give a register pressure estimation of 5 registers per unrolled iteration of `J` rather than the actual value of 2. Both values provided come from the same source, requiring us to use the same registers. Although the original estimation is conservatively safe, it is more conservative than necessary. To improve our estimation, we consider register sharing.

To relate all references that access the same values, our scalar replacement algorithm considers the oldest reference in a dependence chain as the reference that provides the value for the entire chain. Using this technique alone to capture sharing within the context of unroll-and-jam, however, can cause us to miss registers that are used. Consider the following example.

```
     DO 10 I = 1,N
        DO 10 J = 1,N
10         C(I,J) = A(I+1,J) + A(I,J) + A(I,J)
```

The second reference to `A(I,J)` has two incoming input dependences: one with a distance vector of $\langle 1,0\rangle$ from `A(I+1,J)` and one with a distance vector of $\langle 0,0\rangle$ from `A(I,J)`. After unrolling the `I`-loop by 1 we have

```
     DO 10 I = 1,N,2
        DO 10 J = 1,N
1          C(I,J)   = A(I+1,J) + A(I,J)   + A(I,J)
10         C(I+1,J) = A(I+2,J) + A(I+1,J) + A(I+1,J)
```

Now, the reference `A(I+1,J)` in statement `1` provides the value used in statement `10` by both references to `A(I+1,J)`, and the first reference to `A(I,J)` in statement `1` provides the value used in the second reference to `A(I,J)` in statement `1`. Here, we cannot isolate which of the two references to `A` will ultimately provide the value for the second `A(I,J)` before unrolling because they both provide the value at different points. Therefore, we cannot pick just the edge from the oldest reference, `A(I+1,J)`, to calculate its register pressure. Instead, we can use a stepwise approximation where each possible oldest reference within a dependence chain is considered.

The first step is to partition the dependence graph so that all references using the same registers after unrolling will be put in the same partition. The algorithm in Figure 3.6 accomplishes this task. Once we have the references partitioned, we apply the algorithm in Figure 3.7 to calculate the coefficients used in R. First, in the procedure *OldestValue*, we compute the node in each register partition that is first to reference the value that flows throughout the partition. We look for the reference, v, that has no incoming dependence from another member of its partition or has only loop-carried incoming dependences that are carried by v-invariant loops. We consider only those edges that can create reuse in the unrolled loop as described in Section 3.4.1. After unroll-and-jam, the oldest reference will provide the value for scalar replacement for the entire partition and is called the *generator* of the partition.

Next, in procedure *SummarizeEdges*, we determine the distance vector that will encompass all of the outgoing edges (one to each node in the partition) from the generator of each partition. Because we want

```
Procedure PartitionNodes($G, j, k$)

Input: $G = (V, E)$, the dependence graph
         $j, k$ = loops to be unrolled

    put each $v \in V$ in its own partition, $P(v)$
    while $\exists$ an unmarked node do
        let $v$ be an arbitrary unmarked node
        mark $v$
        forall $w \in V | ((\exists e = (w, v) \vee e = (v, w)) \wedge$
          ($e$ is input or true and is consistent) $\wedge$
          (for $i = 1 \ldots n-1, i \neq j, i \neq k, d_i(e) = 0$)) do
            mark $u$
            merge $P(u)$ and $P(v)$
        enddo
    enddo
end
```

Figure 3.6 PartitionNodes

to be conservative, for $i = 1, \ldots, n-1$ we pick the minimum d_i for all of the generator's outgoing edges. This guarantees that the innermost-loop reuse within a partition will be measured as soon as possible in the unrolled loop body. For d_n, we pick the maximum value to make sure that we do not underestimate the number of registers used. After computing a summarized distance vector, we use it to accumulate the coefficients to the equation for register pressure.

At this point in the algorithm, we have created a distance vector and formula to estimate the register requirements of a partition partially given a set of unroll values. In addition, we must account for the intermediate points where the generator of a partition does not provide the reuse at the innermost loop level for some of the references within the partition as shown in the previous example (i.e. `A(I-1,J)` and `A(I,J)`). We will underestimate register requirements if we fail to handle these situations.

First, using procedure *Prune*, we remove all nodes from a partition for which there are no intermediate points where a reference other than the current generator provides the values for scalar replacement. Any reference that occurs on the same iteration of loops 1 to $n-1$ as the current generator of the partition will have no such points. Their value will always be provided by the current generator. Next, we compute the additional number of registers needed by the remaining nodes in the partition by calculating the coefficients of R for an intermediate generator of the modified partition. The only difference from the computation for the original generator is that we modify the coefficients of R derived from the summarized distance vector of the modified partition by accounting for the registers that we have already counted with the previous generator. This is done by computing the coefficients for R as if an intermediate generator were the generator in an original partition. Then, we subtract from R the number of references that have already been handled by the previous generator. We use the distance vector of the edge, $\hat{e}$, from the previous generator, v, to the new intermediate generator, u, to compute the latter value. Given d^s for the current summarized distance vector and $d(\hat{e})$ for the edge between the current and previous generators, we compute $R_{d^s} - R_{(d^s + d(\hat{e}))}$. The value $d^s + d(\hat{e})$ would represent the summarized distance vector if the previous generator were the generator of the modified partition. In our example, $d^s = \langle 0, 0 \rangle$ and $d(\hat{e}) = \langle 1, 0 \rangle$ giving $(X_1 - 0)^+ - (X_1 - 1)^+$ additional references removed. These steps for intermediate points are repeated until the partition has 1 or less members.

With the allowance of multiple edges, references may be invariant with respect to only one of the unrolled loops and still have a dependence carried by the other unrolled loop. In this case, we treat the references as a member of V^I with respect to the first loop and V^C with respect to the second.

Partitioning is not necessary to handle multiple edges for M because sharing does not occur. Instead, we consider each reference separately using a summarized vector for each $v \in V$ that consists of the minimum d_i over all incoming edges of v for $i = 1, \ldots, n-1$. Although this is a slightly different approximation then used for R, it is more accurate and will result in a better estimate of β_L.

```
Procedure ComputeR($P, G, R$)

Input: $P$ = partition of memory references
       $G = (V, E)$, the dependence graph
       $R$ = register pressure formula

  foreach $p \in P$ do
    $v$ = OldestValue($p$)
    $d$ = SummarizeEdges($v$)
    update $R$ using $d$
    Prune($p$,$v$)
    while size($p$) > 1 do
      $u$ = OldestValue($p$)
      $d$ = SummarizeEdges($u$)
      update $R$ using $R_d - R_{d+d(\hat{e})}, \hat{e} = (v, u)$
      Prune($p$,$u$)
      $v = u$
    enddo
  enddo
end

Procedure OldestValue($p$)
  foreach $v \in p$
    if $\forall e = (u, v), v$ is invariant wrt the loop at level($e$)$\vee$
                $P(u) \neq P(v))$ then return $v$
end

Procedure SummarizeEdges($v$)
  for $i = 1$ to $n - 1$
    foreach $e = (v, w) \in E, w$ is a memory read
      $d_i^s = \min(d_i^s, d_i(e))$
  foreach $e \in E, w$ is a memory read
    $d_n^s = \max(d_n^s, d_n(e))$
  return $d^s$
end

Procedure Prune($p$,$v$)
  remove $v$ from $p$
  foreach $e = (v, w) \in E$
    if $d(e) = (=, =, \ldots, =, *)$ then remove $w$
end
```

Figure 3.7 Compute Coefficients for Optimization Problem

3.4.5 Multiple-Induction-Variable Subscripts

In Section 3.2, we discussed a method for updating the dependence graph for references with one induction variable in each subscript position. Unfortunately, this method does not work on subscripts containing multiple induction variables, called MIV subscripts, that have incoming consistent dependences [GKT91]. Consider the following loop.

```
      DO 10 I = 1,N
        DO 10 J = 1,N
10        A(I) = A(I) + B(I-J)
```

The load of `B(I-J)` has a loop-independent incoming consistent dependence from itself carried by the `I`-loop. After unroll-and-jam of the `I`-loop by a factor of 2, we have

```
      DO 10 I = 1,N,3
        DO 10 J = 1,N
          A(I) = A(I) + B(I-J)
          A(I) = A(I) + B(I-J+1)
10        A(I) = A(I) + B(I-J+2)
```

Here, there is an input dependence carried by the innermost loop from `B(I-J+2)` to both `B(I-J+1)` and `B(I-J)`. It was not possible for new references that contained only single-induction-variable subscripts, called SIV subscripts, to have multiple edges that have the same source and that arose from one original edge. In the above example, even though the `I`-loop step value has increased, the presence of `J` in the subscript removes its effect for the execution of the innermost loop. The formula presented in Section 3.2 is invalid in this case since the change in loop step does not have the same effect on dependence distance and location.

In general, we would need to re-apply dependence analysis to MIV subscripts to get the full and correct dependence graph. This is because of the interaction between multiple MIV subscripts. In addition, it would be very difficult to compute M and R given the behavior of such subscripts. Therefore, we restrict the type of MIV subscripts that we handle to a small, predictable subset, V^{MIV}. In our experimentation, the only MIV subscripts that we found fit into the following description of V^{MIV}.

In order for a reference to be classified as MIV in our system, it must have the following properties:

1. The reference is incident only upon consistent dependences.
2. The reference contains only one MIV subscript position.
3. The inner-loop induction variable does not appear in the subscript or it appears only in the MIV subscript position.
4. At most, one unrolled-loop induction variable is in the MIV subscript position.
5. The coefficients of the induction variables in the MIV subscript position are 1 for an unrolled loop and 1 or -1 for the innermost loop.

If an MIV reference is invariant with respect to the innermost loop, it is treated as a member of V_r^I or V_w^I. If a loop is unrolled and its induction variable is not in the MIV subscript position, we use the classification of the reference in relation to the unrolled-loop induction variable when unroll-and-jam is applied (e.g, `B(I-J)` would be treated as a member of V_r^I with respect to a `K`-loop). Any reference containing multiple-induction-variable subscripts that do not fit the above criteria are classified as members of $V^{\emptyset}$.

To update the dependence graph for V^{MIV}, we restrict ourselves to the innermost loop to satisfy the needs of scalar replacement only. Essentially, we will apply the strong SIV test for the innermost-loop induction variable on each MIV reference pair [GKT91]. Given the references, $A(a_nI_n + a_0)$ and $A(b_nI_n + b_0)$, where $a_n = b_n$, there is a dependence between the two references with a consistent threshold d if $d = \frac{a_0 - b_0}{a_n}$ is an integer. Since we restrict a_n and b_n to be 1 or -1, there will always be a dependence.

To compute M^{MIV} and R^{MIV}, we must recognize that the effect of unroll-and-jam on MIV subscripts is to increase the distance from the first to the last access of a value by the unroll amount of the loop. Consider the following loop.

```
     DO 10 I = 1,N
       DO 10 J = 1,N
10       ... =  B(I-J) + B(I-J+1)
```

Unroll-and-jam of `I` by 1 produces

```
     DO 10 I = 1,N,3
       DO 10 J = 1,N
         ... = B(I-J) + B(I-J+1)
10       ... = B(I-J+1)+ B(I-J+2)
```

In the original loop, the dependence distance for `B`'s partition was 1. After unroll-and-jam by 1, the distance is 2. This example shows that each $v \in V_r^{MIV}$ requires $X_i + d_n(e)$ registers, assuming a step value of 1 for loop i, when unrolling the loop whose induction variable is in the MIV subscript position. In the loop,

```
     DO 10 I = 1,N
       DO 10 J = 1,N
10       ... =  B(I-J)
```

we must unroll the `I`-loop at least once to get a dependence carried by the innermost loop. To quantify this, we have

$$R_r^{MIV} = \sum_{v \in V_r^{MIV}} \textsc{pos}(X_i - d_i(e_v)) \times (X_i + d_n(e_v))$$

where $d_n(e_v)$ is the maximum distance in a partition and $\textsc{pos}(x)$ returns 1 if $x > 0$, otherwise it returns 0. Here, $\textsc{pos}(X_i - d_i(e_v))$ ensures that a dependence will be carried by the innermost loop after unroll-and-jam.

Only the MIV reference that is first to access a value will be left as a reference to memory after unroll-and-jam and scalar replacement. This is because each later reference introduced by unroll-and-jam will be the sink of an innermost dependence, as shown in the examples. Therefore,

$$M_r^{MIV} = \sum_{v \in V_r^{MIV}} 1.$$

For each $v \in V_w^{MIV}$, unrolling at least $d_n(e_v)$ will create an identical subscript expression, resulting in an incoming loop-independent dependence. Therefore,

$$M_w^{MIV} = \sum_{v \in V_w^{MIV}} \min(X_i, d_n(e_v)),$$

where $d_n(e_v)$ is the minimum over all of v's incoming output dependence edges.

3.4.6 Non-Perfectly Nested Loops

It may be the case that the loop nest on which we wish to perform unroll-and-jam is not perfectly nested. Consider the following loop.

```
     DO 10 I = 1,N
        DO 20 J = 1,N
20         A(I) = A(I) + B(J)
        DO 10 J = 1,N
10         C(J) = C(J) + D(I)
```

On the RS/6000, where $\beta_M = 1$, the `I`-loop would not be unrolled for the loop surrounding statement `20`, but would be unrolled for the loop surrounding statement `10`. To handle this situation, we can distribute the I-loop around the J-loops as follows

```
      DO 20 I = 1,N
         DO 20 J = 1,N
20          A(I) = A(I) + B(J)
      DO 10 I = 1,N
         DO 10 J = 1,N
10          C(J) = C(J) + D(I)
```

i

and unroll each loop independently.

In general, we compute unroll amounts with a balance-optimization formula for each innermost loop body as if it were within a perfectly nested loop body. A vector of unroll amounts that includes a value for each of a loop's surrounding loops and a value for the loop itself is kept for each loop within the nest. When determining if distribution is necessary, if we encounter a loop that has multiple inner loops at the next level, we compare each of the unroll vectors for each of the inner loops to determine if any unroll amounts differ. If they differ, we distribute each of the loops from the current loop to the loop at the outermost level where the unroll amounts differ. We then proceed by examining each of the newly created loops for additional distribution. To handle safety, if any loop that must be distributed cannot be distributed, we set the unroll vector values in each vector to the smallest unroll amount in all of the vectors for each level. Distribution is unsafe if it breaks a recurrence [AK87]. The complete algorithm for distributing loops for unroll-and-jam is given in Figure 3.8.

3.5 Experiment

Using the experimental system described in Chapter 2, we have implemented unroll-and-jam as presented in this chapter. In addition to controlling floating-point register pressure, we found that address-register pressure needed to be controlled. For the RS/6000, any expression having no incoming innermost-loop-carried or loop-independent dependence required an address register. Although this removes our independence from a particular compiler, it could not be avoided for performance reasons. To allow for an imperfect scalar optimizer, the effective register-set size for both floating-point and address registers was chosen to be 26. This value was determined experimentally.

DMXPY. For our first experiment, we choose DMXPY from the Level 2 BLAS library [DDHH88]. This version of DMXPY is a hand-optimized vector-matrix multiply that is machine-specific and difficult to understand. We applied our transformations to the unoptimized version and compared the performance with the BLAS2 version.

Array Size	Iterations	Original	BLAS2	UJ+SR	Speedup
300	700	7.78s	5.18s	5.06s	1.54

As the table shows, we were able to attain slightly better performance than the BLAS version, while allowing the kernel to be expressed in a machine-independent fashion. The hand-optimized version was not specific to the RS/6000 and more hand optimization would have been required to tune it to the RS/6000. However, our automatic techniques took care of the machine-specific details without the knowledge or intervention of the programmer and still obtained good performance.

Matrix Multiply. The next experiment was performed on two versions of matrix multiply. The JKI loop ordering for better cache performance is shown below.

```
      DO 10 J = 1,N
         DO 10 K = 1,N
            DO 10 I = 1,N
10             C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

The results from our transformation system show that integer factor speedups are attainable on both small and large matrices for both versions of the loop. Not only did we improve memory reuse within the loop,

```
Procedure DistributeLoops($L, i$)

Input: $L$ = loop to check for distribution
       $i$ = nesting level of $L$

  if $L$ has multiple inner loops at level $i+1$ then
    $l$ = outermost level where the unroll amounts differ for the
        loop nests.
    if $l > 0$ then
      if loops at levels $l, \ldots, i$ can be distributed then
        $\mathcal{L} = L$
        while level($\mathcal{L}$) $\leq l$ do
          distribute $\mathcal{L}$ around its inner loops
          $\mathcal{L}$ = parent($\mathcal{L}$)
        enddo
      else
        for $j = 1$ to $i$ do
          $m$ = minimum of all unroll vectors for the inner loops of
              $L$ at level $j$
          assign to those vectors at level $j$ the value $m$
        enddo
      endif
  endif
  $N$ = the first inner loop of $L$ at level $i+1$
  if $N$ exists then
    DistributeLoops($N, i+1$)
  $N$ = the next loop at level $i$ following $L$
  while $N$ exists do
    DistributeLoops($N, i$)
    $N$ = the next loop at level $i$ following $N$
  enddo
end
```

Figure 3.8 Distribute Loops for Unroll-and-jam

but also available instruction-level parallelism.

Loop Order	Array Size	Iterations	Original	SR	UJ+SR	Speedup
JIK	50x50	500	4.53s	4.53s	2.37s	1.91
	500x500	1	135.61s	135.61s	44.16s	3.07
JKI	50x50	500	6.71s	6.71s	3.3s	2.04
	500x500	1	15.49s	15.49s	6.6s	2.35

Comparing the results of the two versions of matrix multiply shows how critical memory performance really is. The JKI version has better overall cache locality and on large matrices, that property gives dramatically better performance than the JIK version.

Linear Algebra Kernels. We also experimented with both the point and block versions of LU decomposition with and without partial pivoting (LU and LUP, respectively). In the table below, we show the results of scalar replacement and unroll-and-jam.

Kernel	Array Size	Block Size	Original	SR	UJ+SR	Speedup
Block LU	300x300	32	1.37s	0.91s	0.56s	2.45
	300x300	64	1.42s	1.02s	0.72s	1.97
	500x500	32	6.58s	4.51s	2.48s	2.65
	500x500	64	6.59s	4.54s	2.79s	2.36
Block LUP	300x300	32	1.42s	0.97s	0.67s	2.12
	300x300	64	1.48s	1.10s	0.77s	1.92
	500x500	32	6.85s	4.81s	2.72s	2.52
	500x500	64	6.83s	4.85s	3.02s	2.26

As shown in the above examples, a factor of 2 to 2.5 was attained. Each of the kernels contained an inner-loop reduction that was amenable to unroll-and-jam. In these instances, unroll-and-jam introduced multiple parallel copies of the inner-loop recurrence while simultaneously improving data locality.

NAS Kernels. Next, we studied three of the NAS kernels from the SPEC benchmark suite. A speedup was observed for each of these kernels with Emit and Gmtry doing much better. The reason is because one of the main computational loops in both Emit and Gmtry contained an outer-loop reduction that was unroll-and-jammed.

Kernel	Original	SR	UJ+SR	Speedup
Vpenta	149.68s	149.68s	138.69s	1.08
Emit	35.1s	33.83s	24.57s	1.43
Gmtry	155.3s	154.74s	25.95s	5.98

Geophysics Kernels. We also included in our experiment two geophysics kernels: one that computed the adjoint convolution of two time series and another that computed the convolution. Each of these loops required the optimization of trapezoidal, rhomboidal and triangular iteration spaces using techniques that we develop in Chapter 5. Again, these kernels contained inner-loop reductions.

Kernel	Iterations	Array Size	Original	SR	UJ+SR	Speedup
Afold	1000	300	4.59s	4.59s	2.55s	1.80
	1000	500	12.46s	12.46s	6.65s	1.87
Fold	1000	300	4.61s	4.61s	2.53s	1.82
	1000	500	12.56s	12.56s	6.63s	1.91

Applications. To complete our study we ran a number of Fortran applications through our translator. We chose programs from SPEC, Perfect, RICEPS and local sources. Those programs that belong to our benchmark suites but are not included in this experiment contained no opportunities for our algorithm. The results of performing scalar replacement and unroll-and-jam on these applications are shown in the following table.[‡]

Program	Original	SR	UJ+SR	Speedup
Arc2d	410.13s	407.57s	401.96	1.02
CoOpt	122.88s	120.44s	116.67s	1.05
Flo52	61.01s	58.61s	58.8s	1.04
Matrix300	149.6s	149.6s	33.21s	4.50
Onedim	4.41s	4.41s	3.96s	1.11
Simple	963.20s	934.13s	928.84s	1.04
Tomcatv	37.66s	37.66s	37.41s	1.01

The applications that observed the largest improvements with unroll-and-jam (Matrix300, Onedim) were dominated by the cost of loops containing reductions that were highly optimized by our system. Although many of the loops found in other programs received a 15%-40% improvement, the applications themselves were not dominated originally by the costs of these loops.

Throughout our study, we found that unroll-and-jam was most effective in the presence of reductions. Unrolling a reduction introduced very few stores and improved cache locality and instruction-level parallelism. Stores seemed to limit the perceived available parallelism on the RS/6000. Unroll-and-jam was least effective in the presence of a floating-point divide because of its high computational cost. Because a divide takes 19 cycles on the RS/6000, its cost dominated loop performance enough to make data locality a minimal factor.

3.6 Summary

In this chapter, we have developed an optimization problem that minimizes the distance between machine balance and loop balance, bringing the balance of the loop as close as possible to the balance of the machine. We have shown how to use a few simplifying assumptions to make the solution times fast enough for inclusion in compilers. We have implemented these transformations in a Fortran source-to-source preprocessor and shown its effectiveness by applying it to a substantial collection of kernels and whole programs. These results show that, over whole programs, modest improvements are usually achieved with spectacular results occurring on a few programs. The methods are particularly successful on kernels from linear algebra. These results are achieved on an IBM RS/6000, which has an extremely effective optimizing compiler. We would expect more dramatic improvements over a less sophisticated compiler. These methods should also produce larger improvements on machines where the load penalties are greater.

[‡]Our version of Matrix300 is after procedure cloning and inlining to create context for unroll-and-jam [BCHT90].

Chapter 4

Loop Interchange

The previous two chapters have addressed the problem of improving the performance of memory-bound loops under the assumption that good cache locality already exists in program loops. It is the case, however, that not all loops exhibit good cache locality, resulting in idle computational cycles while waiting for main memory to return data. For example, in the loop,

```
      DO 10 I = 1, N
        DO 10 J = 1, N
10        A = A + B(I,J)
```

references to successive elements of B are a long distance apart in number of memory accesses. Most likely, current cache architectures would not be able to capture the potential cache-line reuse available because of the volume of data accessed between reuse points. With each reference to B being a cache miss, the loop would spend a majority of its time waiting on main memory. However, if we interchange the I- and J-loops to get

```
      DO 10 J = 1, N
        DO 10 I = 1, N
10        A = A + B(I,J)
```

the references to successive elements of B immediately follow one another. In this case, we have attained locality of reference for B by moving reuse points closer together. The result will be fewer idle cycles waiting on main memory.

In this chapter we show how the compiler can automate the above process to attain a loop ordering with good memory performance. We begin with a model of memory performance for program loops. Then, we show how to use this model to choose the loop ordering that maximizes memory performance. Finally, we present an experiment with an implementation of this technique.

4.1 Performance Model

Our model of memory performance consist of two parts. The first part models the ability of cache to retain values between reuse points. The second part models the costs associated with each memory reference within an innermost loop.

4.1.1 Data Locality

When applied to data locality within cache, a data dependence can be thought of as a potential opportunity for reuse. The reason the reuse opportunity is only potential is cache interference — where two data items need to occupy the same location in cache at the same time. Previous studies have shown that interference is hard to predict and often prevents outer-loop reuse [CP90, LRW91]. However, inner-loop locality is likely to be captured by cache because of short distances between reuse points. Given these factors, our model of cache will assume that all outer-loop reuse will be prevented by cache interference and that all inner-loop reuse will be captured by cache. This assumption allows us to ignore the unpredictable effects of set associativity and concentrate solely on the cache line size, miss penalty and access cost to measure memory performance. In essence, by ignoring set associativity, we assume that the cache will closely resemble a fully associative cache in relation to innermost-loop reuse. Although precision is lowered, our experimentation shows that loop interchange based on this model is effective in attaining locality of reference within cache.

4.1.2 Memory Cycles

To compute the cost of an array reference, we must first know the array storage layout for a particular programming language. Our model assumes that arrays are stored in column-major order. Row-major order can be handled with slight modifications. Once we know the storage layout, we can determine where in the memory hierarchy values reside and the associated cost of accessing each value by analyzing reuse properties. In the rest of this section, we show how to determine the reuse properties of array references based upon the dependence graph and how to assign an average cost to each memory access.

One reuse property, *temporal reuse*, occurs when a reference accesses data that has been previously accessed in the current or a previous iteration of an innermost loop. These references are represented in the dependence graph as the sink of a loop-independent or innermost-loop-carried consistent dependence. If a reference is the sink of an output or antidependence, each access will be out of cache (assuming a write-back cache) and will cost $\mathcal{T} = C_h$ cycles, where C_h is the cache access cost. If a reference is the sink of a true or input dependence, then it will be removed by scalar replacement, resulting in a cost of 0 cycles per access. In Figure 4.1, the reference to `A(I-1,J)` has temporal reuse of the value defined by `A(I,J)` 1 iteration earlier and will be removed by scalar replacement. Therefore, it will cost 0 cycles.

The other reuse property, *spatial reuse*, occurs when a reference accesses data that is in the same cache line as some previous access. One type of reference possessing spatial reuse has the innermost-loop induction variable contained only in the first subscript position and has no incoming consistent dependence. This reference type accesses successive locations in a cache line on successive iterations of the innermost loop. It requires the cost of accessing cache for every memory access plus the cache miss penalty, C_m, for every access that goes to main memory. Assuming that the cache line size is C_l words and the reference has a stride of s words between successive accesses, $\lfloor \frac{C_l}{s} \rfloor$ successive accesses will be in the same cache line. Therefore, the probability that a reference with spatial reuse will be a cache miss is

$$\mathcal{P}_m = \frac{1}{\lfloor \frac{C_l}{s} \rfloor},$$

giving the following formulation for memory cost

$$\mathcal{S}_l = C_h + \mathcal{P}_m * C_m.$$

In Figure 4.1, the reference to `A(I,J)` has spatial reuse of the form just described. Given a cache with $C_h = 1, C_m = 8$ and $C_l = 16$, `A(I,J)` costs $1 + \frac{8}{16} = 1.5$ cycles/access.

The other type of reference having spatial reuse accesses memory within the same cache line as a different reference but does not access the same value. These references will be represented as the sink of an outer-loop-carried consistent dependence with a distance vector of $\langle 0, \ldots, 0, d_i, 0, \ldots, 0, d_n \rangle$, where the induction variable for loop i is only in the first subscript position. In this case, memory accesses will be cache hits on all but possibly the first few accesses and essentially cost $\mathcal{S}_T = C_h$. In Figure 4.1, the reference to `C(J-1,I)` has this type of spatial reuse due to the reference to `C(J,I)`. Under the previous cache parameters, `C(J-1,I)` requires 1 cycle/access.

The remaining types of references can be classified as those that contain no reuse and those that are stored in registers. Array references without spatial and temporal reuse fall into the former category and require $\mathcal{N} = C_h + C_m$ cycles. References to scalars fall into the latter category and require 0 cycles. `C(J,I)` in Figure 4.1 has no reuse since the stride for consecutive elements is too large to attain spatial reuse. It has a cost of 9 cycles/access under the current cache parameters.

```
      DO 10 J = 1,N
        DO 10 I = 1,N
10        A(I,J) = A(I-1,J) + C(J,I) + C(J-1,I)
```

Figure 4.1 Example Loop for Memory Costs

4.2 Algorithm

This section presents an algorithm for ordering loop nests to maximize memory performance that is based upon the memory-cost model described in the previous section. First, we show how to compute the order for memory performance for perfectly nested loops. Then, we show how to extend the algorithm to handle non-perfectly nested loops.

4.2.1 Computing Loop Order

Using the model of memory from Section 4.1, we will apply a loop interchange algorithm to a loop nest to minimize the cost of accessing memory. The algorithm considers the memory costs of the references in the innermost-loop body when each loop is interchanged to the innermost position and then orders the loops from innermost to outermost based upon increasing costs. Although costs are computed relative only to the innermost position, they reflect the data-locality merit of a loop relative to all other loops within the nest. Loops with a lower memory cost have a higher locality and have a higher probability of attaining outer-loop reuse if interference happens not to occur. To actually compute the memory costs associated with each loop as described, the distance vectors of all of the dependence edges in the loop nest are viewed as if the entry for the loop under consideration were shifted to the innermost position with all other entries remaining in the same relative position. This will allow determination of which edges can be made innermost to capture reuse.

After computing the memory cost for each loop and computing the order for the loops, we must ensure that no violation of dependences will occur. Safety is ensured using the technique of McKinley, et al. [KM92]. Beginning, with the outermost position in the loop, we select the loop with the highest memory cost that can safely go in the current loop position. A loop positioning is safe if no resulting dependence has a negative threshold [AK87]. After selecting the outermost loop, we proceed by iteratively selecting the loop with the next highest cost for the next outermost position until a complete ordering is obtained. The algorithm for computing loop order for perfectly nested loops is shown in Figure 4.2.

An Example. To illustrate the algorithm in Figure 4.2, consider the following loop under two sets of memory costs.

```
      DO 10 I = 1,N
         DO 10 J = 1,N
 10         A(J) = A(J) + B(J,I)
```

Given $C_l = 8, C_h = 1$ and $C_m = 8$, the cost of the `I`-loop in the innermost position is $0 + 0 + 9 = 9$ cycles/iteration and the cost of the `J`-loop is $3*(1+\frac{1}{8}*8) = 6$ cycles/iteration. This cost model would argue for an `I,J` loop ordering. However, if C_h were 3 cycles, the cost of the `I`-loop becomes 11 cycles/iteration

```
Procedure LoopOrder($L$)

Input: $L$ = loop nest

  compute memory costs for each loop
  $\mathcal{P}$ = SortByCost($L$)
  if $\exists$ a dependence with a negative threshold then
    $\mathcal{L} = \mathcal{P}$
    for $i = 1$ to $|\mathcal{L}|$ then
      $\mathcal{P}_i$ = the outermost loop, $l_j \in \mathcal{L}$ |
          putting $l_j$ at level $i$ will not create a dependence
          with a negative threshold
      remove $\mathcal{P}_i$ from $\mathcal{L}$
    enddo
  endif
  store $\mathcal{P}$ for this perfectly nested loop
end
```

Figure 4.2 Algorithm for Ordering Loops

and the cost of the J-loop becomes 12 cycles/iteration, giving a J,I loop ordering. As illustrated by this example, our cost model can optimize for a loop/architecture combination that calls for either better cache reuse or better register reuse. Unfortunately, we do not have access to a machine with a high load penalty to validate this effect of our model. It may be the case that TLB misses, due to long strides for references with no reuse, would dominate the cost of memory accesses. In this case, our model would need to be updated to reflect TLB performance.

4.2.2 Non-Perfectly Nested Loops

It may be the case that the loop nest on which we wish to perform interchange is not perfectly nested. Consider the following loop.

```
      DO 10 I = 1,N
         DO 20 J = 1,N
20          A(I,J) = A(I,J) + B(I,J)
         DO 10 J = 1,N
10          C(J,I) = C(J,I) + D(J,I)
```

It is desirable to interchange the I- and J-loops for statement 20, but is not desirable for statement 10. To handle this situation, we will use an approach similar to that used in unroll-and-jam, where each innermost loop body will be considered as if it were perfectly nested when computing memory costs. If interchange across a non-perfectly nested portion of the loop nest is required or different loop orderings of common loops are requested, then the interchanged loop is distributed, when safe, and the interchange is performed as desired [Wol86a]. Distribution safety can be incorporated into *LoopOrder* at the point where interchange safety is tested. To be conservative, any loop containing a distribution-preventing recurrence and any loop nested outside of that loop cannot be interchanged inward. In the previous example, the result after distribution and interchange would be

```
      DO 20 J = 1,N
         DO 20 I = 1,N
20          A(I,J) = A(I,J) + B(I,J)
      DO 10 I = 1,N
         DO 10 J = 1,N
10          C(J,I) = C(J,I) + D(J,I)
```

In Figure 4.3, we show the complete algorithm for ordering nested loops.

4.3 Experiment

We have implemented our loop interchange algorithm in the Parascope programming environment along with scalar replacement and unroll-and-jam. Our experiment was performed on the IBM RS/6000 model 540 which has the following cache parameters: $C_h = 1, C_m = 8$ and $C_l = 128$ bytes. The order in which transformations are performed in our system is loop interchange followed by unroll-and-jam and scalar replacement. First, we get the best data locality in the innermost loop and then we improve the resulting balance.

Matrix Multiply. We begin our study with the effects of loop order on matrix multiply using matrices too large for cache to capture all reuse. In the table below, the performance of the various versions using two different-sized matrices is given.

Loop Order	Array Size	Time
IJK	300x300	12.51s
	500x500	131.71s
IKJ	300x300	50.93s
	500x500	366.77s
JIK	300x300	12.46s
	500x500	135.61s
JKI	300x300	3.38s
	500x500	15.49s
KIJ	300x300	51.32s
	500x500	366.62s
KJI	300x300	3.45s
	500x500	15.83s

```
Procedure Interchange($L, i$)

Input: $L$ = loop nest
        $i$ = level of outermost loop

  for $N$ = each possible perfect loop nest in $L$ do
    LoopOrder($N$)
  InterchangeWithDistribution($L$, $i$)
end

Procedure InterchangeWithDistribution($L$, $i$)

  if $L$ has multiple inner loops at level $i + 1$ then
    if interchange is desired across level $i + 1$ or
        multiple orderings requested for levels 1 to $i$ then
      distribute $L$
      for $N = L$ and its distributed copies do
        Interchange($N, i$)
      return
    endif
  if $L$ is innermost then
    let $N$ = ordering for perfect nest produced by LoopOrder
    order perfect nest of loops by $N$
  endif
  $N$ = the first inner loop of $L$ at level $i + 1$
  if $N$ exists then
    InterchangeWithDistribution($N$, $i + 1$)
  $N$ = the next loop at level $i$ following $L$
  while $N$ exists do
    InterchangeWithDistribution($N$, $i$)
    $N$ = the next loop at level $i$ following $N$
  enddo
end
```

Figure 4.3 Algorithm for Non-Perfectly Nested Loops

The loop order with the best performance is JKI because of stride-one access in cache. This is the loop order that our transformation system will derive given any of the possible orderings. The key observation to make is that the programmer can specify matrix multiply in the manner that is most understandable to him without concern for memory performance. Using our technique, the compiler will give the programmer good performance without requiring knowledge of the implementation of the array storage layout for a particular programming language nor the underlying machine. In essence, machine-independent programming is made possible.

NAS Kernels. Next, we studied the effects of loop interchange on a couple of the Nas kernels from the SPEC benchmark suite. As is shown in the results below, getting the proper loop ordering for memory performance can have a dramatic effect.

Kernel	Original	SR	LI+SR	UJ+SR	LI+UJ+SR	Speedup
Vpenta	149.68s	149.68s	115.62s	138.69s	115.62s	1.29
Gmtry	155.30s	154.74s	17.89s	25.95s	18.07s	8.68

The speedup reported for Gmtry does not include the application of unroll-and-jam. Although we have no analysis tool to determine what exactly caused the slight degradation in performance, the most likely suspect is cache interference. Unroll-and-jam probably created enough copies of the inner loop to incur cache interference problems due to set associativity.

Applications. To complete our study, we ran our set of applications through the translator to determine the effectiveness of loop interchange. In the following table, the results are shown for those applications where loop interchange was applicable.

Program	Original	SR	LI+SR	UJ+SR	LI+UJ+SR	Speedup
Arc2d	410.13s	407.57s	190.69s	401.96	192.53s	2.15
Simple	963.20s	934.13s	850.18s	928.84s	847.82s	1.14
Wave	445.94s	431.11s	414.63s	431.11s	414.63s	1.08

Again, we have shown that remarkable speedups are attainable with loop interchange. Arc2d improved by a factor of over 2 on a single processor. This application was written in a "vectorizable" style with regard for vector operations rather than data locality. What our transformation system has done is to allow the code to be portable across different architectures and still attain good performance. The very slight degradation when unroll-and-jam was added is probably again due to cache interference.

Throughout our study, one factor stood out as the key to performance on machines like the RS6000 — stride-one access in the cache. Long cache lines, a low access cost and a high miss penalty create this phenomena. Attaining spatial reuse was a simple yet extremely effective approach to obtaining high performance.

4.4 Summary

In this chapter, we have shown how to automatically order loops in a nest to give good memory performance. The model that we have presented for memory performance is simple, but extremely effective. Using our transformation system, the programmer is freed from having to worry about his loop order, allowing him to write his code in a machine-independent form with confidence that it will be automatically optimized to achieve good memory performance. The results presented here represent a positive step toward encouraging machine-independent programming.

Chapter 5

Blockability

In Chapter 1, we presented two transformations, unroll-and-jam and strip-mine-and-interchange, to effect iteration-space blocking. Although these transformations have been studied extensively, they have mainly been applied to kernels written in an easily analyzed algorithmic style. In this chapter, we examine the applicability of iteration-space blocking on more complex algorithmic styles. Specifically, we present the results of a project to see if a compiler could automatically generate block algorithms similar to those found in LAPACK from the corresponding point algorithms expressed in Fortran 77. In performing this study, we address the question, "What information does a compiler need in order to derive block versions of real-world codes that are competitive with the best hand-blocked versions?"

In the course of this study, we have found transformation algorithms that can be successfully used on triangular loops, which are quite common in linear algebra, and trapezoidal loops. In addition, we have discovered an algorithmic approach that can be used to analyze and block programs that exhibit complex dependence patterns. The latter method has been successfully applied to block LU decomposition without pivoting. The key to many of these results is a transformation known as *index-set splitting*. Our results with this transformation show that a wide class of numerical algorithms can be automatically optimized for a particular machine's memory hierarchy even if they are expressed in their natural form. In addition, we have discovered that specialized knowledge about which operations commute with one another can enable compilers to block codes that were previously thought to be unblockable by automatic means.

This chapter begins with a review of iteration-space blocking. Next, the transformations that we found were necessary to block algorithms like those found in LAPACK are presented. Then, we present a study of the application of these transformations to derive the block LAPACK-like algorithms from their corresponding point algorithms. Finally, for those algorithms that cannot be blocked by a compiler, we propose a set of language extensions to allow the expression of block algorithms in a machine-independent form.

5.0.1 Iteration-Space Blocking

To improve the memory behavior of loops that access more data than can be handled by a cache, the iteration space of a loop can be blocked into sections whose temporal reuse can be captured by the cache. Strip-mine-and-interchange is a transformation that achieves this result [Wol87, WL91]. The effect is to shorten the distance between the source and sink of a dependence so that it is more likely for the datum to reside in cache when the reuse occurs. Consider the following loop nest.

```
      DO 10 J = 1,N
        DO 10 I = 1,M
10        A(I) = A(I) + B(J)
```

Assuming that the value of `M` is much greater than the size of the cache, we would get temporal reuse of the values of `B`, while missing the temporal reuse of the values of `A` on each iteration of `J`. To capture `A`'s reuse, we can use strip-mine-and-interchange as shown below.

```
      DO 10 J = 1,N,JS
        DO 10 I = 1,M
          DO 10 JJ = J, MIN(J+JS-1,N)
10          A(I) = A(I) + B(JJ)
```

Now, we can capture the temporal reuse of `JS` values of `B` out of cache for every iteration of the `J`-loop if `JS` is less that the size of the cache and no cache interference occurs, and we can capture the temporal reuse of `A` in registers [LRW91].

As stated earlier, both strip-mine-and-interchange and unroll-and-jam make up the transformation technique known as *iteration-space blocking*. Essentially, unroll-and-jam is strip-mine-and-interchange with the innermost loop unrolled. The difference is that unroll-and-jam is used to block for registers and strip-mine-and-interchange for cache.

5.1 Index-Set Splitting

Iteration-space blocking cannot always be directly applied as shown in the previous section. Sometimes a transformation called *index-set splitting* must be applied to allow blocking. Index-set splitting involves the creation of multiple loops from one original loop, where each new loop iterates over a portion of the original iteration space. Execution order is not changed and the original iteration space is still completely executed. As an example of index-set splitting, consider the following loop.

```
      DO 10 I = 1,N
10      A(I) = A(I) + B(I)
```

We can split the index set of `I` at iteration `100` to obtain

```
      DO 10 I = 1,MIN(N,100)
10      A(I) = A(I) + B(I)
      DO 20 I = MIN(N,100)+1,N
20      A(I) = A(I) + B(I)
```

Although this transformation does nothing by itself, its application can enable the blocking of complex loop forms. This section shows how index-set splitting allows the blocking of triangular-, trapezoidal-, and rhomboidal-shaped iteration spaces and the partial blocking of loops with complex dependence patterns.

5.1.1 Triangular Iteration Spaces

When the iteration space of a loop is not rectangular, iteration-space blocking cannot be directly applied. The problem is that when performing interchange of loops that iterate over a triangular region, the loop bounds must be modified to preserve the semantics of the loop [Wol86a, Wol87]. Below, we will derive the formula for determining loop bounds when blocking is performed on triangular iteration spaces. We begin with the derivation for strip-mine-and-interchange and then extend it to unroll-and-jam.

The general form of a strip-mined triangular loop is given below. α and β are integer constants (β may be a loop invariant) and $\alpha > 0$.

```
     DO 10 I = 1,N,IS
       DO 10 II = I,I+IS-1
         DO 10 J = $\alpha$II+$\beta$,M
10         loop body
```

Figure 5.1 gives a graphical description of the iteration space of this loop. To interchange the `II` and `J` loops, we have to account for the fact that the line `J=`α`II+`β intersects the iteration space at the point `(I,`α`I+`β`)`. Therefore, when we interchange the loops, the `II`-loop must iterate over a trapezoidal region, requiring its upper bound to be $\frac{J-\beta}{\alpha}$ until $\frac{(J-\beta)}{\alpha} >$ `I+IS-1`. This gives the following loop nest.

```
     DO 10 I = 1,N,IS
       DO 10 J = $\alpha$I+$\beta$,M
         DO 10 II = I,MIN((J-$\beta$)/$\alpha$,I+IS-1)
10         loop body
```

This formula can be trivially extended to handle the cases where $\alpha < 0$ and where a linear function of `I` appears in the upper bound instead of the lower bound (see Appendix A).

Given the formula for triangular strip-mine-and-interchange, we can extend it to triangular unroll-and-jam as follows. The iteration space defined by the two inner loops is a trapezoidal region, making unrolling the innermost loop non-trivial because the number of iterations vary with `J`. To overcome this, we can use index-set splitting on `J` to create one loop that iterates over the triangular region below the line `J=`α`(I+IS-1)+`β

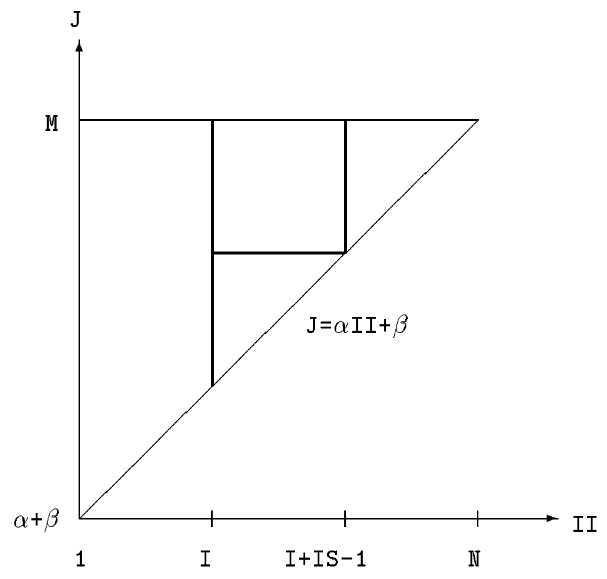

Figure 5.1 Upper Left Triangular Iteration Space

and one loop that iterates over the rectangular region above the line. Since we know the length of the rectangular region, the second loop can be unrolled to give the following loop nest.

```
    DO 10 I = 1,N,IS
      DO 20 II = I,I+IS-2
        DO 20 J = αII+β,MIN(α(I+IS-2)+β,M)
20        loop body
      DO 10 J = α(I+IS-1)+β,M
10      unrolled loop body
```

Depending upon the values of α and β, it may also be possible to determine the size of the triangular region; therefore, it may be possible to completely unroll the first loop nest to eliminate the overhead. Additionally, triangular unroll-and-jam can be trivially extended to handle other common triangles (see Appendix A).

To see the potential of triangular unroll-and-jam, consider the following loop that is used in a back solve after LU decomposition.

```
      DO 10 I = 1,N
        DO 10 J = I+1,N
10        A(J) = A(J) + B(J,I) * A(I)
```

We used an automatic system to perform triangular unroll-and-jam and scalar replacement on the above loop. We then ran the result on arrays of DOUBLE-PRECISION REALS on an IBM RS/6000 540. The results are shown in the table below.

Size	Iterations	Original	Xformed	Speedup
300	500	6.09s	3.49s	1.74
500	200	6.78s	3.82s	1.77

5.1.2 Trapezoidal Iteration Spaces

While the previous method applies to many of the common non-rectangular-shaped iteration spaces, there are still some important loops that it will not handle. In linear algebra, seismic and partial differential equation codes, we often find loops with trapezoidal-shaped iteration spaces. Consider the following example, where L is assumed to be a constant and $\alpha > 0$.

```
    DO 10 I = 1,N
      DO 10 J = L,MIN(αI+β,N)
10      loop body
```

Here, as is often the case, a MIN function has been used to handle boundary conditions, resulting in the loop's trapezoidal shape as shown in Figure 5.2. The formula to handle a triangular region does not apply in this case, so we must extend it.

The trapezoidal iteration space contains one rectangular region and one triangular region separated at the point where αI+β = N. Because we already know how to handle rectangular and triangular regions and because we want to reduce the execution overhead in a rectangular region, we can split the index set of I into two separate regions at the point I = $\frac{N-\beta}{\alpha}$. This gives the following loop nest.

```
   DO 10 I = 1,MIN(N,(N-β)/α)
     DO 10 J = L,αI+β
 10    loop body
   DO 10 I = MAX(1,MIN(N,(N-β)/α)+1),N
     DO 10 J = L,N
 10    loop body
```

We can now apply triangular iteration-space blocking to the first loop and rectangular unroll-and-jam to the second loop. In addition to loops with MIN functions in the upper bound, index-set splitting of trapezoidal regions can be extended to allow MAX functions in the lower bound of the inner loop (see Appendix A).

The lower bound, L, of the inner loop in the trapezoidal nest need not be restricted to a constant value. It can essentially be any function that produces an iteration space that can be blocked. In the loop,

```
   DO 10 I = 1,N1
     DO 10 J = I,MIN(I+N2,N3)
 10    F3(I) = F3(I)+DT*F1(K)*WORK(I-K)
```

the lower bound is a linear function of the outer-loop induction variable, resulting in rhomboidal and triangular regions (see Figure 5.3). To handle this loop, blocking can be extended to rhomboidal regions using index-set splitting as in the case for triangular regions (see Appendix A).

To see the potential for improving the performance of trapezoidal loops, see Section 3.5 under **Geophysics Kernels**. The algorithm Afold is the example shown above and computes the adjoint-convolution of two time series. Fold computes the convolution of two time series and contains a MAX function in the lower bound and a MIN function in the upper bound.

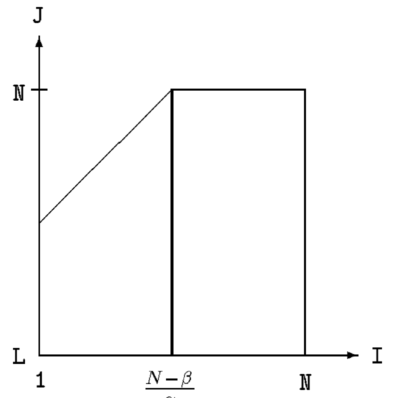

Figure 5.2 Trapezoidal Iteration Space with Rectangle

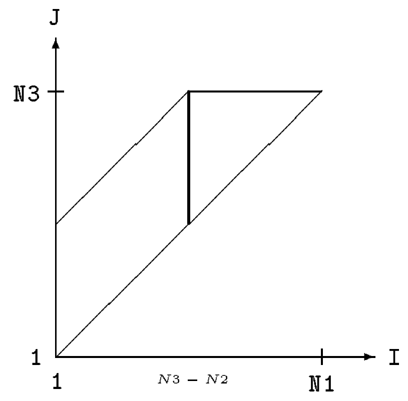

Figure 5.3 Trapezoidal Iteration Space with Rhomboid

5.1.3 Complex Dependence Patterns

In some cases, it is not only the shape of the iteration space that presents difficulties for the compiler, but also the dependence patterns within the loop. Consider the strip-mined example below.

```
      DO 10 I = 1,N,IS
        DO 10 II = I, I+IS-1
          T(II) = A(II)
          DO 10 K = II,N
10          A(K) = A(K) + T(II)
```

To complete blocking, the `II`-loop must be interchanged into the innermost position. Unfortunately, there is an recurrence between the definition of `A(K)` and the load from `A(II)` carried by the `II`-loop. Using only standard dependence abstractions, such as distance and direction vectors, we would be prevented from blocking the loop [Wol82]. However, if we analyze the sections of the arrays that are accessed at the source and sink of the offending dependence using array summary information, the potential to apply blocking is revealed [CK87, HK91]. Consider Figure 5.4. The region of the array `A` read by the reference to `A(II)` goes from `I` to `I+IS-1` and the region written by `A(K)` goes from `I` to `N`. Therefore, the recurrence does not exist for the region from `I+IS` to `N`.

To allow partial blocking of the loop, we can split the index set so that one loop iterates over the common region and one loop iterates over the disjoint region. To determine the split point to create these regions, we set the subscript expression for the larger region equal to the boundary between the common and disjoint regions and solve for the inner induction variable. In our example, we let `K` = `I+IS-1` and solve for `K`. Splitting at this point yields

```
      DO 10 I = 1,N,IS
        DO 10 II = I,I+IS-1
          T(II) = A(II)
          DO 20 K = I,I+IS-1
20          A(K) = A(K) + T(II)
          DO 10 K = I+IS,N
10          A(K) = A(K) + T(II)
```

We can now distribute the `II`-loop and complete blocking on the loop nest surrounding statement `10`.

The method that we have just described may be applicable when the references involved in the preventing dependences have different induction variables in corresponding positions (*e.g.*, `A(II)` and `A(K)` in the previous example). An outline of our application of the method after strip mining is given below.

1. Calculate the sections of the source and sink of the preventing dependence.
2. Intersect and union the sections.

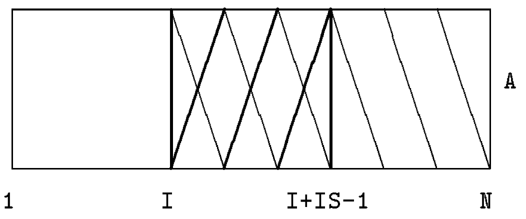

Figure 5.4 Data Space for `A`

3. If the intersection is equal to the union then stop.
4. Set the subscript expression of the larger section equal to the boundary between the disjoint and common sections and solve for the inner-loop induction variable.
5. Split the index set of the inner loop at this point and proceed with blocking on the disjoint region.
6. Repeat steps 4 and 5 if there are multiple boundaries.

If multiple transformation-preventing dependences exist, we process each one with the above steps until failure or a region is created where blocking can be performed.

5.2 Control Flow

In addition to iteration-space shapes and dependence patterns, we must also consider the effects of control flow on blocking. It may be the case that an inner loop is guarded by an `IF`-statement to prevent unnecessary computation. Consider the following matrix multiply code.

```
      DO 10 J = 1,N
        DO 10 K = 1,N
          IF (B(K,J) .NE. 0.0) THEN
            DO 20 I = 1,N
20            C(I,J) = C(I,J) + A(I,K) * B(K,J)
          ENDIF
10    CONTINUE
```

If we were to ignore the `IF`-statement and perform unroll-and-jam on the `K`-loop, we would obtain the following code.

```
      DO 10 J = 1,N
        DO 10 K = 1,N,2
          IF (B(K,J) .NE. 0.0) THEN
            DO 20 I = 1,N
              C(I,J) = C(I,J) + A(I,K) * B(K,J)
20            C(I,J) = C(I,J) + A(I,K+1) * B(K+1,J)
          ENDIF
10    CONTINUE
```

Here, the value of `B(K+1,J)` is never tested and statements that were not executed in the original code may be executed in the unrolled code. Thus, we have performed unroll-and-jam illegally.

One possible method to preserve correctness is to move the guard into the innermost loop and replicate it for each unrolled iteration. This, however, would result in a performance degradation due to a decrease in loop-level parallelism and an increase in instructions executed. Instead, we can use a combination of `IF`-conversion and sparse-matrix techniques that we call `IF`-inspection to allow us to keep the guard out of the innermost loop and still allow blocking [AK87]. The idea is to inspect at run-time the values of an outer-loop induction variable for which the guard is true and the inner loop is executed. Then, we execute the loop nest for only those values. To effect IF-inspection, code is inserted within the IF-statement to record loop bounds information for the loop that we wish to transform. On the true branch of the guard to be inspected we insert the following code, where KC is initialized to 1, FLAG is initialized to false, K is the induction variable of the loop to be inspected and KLB is the store for the lower bound of an executed range.

```
IF (.NOT. FLAG) THEN
  KC = KC + 1
  KLB(KC) = K
  FLAG = .TRUE.
ENDIF
```

On the false branch of the inspected guard, we insert the following code to store the upper bound of each executed range.

```
IF (FLAG) THEN
  KUB(KC) = K-1
  FLAG = .FALSE.
ENDIF
```

Note that we must also account for the fact that the value of the guard could be true on the last iteration of the loop, requiring a test of FLAG to store the upper bound of the last range after the IF-inspection loop body.

After inserting the inspection code, we can distribute the loop we wish to transform around the inspection code and create a new loop nest that executes over the iteration space where the innermost loop was executed. In our example, the result would be

```
      DO 20 K = 1,N
C
C     IF-inspection code
C
 20   CONTINUE
      DO KN = 1,KC
        DO K = KLB(KN),KUB(KN)
          DO I = 1,N
 10         C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

The KN-loop executes over the number of ranges where the guarded loop is executed and the K-loop executes within those ranges. The new K-loop can be transformed as was desired in the original loop nest. The final IF-inspected code for our example is shown in Figure 5.5

If the ranges over which the inner loop is executed in the original loop are large, the increase in run-time cost caused by IF-inspection can be more than counteracted by the improvements in performance on the newly transformed inner loop. To show this, we performed unroll-and-jam on our IF-inspected matrix multiply example and ran it on an IBM RS/6000 model 540 on 300x300 arrays of REALS. In the table below, Frequency shows how often B(K,J) = 0, UJ is the result of performing unroll-and-jam after moving the guard into the innermost loop and UJ+IF is the result of performing unroll-and-jam after IF-inspection.

Frequency	Original	UJ	UJ+IF	Speedup
2.5%	3.33s	3.84s	2.25s	1.48
10%	3.08s	3.71s	2.13s	1.45

5.3 Solving Systems of Linear Equations

LAPACK is a project whose goal is to replace the algorithms in LINPACK and EISPACK with block algorithms that have better cache performance. Unfortunately, scientists have spent years developing this package to attain high performance on a variety of architectures. We believe that this process is the wrong direction for high-performance computing. Compilers, not programmers, should handle the machine-specific details required to attain high performance. It should be possible for algorithms to be expressed in a natural, machine-independent form with the compiler performing the machine-specific optimizations to attain performance.

```
      FLAG = .FALSE.
      DO 10 J = 1,N
       KC = 0
       DO 20 K = 1,N
         IF (B(K,J) .NE. 0.0) THEN
           IF (.NOT. FLAG) THEN
             KC = KC + 1
             KLB(KC) = K
             FLAG = .TRUE.
           ENDIF
         ELSE
           IF (FLAG) THEN
             KUB(KC) = K-1
             FLAG = .FALSE.
           ENDIF
         ENDIF
20     CONTINUE
       IF (FLAG) THEN
         KUB(KC) = N
         FLAG = .FALSE.
       ENDIF
       DO 10 KN = 1,KC
         DO 10 K = KLB(KN),KUB(KN)
           DO 10 I = 1,N
10           C(I,J) = C(I,J) + A(I,K) * B(K,J)
```

Figure 5.5 Matrix Multiply After `IF`-Inspection

In this section, we examine the efficacy of this hypothesis by studying the machine-independent expression of algorithms similar to those found in LAPACK. We examine the blockability of three algorithms for solving systems of linear equations, where an algorithm is "blockable" if a compiler can automatically derive the *best* known block algorithm, similar to the one found in LAPACK, from its corresponding machine-independent point algorithm. Deriving the block algorithms poses two main problems. The first is developing the transformations that allow the compiler to attain the block algorithm from the point algorithm. The second is to determine the machine-dependent blocking factor for the block algorithm. In this section, we address the first issue. The second is beyond the scope of this thesis.

Our study shows that LU decomposition without pivoting is a blockable algorithm using the techniques derived in Section 5.1, LU decomposition with partial pivoting is blockable if information in addition to the index-set splitting techniques is supplied to the compiler and QR decomposition with Householder transformations is not blockable. The study will also show how to improve the memory performance of a fourth non-LAPACK algorithm, QR decomposition with Givens rotations using the techniques of Sections 5.1 and 5.2.

5.3.1 LU Decomposition without Pivoting

Gaussian elimination is a form of LU decomposition where the matrix A is decomposed into two matrices, L and U, such that

$$A = LU,$$

L is a unit lower triangular matrix and U is an upper triangular matrix. This decomposition can be obtained by multiplying the matrix A by a series of elementary lower triangular matrices, $M_k \dots M_1$, as follows [Ste73].

$$\begin{aligned} A &= LU \\ A &= M_1^{-1} \dots M_k^{-1} U \\ U &= M_k \dots M_1 A \end{aligned} \tag{5.1}$$

Using Equation 5.1, an algorithm for LU decomposition without pivoting can be derived. This point algorithm, where statement 20 computes M_k and statement 10 applies M_k to A, is shown below.

```
      DO 10 K = 1,N-1
         DO 20 I = K+1,N
20          A(I,K) = A(I,K) / A(K,K)
         DO 10 J = K+1,N
            DO 10 I = K+1,N
10            A(I,J) = A(I,J) - A(I,K) * A(K,J)
```

Unfortunately, this algorithm exhibits poor cache performance on large matrices. To improve its cache performance, scientists have developed a block algorithm that essentially groups a number of updates to the matrix A and applies them together to a block portion of the array [DDSvdV91]. To attain the best blocking, strip-mine-and-interchange is performed on the outer K-loop for only a portion of the inner loop nest, requiring the technique described in Section 5.1.3 for automatic derivation of the block algorithm. Consider the strip-mined version of LU decomposition below.

```
      DO 10 K = 1,N-1,KS
         DO 10 KK = K,K+KS-1
            DO 20 I = KK+1,N
 20            A(I,KK) = A(I,KK)/A(KK,KK)
            DO 10 J = KK+1,N
               DO 10 I = KK+1,N
 10               A(I,J) = A(I,J) - A(I,KK) * A(KK,J)
```

To complete the blocking of this loop, the KK-loop would have to be distributed around the loop that surrounds statement 20 and around the loop nest that surrounds statement 10 before being interchanged to

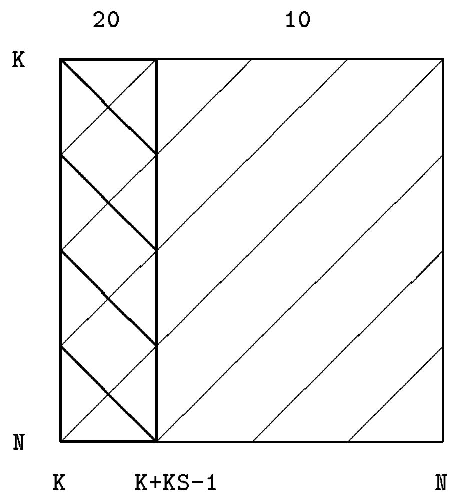

Figure 5.6 Regions Accessed in LU Decomposition

the innermost position. However, there is a recurrence between statements 20 and 10 carried by the KK-loop that prevents distribution unless index-set splitting is done.

If we analyze the regions of the array A accessed for the entire execution of the KK-loop, we find that the region touched by statement 20 is a subset of the region touched by statement 10 (Figure 5.6 gives a graphical description of the data regions). Since the recurrence exists for only a portion of the iteration space, we can split the larger region, defined by the reference to A(I,J) in statement 10, at the point J = K+KS-1. The new loop that covers the disjoint region is shown below.

```
      DO 10 KK = K,K+KS-1
        DO 10 J = K+KS,N
          DO 10 I = KK+1,N
10          A(I,J) = A(I,J) - A(I,KK) * A(KK,J)
```

Now, we can use triangular interchange to put the KK-loop in the innermost position. At this point, we have obtained the best block algorithm, making LU decomposition blockable (see Figure 5.7). Not only does this block algorithm exhibit better data locality, it also has increased parallelism as the J-loop that surrounds statement 10 can be made parallel.

At this point, we would like to perform unroll-and-jam and scalar replacement to further improve the performance of block LU decomposition. Unfortunately, the true dependence from A(I,J) to A(KK,J) in

```
      DO 10 K = 1,N-1,KS
        DO 20 KK = K,MIN(K+KS-1,N-1)
          DO 30 I = KK+1,N
30          A(I,KK) = A(I,KK)/A(KK,KK)
          DO 20 J = KK+1,K+KS-1
            DO 20 I = KK+1,N
20            A(I,J) = A(I,J) - A(I,KK) * A(KK,J)
         DO 10 J = K+KS,N
           DO 10 I =K+1,N
             DO 10 KK = K,MIN(MIN(K+KS-1,N-1),I-1)
10             A(I,J) = A(I,J) - A(I,KK) * A(KK,J)
```

Figure 5.7 Block LU Decomposition

statement `10` is inconsistent and when moved inward by unroll-and-jam would prevent code motion of the assignment to `A(I,J)`. However, if we examine the array sections of the source and sink of the dependence after splitting for trapezoidal and triangular regions, we find that the dependence does not exist in an unrolled loop body.

We applied our algorithm by hand to LU decomposition and compared its performance with the original program and a hand coded version of the right-looking algorithm [DDSvdV91]. In the table below, "Block 1" refers to the right-looking version and "Block 2" refers to our algorithm in Figure 5.7. In addition, we used our automatic system to perform trapezoidal unroll-and-jam and scalar replacement, to our blocked code, producing the version referred to as "Block 2+".* The experiment was run on an IBM RS/6000 model 540 using DOUBLE-PRECISION REALS. The reader should note that these final transformations could have been applied to the Sorensen version as well, with similar improvements.

Array Size	Block Size	Original	Block 1	Block 2	Block 2+	Speedup
300x300	32	1.47s	1.37s	1.35s	0.49s	3.00
300x300	64	1.47s	1.42s	1.38s	0.58s	2.53
500x500	32	6.76s	6.58s	6.44s	2.13s	3.17
500x500	64	6.76s	6.59s	6.38s	2.27s	2.98

5.3.2 LU Decomposition with Partial Pivoting

Although the compiler can discover the potential for blocking in LU decomposition without pivoting using dependence information, the same cannot be said when partial pivoting for numerical stability is added to the algorithm. Using the following matrix formulation

$$U = M_{n-1}P_{n-1}\cdots M_3P_3M_2P_2M_1P_1A,$$

the point version that includes partial pivoting can be derived (see Figure 5.8) [Ste73].* While we can apply index-set splitting to the algorithm in Figure 5.8 after strip mining to break the recurrence carried by the new `KK`-loop involving statement `10` and statement `40` as in the previous section, we cannot break the recurrence involving statements `10` and `25` using this technique.

After index-set splitting, we have the following relevant sections of code.

```
      DO 10 KK = K,K+KS-1
C   ...
         DO 30 J = 1,N
            TAU = A(KK,J)
 25         A(KK,J) = A(IMAX,J)
 30         A(IMAX,J) = TAU
C   ...
         DO 10 J = K+KS,N
            DO 10 I = KK+1,N
 10            A(I,J) = A(I,J) - A(I,KK) * A(KK,J)
```

Distributing the `KK`-loop around both `J`-loops would convert what was originally a true dependence from `A(I,J)` in statement `10` to `A(IMAX,J)` in statement `25` into an anti-dependence in the reverse direction. The rules for the preservation of data dependence prohibit the reversing of a dependence direction, which would seem to preclude the existence of a block analogue similar to the non-pivoting case. However, a block algorithm, that essentially ignores the preventing recurrence and is similar to the non-pivoting case, can still be mathematically derived using the following result from linear algebra (see Figure 5.9) [DDSvdV91, Ste73]. If we have

$$M_1 = \begin{pmatrix} 1 & 0 \\ -m_1 & I \end{pmatrix}, \quad P_2 = \begin{pmatrix} 1 & 0 \\ 0 & \hat{P}_2 \end{pmatrix}$$

*The dependence from `A(I,J)` to `A(KK,J)` in statement `10` needed to be deleted in order for our system to work. Not only do we need section analysis to handle the dependence, but also PFC incorrectly reported the dependence as interchange preventing.

*In this version of the algorithm, row interchanges are performed across every column. Although this is not necessary (it can be done only for columns `K` through `N`), it is done to allow the derivation of the block algorithm.

```
      DO 10 K = 1,N-1
         TAU = ABS(A(K,K))
         IMAX = K
         DO 20 I = K+1,N
            IF (ABS(A(I,K)) .LE. TAU) GOTO 20
            IMAX = I
            TAU = ABS(A(I,K))
20       CONTINUE
         DO 30 J = 1,N
            TAU = A(K,J)
25          A(K,J) = A(IMAX,J)
30          A(IMAX,J) = TAU
         DO 40 I = K+1,N
40          A(I,K) = A(I,K)/A(K,K)
         DO 10 J = K+1,N
            DO 10 I = K+1,N
10             A(I,J) = A(I,J) - A(I,K) * A(K,J)
```

Figure 5.8 LU Decomposition with Partial Pivoting

```
      DO 10 K = 1,N-1,KS
        DO 20 KK = K,MIN(K+KS-1,N-1)
          TAU = ABS(A(K,K))
          IMAX = K
          DO 30 I = K+1,N
            IF (ABS(A(I,K)) .LE. TAU) GOTO 20
            IMAX = I
             TAU = ABS(A(I,K))
30        CONTINUE
          DO 40 J = 1,N
            TAU = A(KK,J)
25          A(KK,J) = A(IMAX,J)
40          A(IMAX,J) = TAU
         DO 50 I = KK+1,N
50          A(I,K) = A(I,KK)/A(KK,KK)
          DO 20 J = KK+1,K+KS-1
            DO 20 I = KK+1,N
20            A(I,J) = A(I,J) - A(I,KK) * A(KK,J)
         DO 10 J = K+KS,N
           DO 10 I =K+1,N
             DO 10 KK = K,MIN(MIN(K+KS-1,N-1),I-1)
10             A(I,J) = A(I,J) - A(I,KK) * A(KK,J)
```

Figure 5.9 Block LU Decomposition with Partial Pivoting

then

$$\begin{aligned}
P_2M_1 &= \begin{pmatrix} 1 & 0 \\ 0 & \hat{P}_2 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ -m_1 & I \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ -\hat{P}_2 m_1 & \hat{P}_2 \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 \\ -\hat{P}_2 m_1 & I \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & \hat{P}_2 \end{pmatrix} \\
&= \hat{M}_1 P_2. \qquad (5.2)
\end{aligned}$$

This result shows that we can postpone the application of the eliminator M_1 until after the application of the permutation matrix P_2 if we also permute the rows of the eliminator. Extending Equation 5.2 to the entire formulation we have

$$\begin{aligned}
U = &\; M_{n-1}P_{n-1}M_{n-2}P_{n-2}M_{n-3}P_{n-3}\cdots M_1P_1A \\
= &\; M_{n-1}\hat{M}_{n-2}P_{n-1}P_{n-2}M_{n-3}P_{n-3}\cdots M_1P_1A \\
= &\; M_{n-1}\hat{M}_{n-2}\hat{M}_{n-3}P_{n-1}P_{n-2}P_{n-3}\cdots M_1P_1A \\
= &\; M_{n-1}\hat{M}_{n-2}\hat{M}_{n-3}\cdots\hat{M}_1P_{n-1}P_{n-2}P_{n-3}\cdots P_1A.
\end{aligned}$$

In the implementation of the block algorithm, P_i cannot be computed until step i of the point algorithm. P_i only depends upon the first i columns of A, allowing the computation of k P_i's and $\hat{M}_i$'s, where k is the blocking factor, and then the block application of the $\hat{M}_i$'s as is done in Figure 5.9 [DDSvdV91].

To install the above result into the compiler, we examine its implications from a data dependence viewpoint. In the point version, each row interchange is followed by a whole-column update in which each row element is updated independently. In the block version, multiple row interchanges may occur before a particular column is updated. The same computations (column updates) are performed in both the point and block versions, but these computations may occur in different locations (rows) of the array because of the application of P_{i+1} to M_i. The key concept for the compiler to understand is that row interchanges and whole-column updates are commutable operations. Data dependence alone is not sufficient to understand this. A data dependence relation maps values to memory locations. It reveals the sequence of values that pass through a particular location. In the block version of LU decomposition, the sequence of values that pass through a location is different from the point version, although the final values are identical. Therefore, from the point of view of a compiler that only understands data dependence, LU decomposition with partial pivoting is not blockable.

Fortunately, a compiler can be equipped to understand that operations on whole columns are commutable with row permutations. To upgrade the compiler, one would have to install pattern matching to recognize both the row permutations and whole-column updates to prove that the recurrence involving statements `10` and `25` of the index-set split code could be ignored. Forms of pattern matching are already done in commercially available compilers, so it is reasonable to believe that we can recognize the situation in LU decomposition. The question is, however, "Will the increase in knowledge be profitable?" To see the potential profitability of making the compiler more sophisticated, consider the table below, where "Block" refers to the algorithm given in Figure 5.9 and "Block+" refers to that algorithm after unroll-and-jam and scalar replacement. This experiment was run on an IBM RS/6000 model 540 using DOUBLE-PRECISION REALS.

Array Size	Block Size	Original	Block	Block+	Speedup
300x300	32	1.52s	1.42s	0.58s	2.62
300x300	64	1.52s	1.48s	0.67s	2.27
500x500	32	7.01s	6.85s	2.58s	2.72
500x500	64	7.01s	6.83s	2.73s	2.57

5.3.3 QR Decomposition with Householder Transformations

The key to Gaussian elimination is the multiplication of the matrix A by a series of elementary lower triangular matrices that introduce zeros below each diagonal element. Any class of matrices that have this property can be used to solve a system of linear equations. One such class, having orthonormal columns, is used in QR decomposition [Ste73].

If A has linearly independent columns, then A can be written uniquely in the form

$$A = QR,$$

where Q has orthonormal columns, $QQ^T = I$ and R is upper triangular with positive diagonal elements. One class of matrices that fits the properties of Q is elementary reflectors or *Householder transformations* of the form $I - 2vv^T$.

The point algorithm for this form of QR decomposition consists of iteratively applying the elementary reflector $V_k = I - 2v_k v_k^T$ to A_k to obtain A_{k+1} for $k = 1, \ldots, n-1$. Each V_k eliminates the values below the diagonal in the kth column. For a more detailed discussion of the QR algorithm and the computation of V_k, see Stewart [Ste73].

Although pivoting is not necessary for QR decomposition, the best block algorithm is not an aggregation of the original algorithm. The block application of a number of elementary reflectors involves both computation and storage that does not exist in the original algorithm [DDSvdV91]. Given

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}.$$

The first step is to factor

$$\begin{pmatrix} A_{11} \\ A_{21} \end{pmatrix} = \begin{pmatrix} Q_{11} & Q_{12} \\ Q_{21} & Q_{22} \end{pmatrix} \begin{pmatrix} R_{11} \\ 0 \end{pmatrix},$$

and then solve

$$\begin{pmatrix} \hat{A}_{12} \\ \hat{A}_{22} \end{pmatrix} = \hat{Q} \begin{pmatrix} A_{12} \\ A_{22} \end{pmatrix},$$

where

$$\begin{aligned} \hat{Q} = &\ (I - 2v_1 v_1^T)(I - 2v_2 v_2^T) \cdots (I - 2v_b v_b^T) \\ = &\ I - 2VTV^T. \end{aligned}$$

The difficulty for the compiler comes in the computation of $I - 2VTV^T$ because it involves space and computation that did not exist in the original point algorithm. To illustrate this, consider the case where the block size is 2.

$$\begin{aligned} \hat{Q} = &\ (I - 2v_1 v_1^T)(I - 2v_2 v_2^T) \\ = &\ I - 2(v_1 v_2) \begin{pmatrix} 1 & (v_1^T v_2) \\ 0 & 1 \end{pmatrix} \begin{pmatrix} v_1^T \\ v_2^T \end{pmatrix} \end{aligned}$$

Here, the computation of the matrix

$$T = \begin{pmatrix} 1 & (v_1^T v_2) \\ 0 & 1 \end{pmatrix}$$

is not part of the original algorithm, making it is impossible to determine the computation of $\hat{Q}$ from the data dependence information.

The expression of this block algorithm requires the choice of a machine-dependent blocking factor. We know of no way to express this algorithm in a current programming language in a manner that would allow a compiler to automatically chose that factor. Can we enhance the expressibility of a language to allow block algorithms to be stated in a machine-independent form? One possible solution is to define looping constructs whose semantics allow the compiler complete freedom in choosing the blocking factor. In Section 5.4, we will address this issue.

5.3.4 QR Decomposition with Givens Rotations

Another form of orthogonal matrix that can be used in QR decomposition is the Givens rotation matrix [Sew90]. We currently know of no best block algorithm to derive, so instead we show that the index-set splitting technique described in Section 5.1.3 and `IF`-inspection have wider applicability.

Consider the Fortran code for Givens QR shown in Figure 5.10 (note that this algorithm does not check for overflow) [Sew90]. The references to `A` in the inner `K`-loop have a long stride between successive accesses, resulting in poor cache performance. Our algorithm from Chapter 4 would recommend interchanging the `J`-loop to the innermost position, giving stride-one access to the references to `A(J,K)` and making the references

```
      DO 10 L = 1,N
        DO 10 J = L+1,M
           IF (A(J,L) .EQ. 0.0) GOTO 10
              DEN = DSQRT(A(L,L)*A(L,L) + A(J,L)*A(J,L))
              C = A(L,L)/DEN
              S = A(J,L)/DEN
              DO 10 K = L,N
                 A1 = A(L,K)
                 A2 = A(J,K)
                 A(L,K) = C*A1 + S*A2
10               A(J,K) = -S*A1 + C*A2
```

Figure 5.10 QR Decomposition with Givens Rotations

to `A(L,K)` invariant with respect to the innermost loop. In this case, loop interchange would necessitate distribution of the `J`-loop around the `IF`-block and the `K`-loop. However, a recurrence consisting of a true and antidependence between the definition of `A(L,K)` and the use of `A(L,L)` seems to prevent distribution. Examining the regular sections for these references reveals that the recurrence only exists for the element `A(L,L)`, allowing index-set splitting of the `K`-loop at `L`, `IF`-inspection of the `J`-loop, distribution (with scalar expansion) and interchange as shown Figure 5.11 [KKP+81]. Below is a table of the results of the performance of Givens QR using DOUBLE-PRECISION REALS run on an IBM RS/6000 model 540.

Array Size	Original	Optimized	Speedup
300x300	6.86s	3.37s	2.04
500x500	84.0s	15.3s	5.49

5.4 Language Extensions

The examination of QR decomposition with Householder transformations has shown that some block algorithms cannot be derived by a compiler from their corresponding point algorithms. In order for us to maintain our goal of machine-independent coding styles, we need to allow the expression of these types of block algorithms in a machine-independent form. Specifically, we need to direct the compiler to pick the machine-dependent blocking factor for an algorithm automatically.

To this end, we present a preliminary proposal for two looping constructs to guide the compiler's choice of blocking factor. These constructs are **`BLOCK DO`** and **`IN DO`**. **`BLOCK DO`** specifies a `DO`-loop whose blocking factor is chosen by the compiler. **`IN DO`** specifies a `DO`-loop that executes over the region defined by a corresponding **`BLOCK DO`** and guides the compiler to the regions that it should analyze to determine the blocking factor. The bounds of an **`IN DO`** statement are optional. If they are not expressed, the bounds are assumed to start at the first value in the specified block and end at the last value with a step of 1. To allow indexing within a block region, we define **`LAST`** to return the last index value in a block. For example, if LU decomposition were not a blockable algorithm, it could be coded as in Figure 5.12 to achieve machine independence.

The principal advantage of the extensions is that the programmer can express a non-blockable algorithm in a natural block form, while leaving the machine-dependent details, namely the choice of blocking factor, to the compiler. In the case of LAPACK, the language extensions could be used, when necessary, to code the algorithms for a source-level library that is independent of the choice of blocking factor. Then, using compiler technology, the library could be ported from machine to machine and still retain good performance. By doing so, we would remove the only machine-dependency problem of LAPACK and make it more accessible

```
          DO 10 L = 1,N
            DO 20 J = L+1,M
               IF (A(J,L) .EQ. 0.0) GOTO 20
                  DEN = DSQRT(A(L,L)*A(L,L) + A(J,L)*A(J,L))
                  C(J) = A(L,L)/DEN
                  S(J) = A(J,L)/DEN
                  A1 = A(L,L)
                  A2 = A(J,L)
                  A(L,L) = C(J)*A1 + S(J)*A2
                  A(J,L) = -S(J)*A1 + C(J)*A2
C
C         IF-Inspection Code
C
               ENDIF
 20         CONTINUE
            DO 10 K = L+1,N
              DO 10 JN = 1,JC
                DO 10 J = JLB(JN),JUB(JN)
                   A1 = A(L,K)
                    A2 = A(J,K)
                    A(L,K) = C(J)*A1 + S(J)*A2
 10                 A(J,K) = -S(J)*A1 + C(J)*A2
```

Figure 5.11 Optimized QR Decomposition with Givens Rotations

```
BLOCK DO K = 1,N-1
  IN K DO KK
    DO I = KK+1,N
      A(I,KK) = A(I,KK)/A(KK,KK)
    ENDDO
    DO J = KK+1,LAST(K)
      DO I = KK+1,N
        A(I,J) = A(I,J) - A(I,KK) * A(KK,J)
      ENDDO
    ENDDO
  ENDDO
  DO J = LAST(K)+1,N
    DO I = K+1,N
      IN K DO KK = K,MIN(LAST(K),I-1)
        A(I,J) = A(I,J) - A(I,KK) * A(KK,J)
      ENDDO
    ENDDO
  ENDDO
ENDDO
```

Figure 5.12 Block LU in Extended Fortran

for new architectures. To realize this goal, research efforts must focus on effective techniques for the choice of blocking factor.

5.5 Summary

We have set out to determine whether a compiler can automatically restructure computations well enough to avoid the need for hand blocking and encourage machine-independent programming. To that end, we have examined a collection of programs similar to LAPACK for which we were able to acquire both the block and corresponding point algorithms. For each of these programs, we determined whether a plausible compiler technology could succeed in obtaining the block version from the point algorithm.

The results of this study are encouraging: we can block triangular, trapezoidal and rhomboidal loops and we have found that many of the problems introduced by complex dependence patterns can be overcome by the use of the transformation known as "index-set splitting". In many cases, index-set splitting yields codes that exhibit performance at least as good as the best block algorithms produced by LAPACK developers. In addition, we have shown that, in the special case of LU decomposition with partial pivoting, knowledge about which operations commute can enable a compiler to succeed in blocking codes that could not be blocked by any compiler based strictly on dependence analysis.

Chapter 6

Conclusion

This dissertation has dealt with compiler-directed management of the memory hierarchy. We have described algorithms to improve the balance between floating-point operations and memory requirements in program loops by reducing the number of memory references and improving cache performance. These algorithms work under the assumption that the compiler for the target machine is effective at scheduling and register allocation. The implementation of our algorithms has validated our methodology by showing that integer-factor speedups over quality commercial optimizers are possible on whole applications. Our hope is that these results will be used to encourage programmers to write their applications in a natural, machine-independent form, leaving the compiler to handle machine-dependent optimization details.

In this chapter, we review this thesis. First, we discuss the contributions that we have made to register allocation and automatic management of cache. Next, we discuss the issues related to cache performance that still must be solved and finally, we present some closing remarks.

6.1 Contributions

6.1.1 Registers

We have developed and implemented an algorithm to perform scalar replacement in the presence of inner-loop conditional control flow. The goal of scalar replacement is to expose the flow of values in arrays with scalar temporaries so that standard data-flow analysis will discover the potential for register allocation. By mapping partial redundancy elimination to scalar replacement, we are able to replace array references whose defining value is only partially available, something that was not done before this thesis. The implementation of this algorithm has shown that significant improvements are possible on scientific applications.

We have also developed and implemented an algorithm to apply unroll-and-jam to a loop nest to improve its balance between memory references and floating-point operations automatically. Our algorithm chooses unroll amounts for one or two loops to create a loop that is balanced as much as possible on a particular architecture. Included in this algorithm is an estimate of floating-point register pressure that is used to prevent spill code insertion in the final compiled loop. The results of an experiment using this technique have shown that integer-factor speedups are possible on some applications. In particular, reductions benefit greatly from unroll-and-jam because of both improved balance and easily attained instruction-level parallelism.

The algorithms for scalar replacement and unroll-and-jam eliminate the need for hand optimization to effect register allocation of array values. Not only do the algorithms capture reuse in inner loops, but also in outer loops. Although the outer-loop reuse can be obtained by hand, the process is extremely tedious and error prone and produces machine-dependent programs. In one particular case, we have shown that hand optimization actually produces slightly worse code than the automatically derived version. By relying on compiler technology to handle machine-dependent details, programs become more readable and are portable across different architectures.

Because scalar replacement and unroll-and-jam can greatly increase register pressure within loops, one question might be "How many registers are enough?" The answer depends upon the balance of the target machine. For machines that can perform multiple floating-point operations per memory operation, more registers are needed to compensate for a lower memory bandwidth. However, balanced architectures will require fewer registers because memory bandwidth is not as much of a bottleneck.

6.1.2 Cache

We have developed and implemented an algorithm to attain the best loop ordering for a loop nest in relation to memory-hierarchy performance. Our algorithm is simple, but effective. It safely ignores the effects of cache interference on reuse by only considering reuse in the innermost loop. The algorithm is driven by cache line size, access cost and miss penalty. Implementation has shown that the algorithm is capable of achieving dramatic speedups on whole applications on a single processor. Using this technology, it is possible for a programmer to order loop nests independent of language implementation of array storage and cache structure.

We have also shown that current compiler techniques are not sufficient to perform iteration-space blocking on real-world algorithms. Trapezoidal-, rhomboidal- and triangular-shaped iteration spaces, which are common in linear algebra and geophysics codes, require a transformation known as index-set splitting to be considered blockable. We have derived formulas to handle these common loop shapes with index-set splitting and shown that blocking these loops can result in significant speedups.

In addition, we have applied index-set splitting to dependences that prevent iteration-space blocking. The objective is to create new loops where the preventing dependences do not exist and blocking can be performed. Using this technique, we have been able to derive automatically the best-known block algorithms for LU decomposition with and without pivoting. Previously, compiler technology was unable to accomplish this. Unfortunately, not all block algorithms can be derived automatically by a compiler. Those block formulations that represent a change of algorithm from their corresponding point algorithm cannot be obtained automatically. To handle these situations, we have proposed that a set of programming-language extensions be developed to allow a programmer to specify block algorithms in a machine-independent manner.

Finally, our study of blockability has also led to a transformation called `IF`-inspection to handle inner loops that are guarded by control conditions. With this transformation, we determine exactly which iterations of an innermost loop will execute and then, we optimize the memory performance of the loop nest for those iterations. Large integer-factor speedups have been shown to be possible with `IF`-inspection.

6.2 Future Work

Although we have addressed many memory hierarchy issues in this dissertation, there is still much left to do. Most of that work lies in the area of cache management. In this section, we will survey some of the major issues yet to be solved.

In the computation of loop balance, memory references are assigned a uniform cost under the assumption that all accesses are made out of the cache. This is not always the case. Can we do a better job of computing the memory requirements of a loop? By applying our memory-hierarchy cost model presented in Chapter 4, we can compute the actual number of cycles needed to access memory within the innermost loop body. Using this measurement we will be able to get a better measurement of the relationship between computation and memory cycles, resulting in a better measure of loop balance. The question is "Does this increased precision matter to performance?" An implementation and comparison between the two computations of balance would answer the question.

Our treatment of iteration-space blocking for cache is not complete. Although we have studied extensions to current compiler techniques to allow a larger class of algorithms to be blocked automatically, these additional techniques alone are insufficient for implementation in a real compiler. Picking block sizes and dealing with cache interference because of set associativity are two issues that must be solved to make automatic blocking a viable technology. The optimal block size for a loop is dependent upon the behavior of the set associativity of the cache and has been shown to be difficult to determine [LRW91]. Can the compiler predict these effects at compile time or is it hopeless to perform automatic blocking with today's cache architectures? Additionally, an implementation of the techniques developed in Chapter 5 needs to be done to show its viability.

As our study of blockability revealed, not all algorithms can be written in a style that will allow them to be blocked optimally. The language extensions presented in Chapter 5 provide a vehicle for programmers to express those block algorithms in a machine-independent manner. It must be determined exactly which extensions are needed and how to implement them effectively. Assuming that we can automatically determine block sizes, can we use the language extensions to allow variable block sizes to increase performance?

One transformation for memory performance that has not been discussed in this thesis is software prefetching. Previous work in this area has ignored the effects of cache-line length on the redundancy of prefetching in the presence of limited issue slots [CKP91]. Can we take advantage of our loop interchange algorithm to attain stride-one accesses and use cache-line size to derive an efficient and effective method for using software prefetching? Future efforts should be directed toward the development of an effective software prefetching algorithm.

Finally, we have not studied the effects of multi-level caches on the performance of scientific applications. Many current systems use a MIPS R3000 or an Intel i860 with a second-level cache to attain better cache performance. Are current compiler techniques (with the addition of the rest of our future work) good enough to handle this increase in complexity? What is the correct way to view the higher-level caches? What is the true payoff of level-2 caches? How much improvement can be attained with the extra level of cache? These issues and others must be answered before such complex memory hierarchies become effective.

6.3 Final Remarks

The complexity in the design of modern memory hierarchies and the lack of sophistication in modern commercial compilers have put a significant burden on the programmer to achieve any large fraction of performance available on high-performance architectures. Given that future machine designs are certain to have increasingly complex memory hierarchies, compilers will need to adopt increasingly sophisticated memory-management strategies to offset the need for programmers to perform hand optimization. It is our belief that programmers should not be burdened with architectural details, but rather concentrate solely on program logic. To this end, our goal has been to find compiler techniques that would make it possible for a programmer to express numerical algorithms naturally with the expectation of good memory-hierarchy performance. We have demonstrated that there exist readily implementable methods that can manage the floating-point register set and improve the effectiveness of cache. By accomplishing these objectives, we have taken a significant step towards achieving our goal.

Appendix A

Formulas for Non-Rectangular Iteration Spaces

A.1 Triangular Loops

A.1.1 Upper Left: $\alpha > 0$

```
      DO 10 I = 1,N
       DO 10 J = αI+β,M
10       loop body
```

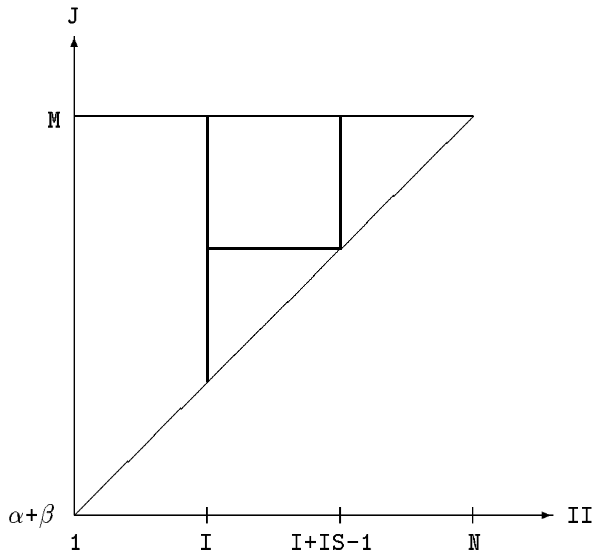

Figure A.1 Upper Left Triangular Iteration Space

Strip-Mine-and-Interchange Formula

```
      DO 10 I = 1,N,IS
       DO 10 J = αI+β,M
        DO 10 II = I,MIN((J-β)/α,I+IS-1)
10       loop body
```

Unroll-and-Jam Formula

```
      DO 10 I = 1,N,IS
       DO 20 II = I,I+IS-2
        DO 20 J = αII+β,MIN(α(I+IS-2)+β,M)
20       loop body
       DO 10 J = α(I+IS-1)+β,M
10      unrolled loop body
```

A.1.2 Upper Right: $\alpha < 0$

```
          DO 10 I = 1,N
           DO 10 J = αI+β,M
       10   loop body
```

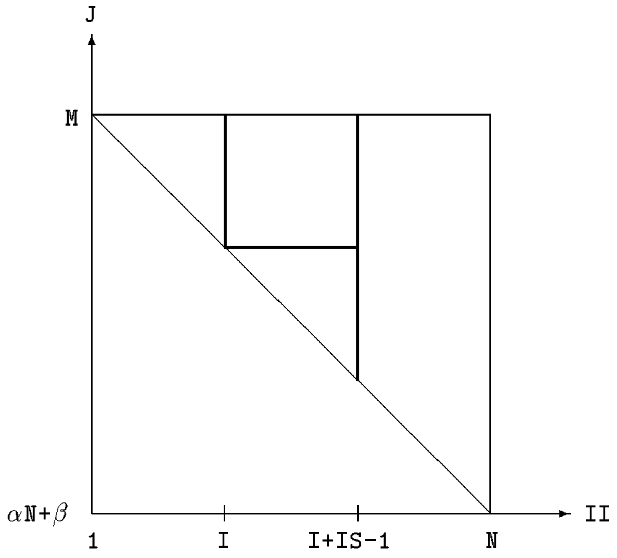

Figure A.2 Upper Right Triangular Iteration Space

Strip-Mine-and-Interchange Formula

```
          DO 10 I = 1,N,IS
           DO 10 J = α(I+IS-1)+β,M
            DO 10 II = MAX(I,(J-β)/α),I+IS-1
       10    loop body
```

Unroll-and-Jam Formula

```
          DO 10 I = 1,N,IS
           DO 20 II = I+1,I+IS-1
            DO 20 J = αII+β,MIN(α(I+1)+β,M)
       20    loop body
           DO 10 J = αI+β,M
       10   unrolled loop body
```

A.1.3 Lower Right: $\alpha > 0$

```
      DO 10 I = 1,N
       DO 10 J = L,αI+β
10      loop body
```

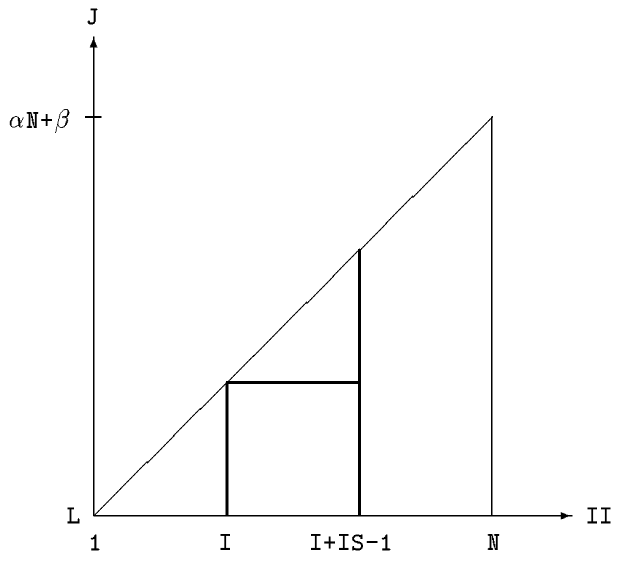

Figure A.3 Lower Right Triangular Iteration Space

Strip-Mine-and-Interchange Formula

```
      DO 10 I = 1,N,IS
       DO 10 J = L,α(I+IS-1)+β
        DO 10 II = MAX(I,(J-β)/α),I+IS-1
10       loop body
```

Unroll-and-Jam Formula

```
      DO 10 I = 1,N,IS
       DO 20 J = L,αI+β
20      unrolled loop body
       DO 10 II = I+1,I+IS-1
        DO 10 J = MAX(α(I+1)+β,L),αII+β
10       loop body
```

A.1.4 Lower Left: $\alpha < 0$

```
      DO 10 I = 1,N
       DO 10 J = L,αI+β
10      loop body
```

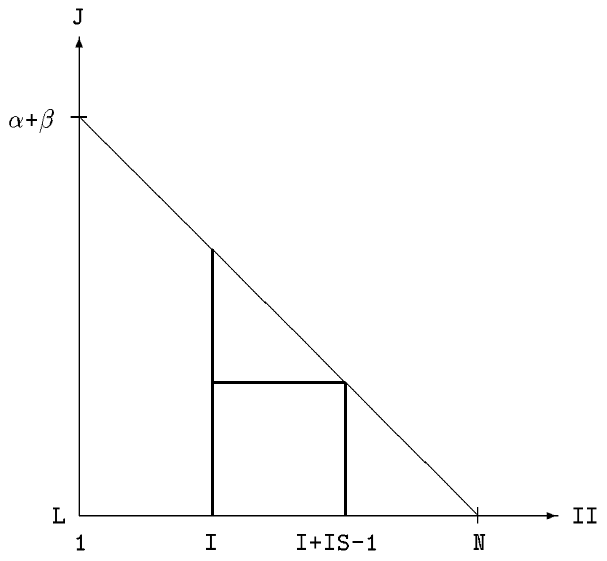

Figure A.4 Lower Left Triangular Iteration Space

Strip-Mine-and-Interchange Formula

```
      DO 10 I = 1,N,IS
       DO 10 J = L,αI+β
        DO 10 II = I,MIN((J-β)/α),I+IS-1)
10       loop body
```

Unroll-and-Jam Formula

```
      DO 10 I = 1,N,IS
       DO 20 J = L,α(I+IS-1)+β
20      unrolled loop body
       DO 10 II = I,I+IS-2
        DO 10 J = MAX(α(I+IS-2)+β,L),αII+β
10       loop body
```

A.2 Trapezoidal Loops

A.2.1 Upper-Bound MIN Function

Assume $\alpha, \beta > 0$.

```
      DO 10 I = 1,N
       DO 10 J = L,MIN(αI+β,N)
10      loop body
```

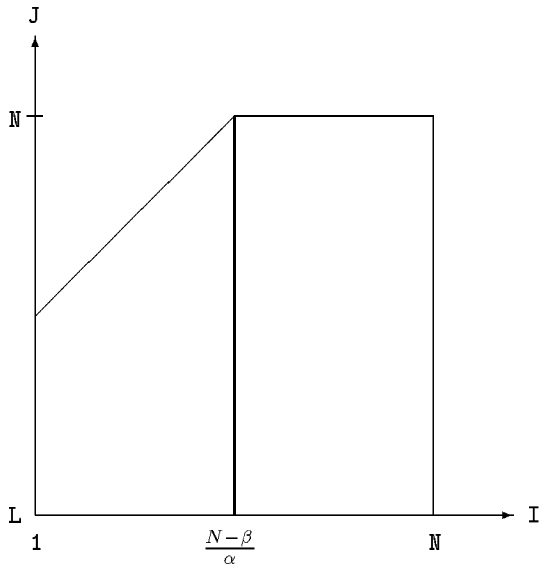

Figure A.5 Trapezoidal Iteration Space with MIN Function

After Index-Set Splitting

```
      DO 10 I = 1,MIN(N,(N-β)/α)
       DO 10 J = L,αI+β
10      loop body
      DO 20 I = MAX(1,MIN(N,(N-β)/α)+1),N
       DO 20 J = L,N
20      loop body
```

A.2.2 Lower-Bound `MAX` Function

Assume $\alpha, \beta > 0$.

```
      DO 10 I = 1,N
       DO 10 J =MAX(αI+β,L),N
10      loop body
```

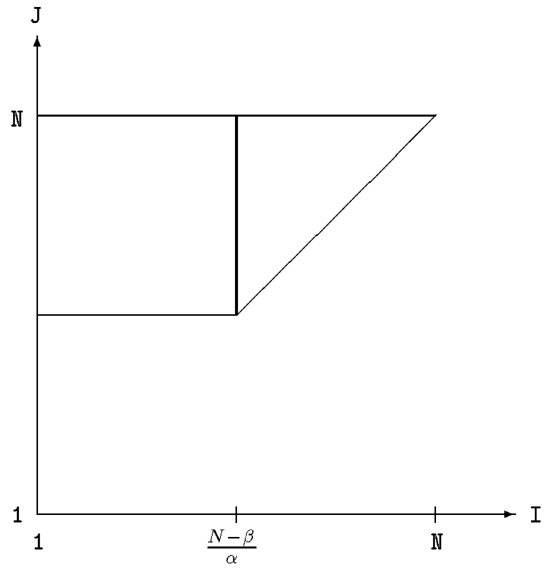

Figure A.6 Trapezoidal Iteration Space with `MAX` Function

After Index-Set Splitting

```
      DO 10 I = 1,MIN(N,(L-β)/α)
       DO 10 J = L,N
10      loop body
      DO 20 I = MAX(1,MIN(N,(L-β)/α)+1),N
       DO 20 J = α + β,N
20      loop body
```

A.3 Rhomboidal Loops

Assume $\alpha > 0$.

```
         DO 10 I = 1,N1
          DO 10 J = αI+N,αI+M
      10   loop body
```

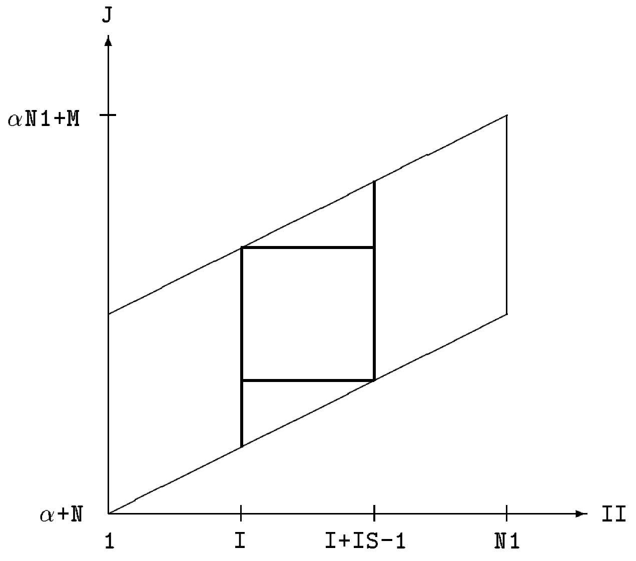

Figure A.7 Rhomboidal Iteration Space

Strip-Mine-and-Interchange Formula

```
         DO 10 I = 1,N1,IS
          DO 10 J = αI+N,αI+M
           DO 10 II = MAX(I,αI+M),MIN(αI+N,I+IS-1)
      10    loop body
```

Unroll-and-Jam Formula

```
         DO 10 I = 1,N1,IS
          DO 20 II = I,I+IS-2
           DO 20 J = αII+N,MIN(α(I+IS-2)+N,α(I+IS-2)+M)
      20    loop body
          DO 30 J = α(I+IS-1)+N,αI+M
      30   unrolled loop body
          DO 10 II = I+1,I+IS-1
           DO 10 J = MAX(α(I+1)+N,α(I+1)+M),αII+M
      10    loop body
```

Bibliography

[AC72] F.E. Allen and J. Cocke. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, 1972.

[AK87] J.R. Allen and K. Kennedy. Automatic translation of Fortran programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[AK88] J.R. Allen and K. Kennedy. Vector register allocation. Technical Report TR86-45, Department of Computer Science, Rice University, 1988.

[AN87] A. Aiken and A. Nicolau. Loop quantization: An analysis and algorithm. Technical Report 87-821, Cornell University, March 1987.

[AS78] W. Abu-Sufah. *Improving the Performance of Virtual Memory Computers*. PhD thesis, Dept. of Computer Science, University of Illinois, 1978.

[ASM86] W. Abu-Sufah and A. Malony. Vector processing on the alliant FX/8 multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 559–566, August 1986.

[BCHT90] P. Briggs, K.D. Cooper, M.W. Hall, and L. Torczon. Goal-directed interprocedural optimization. Technical Report TR90-102, Rice University, CRPC, November 1990.

[BCKT89] P. Briggs, K.D. Cooper, K. Kennedy, and L. Torczon. Coloring heuristics for register allocation. In *Proceedings of the ACM SIGPLAN 89 Conference on Program Language Design and Implementation*, Portland, OR, June 1989.

[BS88] M. Berry and A. Sameh. Multiprocessor schemes for solving block tridiagonal linear systems. *International Journal of Supercomputer Applications*, 2(3):37–57, Fall 1988.

[CAC$^+$81] G.J. Chaitin, M.A. Auslander, A.K. Chandra, J. Cocke, M.E. Hopkins, and P.W. Markstein. Register allocation via coloring. *Computer Languages*, 6:45–57, January 1981.

[Cal86] D.A. Calahan. Block-oriented, local-memory-based linear equation solution on the Cray-2: Uniprocessor algorithm. In *Proceedings of the 1986 International Conference on Parallel Processing*, 1986.

[CCK88] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined machines. *Journal of Parallel and Distributed Computing*, 5, 1988.

[CCK90] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.

[CH84] F. Chow and J. Hennessy. Register allocation by priority-based coloring. In *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction, SIGPLAN Notices Vol. 19, No. 6*, June 1984.

[CK77] John Cocke and Ken Kennedy. An algorithm for reduction of operator strength. *Communications of the ACM*, 20(11), November 1977.

[CK87] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *Proceedings of the First International Conference on Supercomputing.* Springer-Verlag, Athens, Greece, 1987.

[CKP91] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of the Fourth International Conference on Architecural Support for Programming Languages and Operating Systems*, Santa Clara, CA, April 1991.

[CP90] D. Callahan and A. Porterfield. Data cache performance of supercomputer applications. In *Supercomputing '90*, 1990.

[DBMS79] J.J. Dongarra, J.R. Bunch, C.B. Moler, and G.W. Stewart. *LINPACK User's Guide.* SIAM Publications, Philadelphia, 1979.

[DDDH90] J.J. Dongarra, J. DuCroz, I. Duff, and S. Hammerling. A set of level 3 basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 16:1–17, 1990.

[DDHH88] J.J. Dongarra, J. DuCroz, S. Hammerling, and R. Hanson. An extendend set of fortran basic linear algebra subprograms. *ACM Transactions on Mathematical Software*, 14:1–17, 1988.

[DDSvdV91] J.J. Dongarra, I.S. Duff, D.C. Sorensen, and H.A. van der Vorst. *Solving Linear Systems on Vector and Shared-Memory Computers.* SIAM, Philadelphia, 1991.

[DS88] K.H. Drechsler and M.P. Stadel. A solution to a problem with morel and renvoise's "global optimization by suppression of partial redudancies". *ACM Transactions on Programming Languages and Systems*, 10(4):635–640, October 1988.

[Fab79] Janet Fabri. Automatic storage optimization. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, Denver, CO, 1979.

[GJ79] M.R. Garey and D.S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W.H. Freeman and Co., San Francisco, 1979.

[GJG87] D. Gannon, W. Jalby, and K. Gallivan. Strategies for cache and local memory management by global program transformations. In *Proceedings of the First International Conference on Supercomputing.* Springer-Verlag, Athens, Greece, 1987.

[GJMS88] K. Gallivan, W. Jalby, U. Meier, and A.H. Sameh. Impact of hierarchical memory systems on linear algebra design. *International Journal of Supercomputer Applications*, 2(1):12–48, Spring 1988.

[GKT91] G. Goff, K. Kennedy, and C.W. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, Toronto, Ontario, June 1991.

[GM86] P.B. Gibbons and S.S. Muchnick. Efficient instruction scheduling for a pipelined architecture. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, 1986.

[GS84] J.H. Griffin and M.L. Simmons. Los Alamos National Laboratory Computer Benchmarking 1983. Technical Report LA-10051-MS, Los Alamos National Laboratory, June 1984.

[HK91] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[IT88] F. Irigoin and R. Triolet. Supernode partitioning. In *Conference Record of the Fifteenth ACM Symposium on the Principles of Programming Languages*, pages 319–328, January 1988.

[KKP$^+$81] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eight ACM Symposium on the Principles of Programming Languages*, 1981.

[KM92] K. Kennedy and K. McKinley. Optimizing for parallelism and memory hierarchy. In *Proceedings of the 1992 International Conference on Supercomputing*, Washington, DC, July 1992.

[Kuc78] D. Kuck. *The Structure of Computers and Computations Volume 1.* John Wiley and Sons, New York, 1978.

[LHKK79] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Transactions on Mathematical Software*, 5:308–329, 1979.

[LRW91] M.S. Lam, E.E. Rothberg, and M.E. Wolf. The cache performance and optimizations of blocked algorithms. In *Proceedings of the Fourth International Conference on Architecural Support for Programming Languages and Operating Systems*, April 1991.

[LS88] B. Liu and N. Strother. Programming in VS FORTRAN on the IBM 3090 for maximum vector performance. *Computer*, 21(6), June 1988.

[MR79] E. Morel and C. Revoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2), February 1979.

[Por89] A.K. Porterfield. *Software Methods for Improvement of Cache Performance on Supercomputer Applications.* PhD thesis, Rice University, May 1989.

[Sew90] G Sewell. *Computational Methods of Linear Algebra.* Ellis Horwood, England, 1990.

[Ste73] G.W. Stewart. *Introduction to Matrix Computations.* Academic Press, New York, 1973.

[SU70] R. Sethi and J.D. Ullman. The generation of optimal code for arithmetic expressions. *Journal of the ACM*, 17(4):715–728, October 1970.

[Tha81] Khalid O. Thabit. *Cache Managemant by the Compiler.* PhD thesis, Rice University, November 1981.

[WL91] M.E. Wolf and M.S. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, June 1991.

[Wol82] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois, October 1982.

[Wol86a] M. Wolfe. Advanced loop interchange. In *Proceedings of the 1986 International Conference on Parallel Processing*, August 1986.

[Wol86b] M. Wolfe. Loop skewing: The wavefront method revisited. *Journal of Parallel Programming*, 1986.

[Wol87] M. Wolfe. Iteration space tiling for memory hierarchies. In *Proceedings of the Third SIAM Conference on Parallel Processing for Scientific Computing*, December 1987.

[Wol89] M. Wolfe. More iteration space tiling. In *Proceedings of the Supercomputing '89 Conference*, 1989.