Compiler Transformations for High-Performance Computing

DAVID F. BACON, SUSAN L. GRAHAM, AND OLIVER J. SHARP

Computer Science Division, University of California, Berkeley, California 94720

In the last three decades a large number of compiler transformations for optimizing programs have been implemented. Most optimizations for uniprocessors reduce the number of instructions executed by the program, and analyze the properties of scalar quantities using flow analysis techniques. In contrast, optimizations for high-performance vector and parallel processors maximize parallelism and memory locality, mostly by tracking the properties of arrays using loop dependence analysis.

In this survey we give an overview of the important high-level program restructuring techniques for imperative languages such as C and Fortran, and to describe how and when they should be applied on high-performance uniprocessors and on vector and multiprocessor machines. The basic issues involved in optimization are discussed, and the compiler analysis required for the transformations is described in some detail. A basic familiarity with modern computer architecture and program compilation is assumed.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors–*compilers; optimization;* D.1.3 [**Programming Techniques**]: Concurrent Programming; I.2.2 [**Artificial Intelligence**]: Automatic Programming–*program transformation*

General Terms: Compilation, Optimization, Parallelism

Additional Key Words and Phrases: Vectorization, multiprocessors, superscalar processors, dependence analysis

Technical Report No. UCB/CSD-93-781

This research has been sponsored in part by the Defense Advanced Research Projects Agency (DARPA) under contract DABT63-92-C-0026, by NSF grant CDA-8722788, by an IBM Resident Study Program Fellowship to David Bacon, and by a Hertz Fellowship to Oliver Sharp. The content of the paper does not necessarily reflect the position or the policy of the Government or of IBM and no official endorsement should be inferred.

CONTENTS

INTRODUCTION

Optimizing compilers have become an essential component of modern high-performance computer systems. In addition to translating the input program into machine language, they analyze it and apply various transformations to it to reduce its running time or its size.

As optimizing compilers become more effective, programmers can become less concerned about the details of the underlying machine architecture and can employ higher-level, more succinct, and more intuitive programming constructs and program organizations. Simultaneously, hardware designers are able to employ designs that yield greatly improved performance because they need only concern themselves with the suitability of the design as a compiler target, not with its suitability as a direct programmer interface.

One of the approaches used by computer architects to achieve high performance is to use parallelism at various levels of granularity to overlap computation. In this survey we describe transformations that optimize programs written in imperative languages such as Fortran and C for high-performance architectures, including superscalar, vector, and various classes of multiprocessor machines. Almost all of these transformations are applicable to the languages in the Algol family; many are applicable to functional, logic, distributed, and object-oriented languages as well.

These other languages raise additional optimization issues which space does not permit us to cover in this survey. The references include some starting points for investigation of optimizations for LISP and functional languages [Kranz et al. 1986; Appel 1992; Clark and Peyton-Jones 1985], object-oriented languages [Chambers and Ungar 1989], logic programming languages [Aït-Kaci 1991], and the set-based language SETL [Freudenberger et al. 1983].

We have also restricted the discussion to higher-level transformations which require some program analysis. Thus we exclude peephole optimizations and, as much as possible, machine level optimizations, although some discussion of instruction scheduling and register allocation is included. We use the term *optimization* as shorthand for *optimizing transformation.*

Finally, because of the richness of the topic, we have not given a detailed description of intermediate program representations and analysis techniques.

A number of appendices have been included with material that presents additional details and background information.

We make use of a number of different machine models all based on a hypothetical superscalar processor called S-DLX. Appendix A details the machine models, presenting a simplified architecture and instruction set that we use when we need to discuss machine code. While all assembly language examples are commented, the reader will need to refer to the appendix to understand some details of the examples (such as cycle counts).

We assume a basic familiarity with program compilation issues. Appendix B discusses program representations used internally by compilers. Readers unfamiliar with program flow analysis may consult Appendix C, which describes some forms of analysis that are used to identify the usefulness and correctness of applying a transformation, or Aho et al. [1986], which provides a more lengthy treatment.

1 SOURCE LANGUAGE

All of the high-level examples in this survey are written in Fortran-90, primarily in the Fortran-77 subset. In some cases it is necessary to use assembly language to demonstrate the effects of the optimizations; this is described in Section 3.

We have chosen to use Fortran because it is the de facto standard of the high-performance engineering and scientific computing community. Fortran has also been the input language for a number of research projects studying parallelization [Allen et al. 1988a; Balasundaram et al. 1989; Polychronopoulos et al. 1989]. It was chosen by these projects not only because of its ubiquity among the user community, but also because its lack of pointers and its static memory model make it more amenable to analysis.

The optimizations we have presented are not specific to Fortran — in fact, many commercial compilers use the same intermediate language and optimizer for both C and Fortran. The presence of unrestricted pointers in C can reduce opportunities for optimization because it is impossible to determine which variables may be referenced by a pointer. The process of determining which references may refer to the same storage locations is called *alias analysis* and is described more fully in Appendix C.2.4.

Arrays in Fortran are laid out in memory in column-major form: the first subscript varies fastest. This is relevant for understanding the transformations that improve memory locality.

The only changes to Fortran in our examples are that array subscripting is denoted by square brackets to distinguish it from function calls; and we use **do all** loops to indicate textually that all the iterations of a loop may be executed concurrently.

To make the structure of the computation more explicit, we will generally express loops as iterations rather than in Fortran-90 array notation. Programmers should generally use array notation when it is available because it simplifies the code and exposes parallelism to the compiler.

When describing compilation for vector machines, we sometimes use array notation when the mapping to hardware registers is clear. For instance, the loop

```
do all i = 1, 64
  a[i] = a[i] + c
end do all
```

could be implemented with a scalar-vector add instruction. This would be written in Fortran-90 array notation as

```
a[1:64] = a[1:64] + c
```

or in vector machine assembly language as

```
LF     F2, c(R30) ;load c into reg F2
ADDI   R8, R30,#a ;load addr. of a into R8
LV     V1, R8     ;load vector a[1:64] to V1
ADDSV  V1, F2, V1 ;add scalar to vector
SV     V1, R8     ;store vector in a[1:64]
```

Array notation will *not* be used when loop bounds are unknown, because there is no longer an obvious correspondence between the source code and the fixed-length vector operations that perform the computation. To use vector operations, the compiler must perform the transformation called *strip-mining*, which is discussed in Section 6.2.1.

2 TRANSFORMATION ISSUES

For a compiler to apply an optimization to a program, it must do three things:

1. *Decide* upon a part of the program to optimize and a particular transformation to apply to it;

2. *Verify* that the transformation either does not change the meaning of the program or changes it in a restricted way that is acceptable to the user; and

3. *Transform* the program.

In this survey we concentrate on the last step, transformation of the program. However, Section 5 we will introduce dependence analysis techniques which are used for deciding upon and verifying transformations, and Appendices B and C provide some further details.

Step 1 is the most poorly understood and is an active area of current research. Typically compilers apply a verification test for an optimization to an entire procedure, identifying every legal opportunity to use it. A variety of heuristics then determine whether the transformation is applied. Optimizations that rely on similar verification procedures are generally applied together to reduce the cost of program analysis, which may dominate compilation time.

Such ad-hoc strategies work reasonably well for uniprocessor targets, although even then it is possible for a sequence of optimizations to slow down a program. For example, an attempt to reduce the number of instructions executed may actually degrade performance by making less efficient use of the cache. While the sequence of transformations that is best for each application may be unique, compilers generally apply transformations in a relatively fixed sequence.

But as processor architectures become more complex, the number of dimensions in which optimization is possible increases and the decision process is greatly complicated. Some recent work has concentrated on systematizing the process of transformation (described briefly in Section 8); a great deal of further research is needed in this area.

2.1 Correctness

When a program is transformed by the compiler, the meaning of the program should remain unchanged. The easiest way to achieve this is to require that the transformed program perform exactly the same operations as the original, in exactly the order imposed by the semantics of the language. However, such a strict interpretation leaves little room for improvement. A more practical definition is

Definition 1 *A transformation is* legal *if the original and the transformed programs produce exactly the same output for* identical executions.

Definition 1.1 *Two executions of a program are* identical executions *if they are supplied with the same input data and if every corresponding pair of non-deterministic operations in the two executions produces the same result.*

Non-determinism can be introduced by language constructs (such as Ada's **select** statement), or by calls to system or library routines that return information about the state external to the program (such as Unix `time()` or `read()`).

In some cases it is straightforward to cause executions to be identical, for instance by entering the same inputs. In other cases there may be support for determinism in the programming environment; for instance, a compilation option which forces **select** statements to evaluate their guards in a deterministic order. As a last resort, the programmer may have to temporarily replace non-deterministic system calls with deterministic operations in order to compare two versions.

To illustrate many of the common problems encountered when trying to optimize a program and yet maintain correctness, Figure 1 shows a program (a) and a transformed version (b). The transformed version may seem to have the same semantics as the original, but it violates Definition 1 in the following ways:

- Overflow. Let $\mathcal{R}_{max}$ be the largest real number representable by the machine. If `b[k]` $= \mathcal{R}_{max} - 1$ and `a[1]` $= -2$, then changing the order of the additions so that `C=b[k]+2` will cause an overflow to occur in the transformed program that did not occur in the original. Even if the original program would have overflowed, the transformation causes the exception to happen at a different point. This situation complicates debugging, since the transformation is not visible to the programmer. Finally, if there had been a `print` statement between the assignment and use of `C`, the transformation would actually change the output of the program.

- Different results. Even if no overflow occurs, the resulting values of elements of `a` may be slightly different because the order of the additions has been changed. The reason is that floating point numbers are approximations of real numbers, and the order in which the approximations are applied

```
subroutine tricky(a,b,n,m,k)
integer n, m, k
real a[m], b[m]

do i = 1, n
  a[i] = b[k] + a[i] + 2
end do
return
```

(a) original program

```
subroutine tricky(a,b,n,m,k)
integer n, m, k
real a[m], b[m]

C = b[k] + 2
do i = n, 1, -1
  a[i] = a[i] + C
end do
return
```

(b) transformed program

```
k = m+1
n = 0
call tricky(a,b,n,m,k)
```

(c) possible call to `tricky`

```
equivalence (a[1], b[n])

k = n
call tricky(a,b,n,m,k)
```

(d) possible call to `tricky`

Figure 1: Incorrect Program Transformations

(rounding) can affect the result. However, for a sequence of commutative and associative integer operations, if no order of evaluation can cause an exception, then all evaluation orders are equivalent. We call operations which are algebraically but not computationally commutative (or associative) *semi-commutative* (or *semi-associative*) operations. Boolean operations are fully commutative, since they do not cause exceptions or compute approximate values.

- Memory fault. If `k` $>$ `m` but `n` < 1, the reference to `b[k]` is illegal. The reference would not be evaluated in the original program because the loop body is never executed, but it would occur in the call shown in Figure 1 (c).

- Different results. `a` and `b` may be completely or partially aliased to one another, changing the values assigned to `a` in the transformed program. Figure 1 (d) shows how this might occur: if the call is to the original subroutine, `b[k]` is changed when `i` $= 1$, since `k` $=$ `n` and `b[n]` is aliased to `a[1]`. In the transformed version, the old value of `b[k]` is used for all `i`, since it is read before the loop. Even if the reference to `b[k]` were moved back inside the loop, the transformed version would still give different results because the loop traversal order has been reversed.

As a result of these problems, slightly different definitions are used in practice. When bit-wise identical results are desired, the following definition is used:

Definition 2 *A transformation is* legal *if,* for all semantically correct program executions*, the original and the transformed programs produce exactly the same output for identical executions.*

Languages typically have many rules that are stated but not enforced; for instance in Fortran, array subscripts must remain within the declared bounds, but this rule is not enforced at compile- or run-time. A program execution is correct if it does not violate the rules of the language. Note that correctness is a property of a program execution, not of the program itself, since the same program may execute correctly under some inputs and incorrectly under others.

Because exceptions are considered semantically incorrect, the legally transformed program can produce different results when an exception occurs. For languages which define a specific semantics for exceptions, this definition may need to be extended.

However, demanding bit-wise identical results can substantially constrain opportunities for optimization, and is often not necessary. As a result, the most commonly applied correctness criterion is

Definition 3 *A transformation is* legal *if, for all semantically correct executions* of the original program*, the original and the transformed programs perform equivalent operations for identical executions. All permutations of semi-commutative operations are considered equivalent.*

Since we can not predict the degree to which transformations of semi-commutative operations change the output, we must use an operational rather than an observational definition of equivalence. In practice, programmers generally observe whether the numeric results differ by more than a certain tolerance, and if they do, force the compiler to employ Definition 2.

2.2 Scope

Optimizations can be applied to a program at different levels of granularity. As the scope of the transformation is enlarged, the cost of analysis generally increases. Some useful gradations of complexity are:

- statement — arithmetic expressions are the main source of potential optimization within a statement.
- basic block (straight-line code) — this is the focus of early optimization techniques. The advantage for analysis is that there is only one entry point, so control transfer need not be considered in tracking the behavior of the code.
- innermost loop — to effectively target high-performance architectures, compilers need to focus on loops. Most of the transformations discussed in this paper are based on loop manipulation. Many of them have only been studied or widely applied in the context of the innermost loop.
- perfect loop nest — a *loop nest* is a set of loops one inside the next. The nest is called a *perfect nest* if the body of every loop other than the innermost consists only of the next loop in the nest. Because a perfect nest is more easily summarized and reorganized, several transformations only apply to perfect nests.
- general loop nest — any loop nesting, perfect or not.
- procedure — some optimizations, memory access transformations in particular, yield better improvements if they are applied to an entire procedure at once. The compiler must be able to manage the interactions of all the basic blocks and control transfers within the procedure. The standard and rather confusing term for procedure-level optimization in the literature is *global optimization.*
- inter-procedural — considering several procedures together often exposes more opportunities for optimization; in particular, procedure call overhead is often significant, and can sometimes be reduced or eliminated with inter-procedural analysis (see Section 6.6). Relatively few compilers perform inter-procedural optimization.

We will generally confine our attention to optimizations beyond the basic block level.

3 TARGET ARCHITECTURES

In this paper we discuss compilation techniques for high-performance architectures, which for our purposes are superscalar, vector, SIMD, shared-memory multiprocessor, and distributed-memory multiprocessor machines. These architectures have in common that they all use parallelism in some form to improve performance.

The structure of an architecture dictates how the compiler must optimize along a number of different (and sometimes competing) axes. The compiler must attempt to:

- maximize use of computational resources (processors, functional units, vector units);
- minimize the number of operations performed;
- minimize use of memory bandwidth (register, cache, network); and
- minimize use of memory.

While optimization for scalar CPUs has concentrated on minimizing the dynamic instruction count (or more precisely, the number of machine cycles required), high-performance applications are often at least as dependent upon the performance of the memory system as they are on the performance of the functional units.

In particular, the distance in memory between consecutively accessed elements of an array can have a major performance impact. This distance is called the *stride*. If a loop is accessing every fourth element of an array, it is a stride-4 loop. If every element is accessed in order, it is a stride-1 loop. Stride-1 access is desirable because it maximizes memory locality and therefore the efficiency of the cache, translation lookaside buffer (TLB), and paging systems; it also eliminates bank conflicts on vector machines.

Another key to achieving peak performance is the paired use of multiply and add operations in which the result from the multiplier can be fed into the adder. For instance, the IBM RS/6000 has a multiply-add instruction which uses the multiplier and adder in a pipelined fashion; one multiply-add can be issued each cycle. The Cray Y-MP C90 does not have a single instruction; instead the hardware detects that the result of a vector multiply is used by a vector add, and uses a strategy called *chaining* in which the results from the multiplier's pipeline are fed directly into the adder.

Applications that do not use such compound operations may be only half as fast as those that do. It is therefore very important to organize the code so as to issue as many multiply-adds as possible.

There are a number of variables which are used to analyze the performance of the compiled code quantitatively:

- S is the hardware speed in operations per second. Typically, speed is measured either in millions of instructions (MIPS) or in millions of floating point operations (megaflops).
- P is the number of processors.
- F is the number of operations executed by a program.
- T is the time in seconds to run a program.
- $U = F/ST$ is the *utilization* of the machine by a program; a utilization of 1 is ideal, but real programs typically have significantly lower utilizations.
- Q is the ratio of the number of uses of a word in memory to the number of times it is loaded into a register. The higher the value of Q, the more re-use is being achieved and the less memory bandwidth is consumed. Values of Q below 1 indicate that redundant loads are occurring.
- R_∞ is the number of registers that are required by a program assuming that the target machine has an infinite register set (but assuming that registers are re-used when the value they contain is no longer needed). R_m is the number of registers actually available on the machine.
- $\Pi = R_\infty/R_m$ is a measure of the *register pressure* of a program. $\Pi > 1$ indicates that extra loads and stores will have to be generated to save some registers' values and free them for other uses. This process is called *spilling* the registers, and clearly reduces Q.

In Appendix A we present a series of model architectures that we will use throughout this survey to demonstrate the effect of various compiler transformations. The architectures include a superscalar CPU (S-DLX), a vector CPU (DLX-V), a shared memory multiprocessor (MX-s) and a distributed memory multiprocessor (MX-d). We assume that the reader is familiar with basic principles of modern computer architecture, including RISC design, pipelining, caching, and instruction-level parallelism. Our generic architectures are based on DLX, an idealized RISC architecture introduced by Hennessey and Patterson [1990].

4 COMPILER ORGANIZATION

To demonstrate how transformation fits into compiler design, we have presented a diagram of the various parts of an optimizing compiler in Figure 2. Every compiler is different, so we have tried to present one reasonable organization that has a place for all the techniques mentioned in this survey.

The first two phases, lexing and parsing, constitute the *front-end* of the compiler and will not be discussed further. The front-end is responsible for converting the original source into more convenient internal data structures and for checking whether the static semantic constraints of the language have been properly satisfied. The parser generally produces an abstract syntax tree and a symbol table.

The next stage of compilation is analysis, which can take many forms. In our canonical compiler, we assume that the first step is to do *control flow* analysis in order to produce a *control-flow graph* (CFG). The CFG, which converts the different kinds of control transfer constructs in the program into a single form that is easier for the compiler to manipulate, is discussed in more detail in Appendix B.

After control flow has been dealt with, the compiler examines how data is being used with *dataflow analysis* (described in Appendix C.1) and *dependence analysis* (described in Section 5). There are a variety of representations for capturing flow information; we assume that our compiler uses *program dependence graphs* and *static single-assignment form* (described in Appendix B), and dependence vectors (described in Section 5). Some compilers will only use one or two of these intermediate forms, while others will use entirely different ones.

After analyzing the code, the compiler can begin to transform it. The compiler designer must decide on an order in which to apply transformations; some transformations may enable others which in turn enable the original transformation to improve the code further. Although the picture implies that analysis is complete before the transformations are applied, in practice it is often necessary to re-analyze code after it has been modified.

Once the program has been fully transformed, the last stage of compilation is to convert it into assembly code. This translation is done by the *back-end* and will not be discussed in detail; our compiler uses the common approach of converting the high-level representations used during transformation into the low-level *register-transfer language* (RTL) [Davidson and Fraser 1984]. RTL is used for register allocation, instruction selection, and instruction reordering to exploit processor scheduling policies.

5 DEPENDENCE ANALYSIS

Among the various forms of analysis used by optimizing compilers, the one we rely on most heavily in this survey is *dependence analysis* [Wolfe 1989b; Banerjee 1988b]. This section briefly introduces dependence analysis, its terminology, and the underlying theory. Other forms of program analysis are discussed in Appendix C.

A dependence is a relationship between two computations that places constraints on their execution order. Dependence analysis identifies these constraints, which are then used to determine whether a particular transformation can be applied without changing the semantics of the computation.

5.1 Types of Dependences

There are two kinds of dependences: *control dependence* and *data dependence*. There is a con-

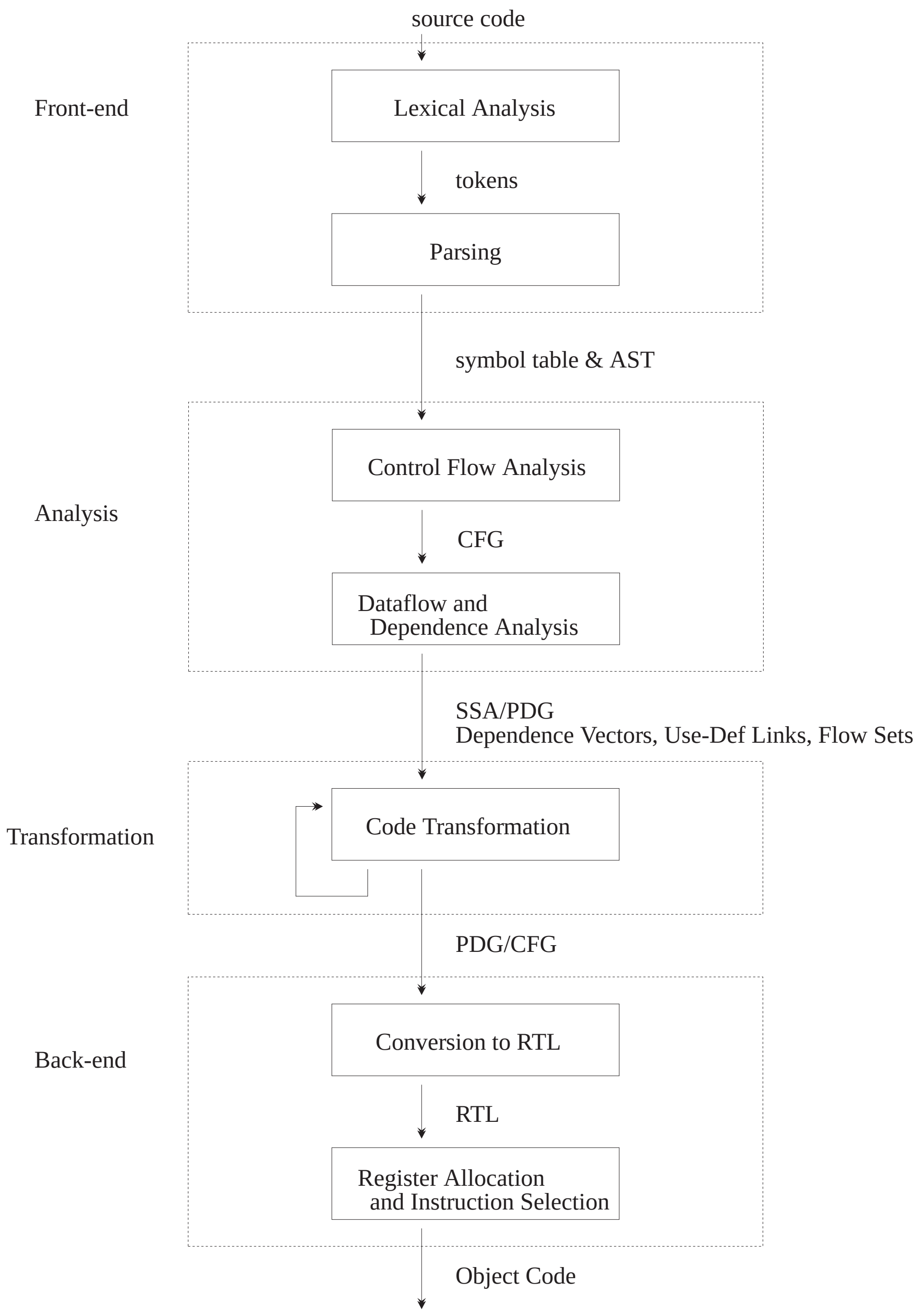


Figure 2: Possible Organization of an Optimizing Compiler

trol dependence between statement 1 and statement 2, written $S_1 \xrightarrow{c} S_2$, when statement S_1 determines whether S_2 will be executed. For example:

```
1       if (a = 3) then
2         b = 10
        end if
```

Two statements have a *data* dependence if they cannot be executed simultaneously due to conflicting uses of the same variable. There are three types of data dependences: *flow dependence* (also called true dependence), *anti-dependence*, and *output dependence*. S_4 has a flow dependence on S_3 (denoted by $S_3 \rightarrow S_4$) when S_3 must be executed first because it writes a value that is read by S_4. For example:

```
3       a = c*10
4       d = 2*a + c
```

S_6 has an anti-dependence on S_5 (denoted by $S_5 \nrightarrow S_6$) when S_6 writes a variable that is read by S_5:

```
5       e = f*4 + g
6       g = 2*h
```

An anti-dependence does not constrain execution as tightly as a flow dependence. As before, the code will execute correctly if S_6 is delayed until after S_5 completes. An alternative solution is to use two memory locations g_5 and g_6 to hold the values read in S_5 and written in S_6, respectively. If the write by S_6 completes first, the old value will still be available in g_5.

An output dependence holds when both statements write the same variable:

```
7       a = b*c
8       a = d+e
```

We denote this condition by writing $S_7 \mathrel{\circ\kern-0.8em\rightarrow} S_8$. Again, as with an anti-dependence, storage replication can allow the statements to execute concurrently. In this short example there is no intervening use of a and no control transfer between the two assignments, so the computation in S_7 is redundant and can actually be eliminated.

We denote an unspecified type of dependence by $S_1 \Rightarrow S_2$. Another common notation for dependences uses $S_3 \delta S_4$ for $S_3 \rightarrow S_4$, $S_5 \overline{\delta} S_6$ for $S_5 \nrightarrow S_6$, and $S_7 \delta^o S_8$ for $S_7 \mathrel{\circ\kern-0.8em\rightarrow} S_8$.

In the case of data dependences, when we write $X \Rightarrow Y$ we are being somewhat imprecise: it is reads and writes to the same data items that are dependent, not entire statements. In the output dependence example above, b, c, d, and e can all be read from memory in any order, and the results of b*c and d+e can be executed as soon as their operands have been read from memory. $S_7 \mathrel{\circ\kern-0.8em\rightarrow} S_8$ actually means that the store of the value b*c into a must precede the store of the value d+e into a. When there is a potential ambiguity, we will distinguish between different variable references within statements.

5.2 Representing Dependences

To capture the dependency information for a piece of code, the compiler creates a *dependence graph*; each node in the graph typically represents one statement. An arc between two nodes indicates that there is a dependence between the computations they represent.

Because it is cumbersome to account for both control and data dependence during analysis, compilers often convert control dependences into data dependences using a technique called *if-conversion* [Allen et al. 1983]. If-conversion introduces additional boolean variables that encode the conditional predicates; every statement whose execution depends on the conditional is then modified to test the boolean variable. In the transformed code, data dependence subsumes control dependence.

5.3 Loop Dependence Analysis

So far we have examined dependence in the context of straightline code with conditionals — analyzing loops is a more complicated problem. In non-looping code, each statement is executed at most once, so the dependence arcs described so far capture all the possible dependence relationships. In looping code, each statement may be executed many times, and for many transformations it is necessary to describe dependences that

```
      do i = 2,n
1       a[i] = a[i] + c
2       b[i] = a[i-1] * b[i]
      end do
```

(a)

```
      do i = 2, n
        do j = 1, n-1
          a[i,j] = a[i,j] + a[i-1,j+1]
        end do
      end do
```

(b)

```
      do i = 1, n
        do j = 2, n-1
          a[j]=(a[j] + a[j-1] + a[j+1])/3
        end do
      end do
```

(c)

Figure 3: Loop-carried Dependence

exist between iterations, called *loop-carried* dependences.

Three loop nests with loop-carried dependences are shown in Figure 3. Consider the simple example (a): there is no dependence between S_1 and S_2 within any single iteration of the loop, but there is one between two successive iterations. S_2 in iteration k reads the value of a$[k-1]$ written by S_1 in iteration $k-1$.

To compute dependence information for loops, the key problem is understanding the use of arrays; scalar variables are relatively easy to manage. To track array behavior, the compiler must analyze the subscript expressions in each array reference.

To discover whether there is a dependency in the loop nest, it is sufficient to determine whether one iteration can ever write a value that is read or written in any other iteration.

Depending on the language, loop increments may be arbitrary expressions. However, the dependence analysis algorithms usually require that the loops have only unit increments. When the ranges in a program have a more complex form, the compiler may be able to normalize them to fit the requirements of the analysis, as described in Section 6.2.16. For the remainder of this section we will assume that all loops are incremented by 1 for each iteration.

Figure 4 shows a generalized perfect nest of d loops. The body of the loop nest reads and writes elements of the m-dimensional array a. The functions f_i and g_i map the current values of the loop iteration variables to an integer which indexes the i^{th} dimension of a. The generalized loop can give rise to any type of data dependence: for instance, two different iterations may write the same element of a, creating an output dependence.

An iteration can be uniquely named by a vector of d elements $I = (i_1, \ldots, i_d)$, where each index falls within the iteration range of its corresponding loop in the nesting (that is, $l_p \leq i_p \leq u_p$). The outermost loop corresponds to the leftmost index.

We wish to discover what loop-carried dependences exist between the two references to a, and to describe somehow those dependences that exist. Clearly, a reference in iteration J can only depend upon another reference in iteration I that was executed before it, not after it. We formalize the notion of "before" with the $\prec$ relation:

$$I \prec J \text{ iff } \exists p : i_p < j_p \wedge \forall q < p : i_q = j_q.$$

Note that this definition must be extended slightly when the loop increment may be negative.

A reference in some iteration J depends upon a reference in iteration I if and only if at least one reference is a write and

$$I \prec J \wedge \forall p : f_p(I) = g_p(J).$$

In other words, there is a dependence when the values of the subscripts are the same in different iterations. If no such I and J exist, the two references are independent across *all* iterations of the loop. In the case of an output dependence

```
    do $i_1 = l_1, u_1$
        do $i_2 = l_2, u_2$
            ...
                do $i_d = l_d, u_d$
1                   a[$f_1(i_1, .., i_d), ..., f_m(i_1, ..., i_d)$] = ...
2                   ... = a[$g_1(i_1, ..., i_d), ..., g_m(i_1, ..., i_d)$]
                end do
            ...
        end do
    end do
```

Figure 4: General Loop Nest

between the same write in different iterations, the condition is simply $\forall p : f_p(I) = f_p(J)$

For example, suppose that we are attempting to describe the behavior of the loop in Figure 3 (b). Each iteration of the inner loop writes the element `a[i,j]`. There is a dependence if any other iteration reads or writes that same element. In this case, there are many pairs of iterations that depend on each other. Consider iterations $I = (1,3)$ and $J = (2,2)$. Iteration I occurs first, and writes the value `a[1,3]`. This value is read in iteration J, so there is a flow dependence from iteration I to iteration J. Using the same notation for dependences, we write $I \rightarrow J$.

When $X \Rightarrow Y$, we define the *dependence distance* as $Y - X = (Y_1 - X_1, \ldots, Y_d - X_d)$. In Figure 3 (b), the dependence distance $J - I = (1, -1)$. When all the dependence distances for a specific pair of references are the same, the potentially unbounded set of dependences can be represented by the dependence distance. When a dependence distance is used to describe the dependences for all iterations, it is called a *distance vector* (introduced by Kuck [1978] and Muraoka [1971]).

A legal distance vector V must be *lexicographically positive*, meaning that $0 \prec V$ (the first non-zero element of the distance vector must be positive). A negative element in the distance vector means that the dependence in the corresponding loop is on a higher-numbered iteration. If the first non-zero element were negative, this would indicate a dependence upon a future iteration, which is impossible.

There is often confusion of dependence between *iterations* with dependence between *array elements*. The operations on array elements create the dependences, but the dependence vectors describe dependences between iterations. For instance, the loop nest that updates the one-dimensional array `a` in Figure 3 (c) has dependences $\{(0,1), (1,0), (1,-1)\}$.

In some cases it is not possible to determine the exact dependence distance at compile-time, or the dependence distance may vary between iterations, but there is be enough information to partially characterize the dependence. A *direction vector* (introduced by Wolfe [1989b]) is commonly used to describe such dependences.

For a dependence $I \Rightarrow J$, the direction vector $W = (w_1, \ldots, w_d)$ where

$$w_p = \begin{cases} < & \text{if } I_p < J_p \\ = & \text{if } I_p = J_p \\ > & \text{if } I_p > J_p \end{cases}$$

In Figure 3 the direction vector for loop (b) is $(<, >)$, and the direction vectors for loop (c) are $\{(=, <), (<, =), (<, >)\}$.

The dependence behavior of a loop is described by the set of dependence vectors for each pair of possibly conflicting references. These can be summarized into a single loop direction vector, at the expense of some loss of information (and potential for optimization). The dependences of the loop in Figure 3 (c) can be summarized as $(\leq, *)$. The symbol $\neq$ denotes both a $<$ and $>$ direction, and $*$ denotes $<$, $>$, and $=$.

5.4 Subscript Analysis

In discussing the analysis of loop-carried dependence, we omitted an important detail: how the compiler decides whether two array references might refer to the same element in different iterations. In examining a loop nest, the compiler first tries to prove that different iterations are independent by applying various tests to the subscript expressions. These tests rely on the fact that the expressions are almost always linear. If the subscript expressions are too complex to analyze, or if a dependency is found, the compiler then tries to describe the dependence with a direction or distance vector.

There are a large variety of tests, all of which can prove independence in some cases. It is infeasible to solve the problem directly, even for linear subscript expressions, because finding dependences is equivalent to the NP-complete problem of finding integer solutions to systems of linear Diophantine equations [Banerjee et al. 1979]. Two general and approximate tests are the GCD [Towle 1976] and Banerjee's inequalities [Banerjee 1988a].

In addition, there are a large number of *exact* tests that exploit some subscript characteristics to determine whether a particular type of dependence exists. These include the Single-Index Test [Banerjee 1979; Wolfe 1989b] and the Delta Test [Goff et al. 1991]. Other approaches examine multiple subscripts simultaneously as in the λ-test [Li et al. 1990], multi-dimensional GCD [Banerjee 1988a], and the power test [Wolfe and Tseng 1992].

There have been several studies of the loop subscript expressions that occur in practice, and how well the various tests perform in analyzing them [Banerjee 1988a; Shen et al. 1989; Lee et al. 1985]. These studies have generally concerned themselves with scientific code, finding that the subscript expressions which appear are almost always simple enough to respond well to inexpensive dependence tests.

6 TRANSFORMATIONS

This section catalogs the program transformations themselves. Our primary emphasis is on loops, since that is where most of the execution time is spent in programs. For each transformation we provide an example, discuss its benefits and shortcomings, identify any variants, and provide citations.

A standard reference on compilers in general is the "Red Dragon" book, to which we refer for some of the most common examples [Aho et al. 1986]. We also drew upon previous summaries [Allen and Cocke 1971; Kuck 1977; Padua and Wolfe 1986; Wolfe 1989b; Rau and Fisher 1993].

Because some transformations were already familiar to programmers who applied them manually, we often cite only the work of researchers who have systematized and automated the implementation of these transformations. Additionally, we omit citations to work that is restricted to basic blocks when global optimization techniques exist. Even so, the origin of some optimizations is murky. For instance, Ershov's ALPHA compiler performed interprocedural constant propagation (albeit in a limited form) in 1964 [Ershov 1966].

6.1 Loop Reordering Transformations

In this section we describe transformations which change the order in which the iterations of a perfect loop nest are executed. These transformations are used to expose parallelism and improve memory locality. When loops are imperfectly nested, *loop distribution* can sometimes be used to create perfect loop nests (see Section 6.2.8).

A major goal of optimizing compilers for high-performance architectures is to discover and exploit parallelism in loops. We will indicate when a loop can be executed in parallel by using a **do all** loop instead of a **do** loop. The iterations of a **do all** loop can be executed in any order, or all at once.

To determine whether a loop can be executed in parallel, its loop-carried dependences must be

```
do i = 1, n
  do j = 2, n
    a[i,j] = a[i,j-1] + c
  end do
end do
```

(a) Outer loop is parallelizable.

```
do i = 1, n
  do j = 1, n
    a[i,j] = a[i-1,j] + a[i-1,j+1]
  end do
end do
```

(b) Inner loop is parallelizable.

Figure 5: Dependence conditions for parallelizing loops.

stride	Cache Misses	TLB Misses	% of stride 1
1	64	2	100
2	128	4	83
4	256	8	63
8	512	16	40
12	768	24	28
16	1024	32	23
64	1024	128	19
256	1024	512	12
512	1024	1024	7.5

```
double precision a[*]

do i = 1, 1024*stride, stride
  a[i] = a[i] + c
end do
```

Figure 6: Predicted effect of stride on performance of an IBM RS/6000 for the above loop. Array elements are double precision (8 bytes); miss rates are per 1024 iterations. Beyond stride-16, TLB misses dominate.

examined. The obvious case is when all the dependence distances for the loop are 0 (direction =), meaning that there are no dependences carried across iterations by the loop. This is the case in Figure 5 (a): the distance vector for the loop is $(0, 1)$, so the outer loop is parallelizable.

More generally, the p^{th} loop in a loop nest is parallelizable if for every distance vector I, $I_p = 0 \vee \exists q < p : I_q > 0$. In Figure 5 (b), the distance vectors are $\{(1, 0), (1, -1)\}$, so the inner loop is parallelizable. This is because both references on the right-hand side of the expression read elements of `a` from row `i-1`, which was written during the previous iteration of the outer loop. Therefore the elements of row `i` may be calculated and written in any order.

6.1.1 Loop Interchange

Loop interchange exchanges the position of two loops in a loop nest, generally moving one of the outer to loops to the innermost position [Wolfe 1989b; Allen and Kennedy 1987; Allen and Kennedy 1984]. It is one of the most powerful transformations and can improve performance in many ways. However, many compilers do not perform this optimization.

Loop interchange may be performed to:

- enable vectorization by interchanging an inner, dependent loop with an outer, independent loop;
- improve vectorization by moving the independent loop with the largest range into the innermost position;
- improve parallel performance by moving an independent loop outwards in a loop nest to increase the granularity of each iteration and reduce the number of barrier synchronizations;
- move more scalars into the inner loop to increase data access locality and hence register reuse;
- reduce stride, ideally to stride-1; and
- increase the number of loop-invariant expressions in the inner loop.

Care must be taken that these benefits do not cancel each other out. For instance, an interchange that improves register re-use may change

```
do i = 1,n
  do j = 1,n
    total[i] = total[i] + a[i,j]
  end do
end do
```

(a) original loop nest

```
do j = 1,n
  do i = i,n
    total[i] = total[i] + a[i,j]
  end do
end do
```

(b) interchanged loop nest

Figure 7: Loop Interchange

a stride-1 access pattern to a stride-n access pattern with much lower overall performance due to increased cache misses. Table 6 demonstrates the dramatic effect of different strides on an IBM RS/6000 [IBM 1992].

In Figure 7 (a), the inner loop accesses array `a` with stride-`n`. By interchanging the loops, we convert the inner loop to stride-1 access (b).

For a large array in which less than one column fits in the cache, this optimization reduces the number of cache misses on `a` from $\mathtt{n}^2$ to $\mathtt{n} \times$ elementsize/linesize, or `n`/16 with 4-byte elements and 64-byte lines. However, the original loop allows `total[i]` to be placed in a register, eliminating the load/store operations in the inner loop. So the optimized version increases the number of load/store operations for `total` from 2`n` to $2\mathtt{n}^2$. If `a` fits in the cache, the original loop is better.

On a vector architecture without a reduction primitive, the transformed loop enables vectorization by eliminating the dependence on `total[i]` in the inner loop.

Interchanging loops is legal when the altered dependences are legal and the loop bounds can be switched. If two loops p and q in a perfect loop nest of d loops are interchanged, each dependence vector $V = (v_1, \ldots, v_p, \ldots, v_q, \ldots, v_d)$ in the original loop nest becomes $V' = (v_1, \ldots, v_q, \ldots, v_p, \ldots, v_d)$ in the transformed loop nest. If V' is lexicographically positive, then the dependence relationships of the original loop are satisfied.

```
do i = 2, n
  do j = 1, n-1
    a[i,j] = a[i-1,j+1]
  end do
end do
```

(a)

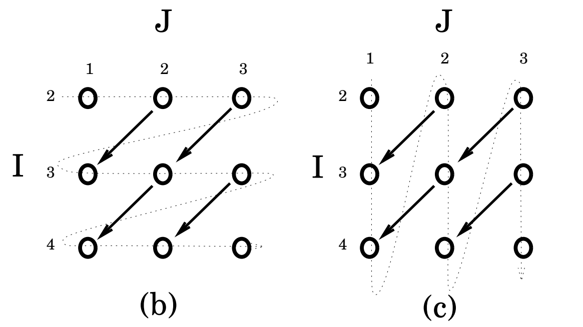


Figure 8: Original loop (a), original traversal order (b), traversal order after interchange (c)

A loop nest of just two loops can be interchanged unless a dependence vector of the form $(<, >)$ exists. This is shown in Figure 8: the loop nest (a) has dependence $(1, -1)$, which gives rise to the loop-carried dependences shown in (b). The order in which the iterations are executed is shown by the dotted line. After reversal (c) iterations are executed before iterations that they depend upon, so the interchange is illegal.

Switching the loop bounds is straightforward when the iteration space is rectangular, as in the loop nest in Figure 7. In this case the loop bounds are independent of one another, and can simply be exchanged. When the iteration space is not rectangular, computing the bounds is more complex. Triangular spaces are often used by programmers and trapezoidal spaces are introduced by loop skewing (discussed in the next section). A further set of techniques are necessary to manage imperfectly nested loops. Some of the variations are discussed in detail by Wolfe [1989b] and Wolf and Lam [1991].

6.1.2 Loop Skewing

Loop skewing is an enabling transformation which is primarily useful in combination with loop interchange [Wolfe 1989b]. Because it only alters the loop bounds, it does not change the computations being performed and is always legal by itself.

Skewing was invented to handle wavefront computations, so called because the updates to the array propagate like a wave across the array [Muraoka 1971; Lamport 1974]. In Figure 9 (a) we show a typical wavefront computation. Each element is computed by averaging its four nearest neighbors. While neither of the loops is parallel in their original form, each array diagonal ("wavefront") can be computed in parallel. The iteration space and dependences are shown in (c), with the dotted lines indicating the wavefronts.

Skewing is performed by simply adding the outer loop index, multiplied by a *skew factor* f, to the bounds of the inner iteration variable, and then subtracting the same quantity from every use of the inner iteration variable inside the loop. The result when $f = 1$ is shown in Figure 9 (b). The transformed code is equivalent to the original, but the effect on the iteration space is to align the diagonal wavefronts of the original loop nest so that for a given value of `j`, all iterations in `i` can be executed in parallel (d).

The original loop nest can be interchanged but neither loop can be parallelized. The skewed loop nest can be interchanged. After skew and interchange, the loop has dependences $\{(1,0),(1,1)\}$. The first dependence allows the inner loop to be parallelized because the corresponding dependence distance is 0. The second dependence allows the inner loop to be parallelized because it is a dependence on previous iterations of the outer loop.

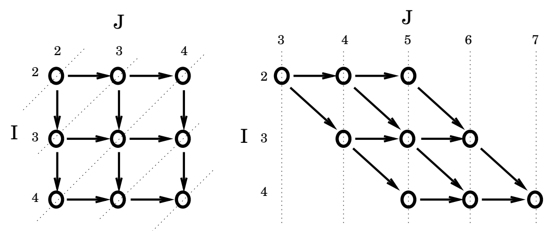
Interchanging skewed loops is complicated by the fact that their bounds depend on the iteration variables of the enclosing loops. For two loops with bounds `i=LI,UI` and `j=LJ,UJ`, where `LJ` and `UJ` are expressions independent of `i`, the skewed inner loop has bounds `j=f*i+LJ, f*i+UJ`.

```
do i = 2, n-1
  do j = 2, m-1
    a[i,j] = (a[i-1,j] + a[i,j-1] +
                a[i+1,j] + a[i,j+1])/4
  end do
end do
```

(a) original code: dependences $\{(1,0),(0,1)\}$

```
do i = 2, n-1
  do j = i+2, i+m-1
    a[i,j-i] = (a[i-1,j-i] + a[i,j-i-1] +
                  a[i+1,j-i] + a[i,j-i+1])/4
  end do
end do
```

(b) skewed code: dependences $\{(1,1),(0,1)\}$



(c) original space (d) skewed space

```
do j = 4, m+n-2
  do i = max(2,j-m+1), min(n-1,j-2)
    a[i,j-i] = (a[i-1,j-i] + a[i,j-i-1] +
                  a[i+1,j-i] + a[i,j-i+1])/4
  end do
end do
```

(e) skewed and interchanged code: dependences $\{(1,0),(1,1)\}$

Figure 9: Loop Skewing

```
do i = 1, n
  do j = 1, n
    a[i,j] = a[i-1, j+1] + 1
  end do
end do
```

(a) Original loop nest: distance vector $(1, -1)$. Interchange is not possible.

```
do i = 1, n
  do j = n, 1, -1
    a[i,j] = a[i-1, j+1] + 1
  end do
end do
```

(b) Inner loop reversed: direction vector $(1, 1)$. Loops may be interchanged.

Figure 10: Loop Reversal

```
do i=1, n
  do j=1, n
    a[i,j] = b[j,i]
  end do
end do
```

(a) original loop

```
do TI=1, n, 64
  do TJ=1, n, 64
    do i=TI, min(TI+63, n)
      do j=TJ, min(TJ+63, n)
        a[i,j] = b[j,i]
      end do
    end do
  end do
end do
```

(b) blocked loop

Figure 11: Loop Blocking

After interchange, the bounds are `j=f*LI+LJ`, `f*UI+UJ` and `i=max(LI,⌈(j-IJ)/f⌉), min(UI, ⌈(j-LJ)/f⌉)`.

An alternative method for handling wavefront computations is *supernode partitioning* [Irigoin and Triolet 1988].

6.1.3 Loop Reversal

Reversal changes the direction in which the loop traverses its iteration range [Wedel 1975]. It is often used in conjunction with other loop iteration space reordering transformations because it changes the dependence vectors [Wolfe 1989b].

As an optimization in its own right, reversal can reduce loop overhead by eliminating the need for a compare instruction on architectures without a compound compare and branch (such as the Alpha [Sites 1992]). The loop is reversed so the iteration variable runs down to zero, allowing the loop to end with a branch if not equal to zero instruction (`BNEZ`).

Reversal can also eliminate the need for temporary arrays in implementing Fortran-90 array statements; this is discussed in more detail in section 6.2.10.

If loop p in a nest of d loops is reversed, then for each dependence vector V, the entry v_p is negated. The reversal is legal if each resulting vector V' is lexicographically positive, that is when $v_p = 0$ or $\exists q < p : v_q > 0$.

For instance, the inner loop of a loop nest with dependences $\{(<, =), (<, >)\}$ can be reversed, because the resulting dependences are all still lexicographically positive.

Figure 10 shows how reversal can enable loop interchange. The original loop nest (a) has the distance vector $(1, -1)$ which prevents interchange because the resulting distance vector $(-1, 1)$ is not lexicographically positive; the reversed loop nest (b) can legally be interchanged.

6.1.4 Loop Blocking

Blocking (also called *tiling*) is used to improve memory locality, primarily the cache [Abu-Sufah et al. 1981; Gannon et al. 1988; Wolfe 1989a; Lam et al. 1991]. However, it can also be used to improve processor, register, TLB, or page locality.

The need for blocking is illustrated by the loop in Figure 11 (a) which assigns `a` the transpose of `b`. With the `j` loop innermost, access to `b` is stride-1, while access to `a` is stride-`n`. Interchanging does not help, since it makes access to

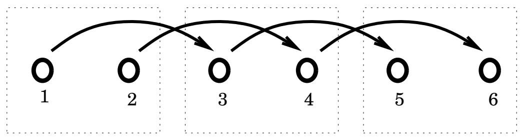
```
  do i = 1, n
1   a[i+k] = b[i]
2   b[i+k] = a[i] + c[i]
  end do
```

(a) Because of the write to a, $S_1 \xrightarrow{(k)} S_2$; because of the write to b, $S_2 \xrightarrow{(k)} S_1$

.

```
  do TI = 1, n, k
    do all i = TI, TI+k-1
1     a[i+k] = b[i]
2     b[i+k] = a[i] + c[i]
    end do all
  end do
```

(b) k iterations can be performed in parallel because that is the minimum dependence distance.



(c) Iteration space when n = 6 and k = 2.

Figure 12: Cycle Shrinking

b stride-n. By iterating over a sub-rectangle of the iteration space, every cache line is fully used (b).

The inner two loops of a matrix multiply have this structure; blocking is critical for achieving high performance in dense matrix multiplication.

A pair of adjacent loops can be blocked if they can legally be interchanged. After blocking, the outer pair of loops can be interchanged to improve locality across blocks, and the inner loops can be exchanged to exploit inner-loop parallelism or register locality.

6.1.5 Cycle Shrinking

Cycle shrinking is a transformation which can expose relatively fine-grained parallelism. When a loop has dependences that prevent it from being executed in parallel (that is, converted to a **do all**), it may still be possible to expose some parallelism if the dependence distance is greater than one. In this case cycle shrinking converts a serial loop into an outer serial loop and an inner parallel loop [Polychronopoulos 1987a].

For instance, in Figure 12 (a), a[i+k] is written in iteration i and read in iteration i+k; the dependence distance is k. Consequently the first k iterations can be performed in parallel provided that none of the subsequent iterations is allowed to begin until the first k are complete. The same is then done for the next k iterations, as shown in (b). The iteration space dependences are shown in (c): each group of k iterations is only dependent on the previous group.

The result is potentially a speedup by a factor of k, but k is likely to be small (usually 2 or 3), so this optimization is normally only able to expose parallelism that can be exploited at the instruction level. Note that k must be constant within the loop and must at least be known to be positive at compile time.

6.2 Other Loop Transformations

6.2.1 Strip Mining

Strip mining is a method of adjusting the granularity of an operation, especially a parallelizable operation [Loveman 1977; Abu-Sufah et al. 1981; Allen 1983]. It is most commonly used on vector machines to convert a serial loop into a series of vector operations, each vector comprising a single "strip".

An example is shown in Figure 13. The strip-mined computation is expressed in array notation, and is equivalent to a **do all** loop. Cleanup code is needed if the iteration length is not evenly divisible by the strip length. Since it is used to parallelize a loop, strip mining generally requires that the inner loop be parallelizable.

If the inner loop contains more than one statement, *loop distribution* can be applied to create loops containing single statements (see Section 6.2.8).

In general strip mining allows the compiler to choose the number of independent computations in the innermost loop; this loop is then converted into a single operation, which is often paral-

```
do i=1, n
  a[i] = a[i] + c
end do
```

(a) original loop

```
TN = (n/64)*64
do TI=1, TN, 64
  a[TI:TI+63] = a[TI:TI+63] + c
end do
do i=TN+1, n
  a[i] = a[i] + c
end do
```

(b) after strip mining

```
; R9 = address of a[TI]
LV      V1, R9       ; V1 <- a[TI:TI+63]
ADDSV   V1, F8, V1   ; V1 <- V1 + c (F8=c)
SV      V1, R9       ; a[TI:TI+63] <- V1
```

(c) vector assembly code for the inner `do all` loop

Figure 13: Strip Mining

lel. Strip mining has been used for vectorization (for instance by the Cray CF77 compiler [Cra 1988]), for SIMD compilation (CM-2 CMF compiler [Bromley et al. 1991]), for combining **send** operations in a loop on distributed-memory multiprocessors [Hiranandani et al. 1992], and for limiting the size of compiler-generated temporary arrays [Abu-Sufah 1979; Wolfe 1989b].

Loop interchange is often important for strip mining, either to make the inner loop a **do all**, or to maximize the length of the strip.

6.2.2 Inverse Strip Mining

Strip mining is used to create a larger unit of work out of smaller ones. When it is necessary to decrease the size of the unit of work, inverse strip mining is used.

For instance, on a distributed memory multiprocessor, sending messages that exceed the hardware buffer size may cause the processor to block until the receiving processor acknowledges receipt of the first buffer. In this case, it is desirable to interleave computation between the sending of the buffers, as is shown in Figure 14. This use has been proposed for the Fortran-D compiler [Hiranandani et al. 1991].

```
SEND(RIGHT, a,1,n)
RECEIVE(LEFT, a,1,n)
do i = 1, n
  a[i] = a[i] + c
end do
```

(a) original code

```
do TI = 1, n, 256
  SEND(RIGHT, a,TI,TI+255)
  RECEIVE(LEFT, a,TI,TI+255)
  do i = TI, TI+255
    a[i] = a[i] + c
  end do
end do
```

(b) loop nest after transformation

Figure 14: Inverse Strip Mining

6.2.3 Strength Reduction of Induction Variable Expressions

A variable whose value is determined only by the number of iterations of an enlosing loop is called an *induction variable.* The loop control variable of a **do** statement is the most common kind of induction variable, but other variables may also be induction variables.

Strength reduction is a standard optimization that replaces an expensive operation with a cheaper one. The most common use of strength reduction, often implemented as a special case, is strength reduction of induction variable expressions [Allen 1969; Cocke and Schwartz 1970; Allen et al. 1981; Aho et al. 1986]. General strength reduction is covered in Section 6.7.1.

Strength reduction can be applied to products involving induction variables by converting them to an equivalent running sum, as shown in Figure 15. This is most important on architectures in which integer multiply operations take more cycles than integer additions (current examples include the SPARC [Sun Microsystems

1991] and the Alpha [Sites 1992]). Strength reduction may also make other optimizations possible, in particular the elimination of induction variables, as is shown in the next section.

6.2.4 Induction Variable Elimination

Once strength reduction has been performed on induction variable expressions, it is often possible to eliminate the original induction variable entirely. This is done by expressing the loop exit test in terms of one of the induction variables and is called *linear function test replacement* [Allen 1969; Aho et al. 1986].

The replacement not only reduces the number of operations in a loop, but it frees the register used by the induction variable.

Figure 15 (d) shows the result of applying induction variable elimination.

6.2.5 Loop-invariant Code Motion

Code motion is applied to a computation inside a loop whose result does not change between iterations. In this case, the computation is moved outside of the loop [Cocke and Schwartz 1970; Aho et al. 1986].

Code motion can be applied at a high level to expressions in the source code, or at a low level to address computations. The latter is particularly relevant when indexing multi-dimensional arrays or dereferencing pointers, as when the inner loop contains an expression such as `a.b->c.d[i]`. Most commercial compilers do code motion.

Figure 16 (a) shows an example in which an expensive transcendental function call is moved outside of the inner loop. The test in the transformed code (b) ensures that if the loop is never executed, the moved code is not executed either, lest it raise an exception.

The pre-computed value is generally assigned to a register. If registers are scarce ($\Pi \geq 1$) and the expression moved is inexpensive to compute, code motion may actually de-optimize the code, since register spills will be introduced in the loop.

Although code motion is sometimes referred to as *code hoisting*, hoisting is a more general term which refers to any transformation that

```
do i = 1, n
  a[i] = a[i] + c
end do
```
(a) original code

```
     LF    F4, c(R30)  ;load c into F4
     LW    R8, n(R30)  ;load n into R8
     LI    R9, #1      ;set i (R9) to 1
     ADDI  R12,R30, #a ;R12=address(a[1])
Loop:MULTI R10, R9, #4 ;R10=i*4
     ADDI  R10,R12,R10 ;R10=address(a[i+1])
     LF    F5, -4(R10) ;load a[i] into F5
     ADDF  F5, F5, F4  ;a[i]:=a[i]+c
     SF    -4(R10), F5 ;store new a[i]
     SLT   R11, R9, R8 ;R11 = i<n?
     ADDI  R9, R9, #1  ;i=i+1
     BNEZ  R11, Loop   ;if i<n, goto Loop
```
(b) initial compiled loop

```
     LF    F4, c(R30)  ;load c into F4
     LW    R8, n(R30)  ;load n into R8
     LI    R9, #1      ;set i (R9) to 1
     ADDI  R10,R30, #a ;R10=address(a[1])
Loop:LF    F5, (R10)   ;load a[i] into F5
     ADDF  F5, F5, F4  ;a[i]:=a[i]+c
     SF    (R10), F5   ;store new a[i]
     ADDI  R9, R9, #1  ;i=i+1
     ADDI  R10, R10,#4 ;R10=address(a[i+1])
     SLT   R11, R9, R8 ;R11 = i<n?
     BNEZ  R11, Loop   ;if i<n, goto Loop
```
(c) after strength reduction – `R10` is a running sum instead of being recomputed from `R9` and `R12`

```
     LF    F4, c(R30)  ;load c into F4
     LW    R8, n(R30)  ;load n into R8
     ADDI  R10, R30,#a ;R10=address(a[1])
     MULTI R8, R8, #4  ;R8=n*4
     ADDI  R8, R10, R8 ;R8=address(a[n+1])
Loop:LF    F5, (R10)   ;load a[i] into F5
     ADDF  F5, F5, F4  ;a[i]:=a[i]+c
     SF    (R10), F5   ;store new a[i]
     ADDI  R10, R10,#4 ;R10=address(a[i+1])
     SLT   R11, R10,R8 ;R11= R10<R8?
     BNEZ  R11, Loop   ;if R11, goto Loop
```
(d) after elimination of induction variable (`R9`)

Figure 15: Induction Variable Optimizations

```
do i = 1,n
  a[i] = a[i] + sqrt(x)
end do
```

(a) original loop

```
if (n > 0) C = sqrt(x)
do i = 1,n
  a[i] = a[i] + C
end do
```

(b) after code motion

Figure 16: Loop-invariant code motion.

```
do i=2, n-1
  a[i] = a[i] + a[i-1] * a[i+1]
end do
```

(a) original loop

```
do i=2, n-2, 2
  a[i] = a[i] + a[i-1] * a[i+1]
  a[i+1] = a[i+1] + a[i] * a[i+2]
end do

if (mod(n-2,2) = 1) then
  a[n-1] = a[n-1] + a[n-2] * a[n]
end if
```

(b) loop unrolled twice

Figure 17: Loop Unrolling

moves an expression to a point where it is executed earlier [Aho et al. 1986]. The most common use of hoisting is to move loop-invariant expressions, but it can also be particularly useful in optimizing automatically generated programs which tend to have many repeated expressions.

6.2.6 Loop Unrolling

Unrolling replicates the body of a loop some number of times called the unrolling factor (u), and then iterates by step u instead of step 1. This transformation can improve the code by

- reducing loop overhead;
- increasing instruction parallelism; and
- improving register, data cache, or TLB locality.

The benefits of unrolling have been studied on several different architectures [Dongarra and Hind 1979]; it is a fundamental technique for generating the long instruction sequences required by VLIW machines [Ellis 1986].

In Figure 17, we show all three of these improvements in an example. Loop overhead is cut in half because two iterations are performed before the test and branch at the end of the loop. Assuming that array elements are assigned to registers, register locality is improved because `a[i]` and `a[i+1]` are used twice in the loop body, reducing the number of loads per iteration from 3 to 2. Instruction parallelism is increased because the second assignment can be performed while the results of the first are being stored and the loop variables are being updated.

If the target machine has double- or multi-word loads, unrolling often allows several loads to be combined into one.

Unrolling has the advantage that it can be applied to any loop, and can be done profitably both at the high and low levels. Some compilers also perform *loop re-rolling* prior to unrolling because programs often contain loops that were unrolled for a different target architecture.

Unrolling can also be done for outer loops, and then combined with *fusion* (described in Section 6.2.9) of the u inner loops [Allen and Cocke 1971]. *Loop quantization* is another approach which allows outer loops to be unrolled without introducing u inner loops [Nicolau 1988].

Figure 18 shows the effects of unrolling in more detail. The original loop (b) takes 6 cycles per result on S-DLX. After unrolling 3 times (c), it takes 8 cycles per iteration or $2\frac{2}{3}$ cycles per result. The original loop stalls for one cycle waiting for the load, and for two cycles waiting for the `ADDF` to complete. In the unrolled loop some of these cycles are filled.

Most compilers for high-performance machines will unroll at least the innermost loop of a nesting.

```
        do i=1, n
          a[i] = a[i] + c
        end do
```

(a) the initial loop

```
Cycle

1    LW    R8, n(R30)   ADDI  R10, R30,#a ;load n into R8      ;R10=address(a[1])
2    LF    F4, c(R30)                     ;load c into F4
3    MULTI R8, R8, #4                     ;R8=n*4
5    ADDI  R8, R10, R8                    ;R8=address(a[n+1])

1  L:LF    F5, (R10)    ADDI  R10, R10,#4 ;load a[i] into F5   ;R10=address(a[i+1])
3    ADDF  F5, F5, F4   SLT   R11, R10,R8 ;a[i]=a[i]+c         ;R11= R10<R8?
6    SF    -4(R10), F5  BNEZ  R11, L      ;store new a[i]      ;if R11, goto L
```

(b) the compiled loop body for S-DLX. This is the loop from Fig 15 (d) after instruction scheduling.

```
1  L:LF    F5, (R10)                      ;load a[i] into F5
2    LF    F6, 4(R10)                     ;load a[i+1] into F6
3    LF    F7, 8(R10)   ADDF  F5, F5, F4  ;load a[i+2] into F8 ;a[i]=a[i]+c
4    ADDI  R10,R10,#12  ADDF  F6, F6, F4  ;R10=address(a[i+3]) ;a[i+1]=a[i+1]+c
5    SLT   R11, R10,R8  ADDF  F7, F7, F4  ;R11= R10<R8?        ;a[i+2]=a[i+2]+c
6    SF    -12(R10),F5                    ;store new a[i]
7    SF    -8(R10), F6                    ;store new a[i+1]
8    SF    -4(R10), F7  BNEZ  R11, L      ;store new a[i+2]    ;if R11, goto L
```

(c) after unrolling 3 times (loop epilogue omitted)

```
1    LW    R8, n(R30)   ADDI  R10, R30,#a ;load n into R8      ;R10=address(a[1])
2    LF    F4, c(R30)   SUBI  R8, R8, #2  ;load c into F4      ;R8=n-2
3    MULTI R8, R8, #4                     ;R8=(n-2)*4
5    ADDI  R8, R10, R8  LF    F5, (R10)   ;R8=address(a[n-1])  ;F5=a[1]
7    ADDF  F6, F5, F4   LF    F5, 4(R10)  ;F6=a[1]+c           ;F5=a[2]

1  L:SF    (R10), F6    ADDI  R10, R10,#4 ;store new a[i]      ;R10=address(a[i+1])
2    ADDF  F6, F5, F4   SLT   R11, R10,R8 ;a[i+1]=a[i+1]+c     ;R11= R10<R8?
3    LF    F5, 4(R10)   BNEZ  R11, L      ;load a[i+2] into F5 ;if R11, goto L
4    [stall]
```

(d) after software pipelining (loop epilogue omitted)

```
1  L:SF    (R10), F6    ADDF  F6, F5, F4  ;store new a[i]      ;a[i+2]=a[i+2]+c
2    SF    4(R10), F8   ADDF  F8, F7, F4  ;store new a[i+1]    ;a[i+3]=a[i+3]+c
3    LF    F5, 16(R10)  ADDI  R10, R10,#8 ;load a[i+4] into F5 ;R10=address(a[i+2])
4    LF    F7, 12(R10)  SLT   R11, R10,R8 ;load a[i+5] into F7 ;R11= R10<R8?
5    BNEZ  R11, L                         ;if R11, goto L
```

(e) after unrolling 2 times and software pipelining (loop prologue and epilogue omitted)

Figure 18: Increasing instruction parallelism in loops.

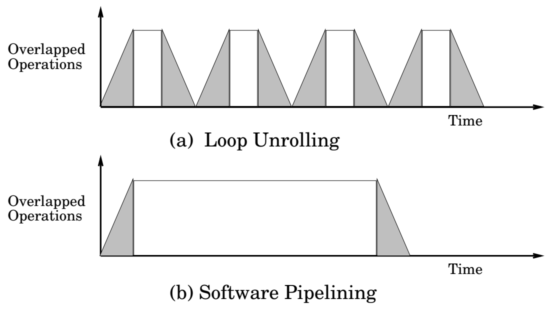


Figure 19: Loop Unrolling vs. Software Pipelining

6.2.7 *Software Pipelining*

Another technique to improve instruction parallelism is software pipelining [Lam 1988]. In hardware pipelining, instruction execution is broken into stages, such as Fetch, Execute, and Write-back. The first instruction is fetched in the first clock cycle. In the next cycle, the second instruction is fetched while the first is executed, and so on. Once the pipeline has been filled, the machine will complete 1 instruction per cycle.

In software pipelining, the operations of a single loop iteration are broken into s stages, and a single iteration performs stage 1 from iteration `i`, stage 2 from iteration `i-1`, etc. Startup code must be generated before the loop to initialize the pipeline for the first $s-1$ iterations and cleanup code must be generated after the loop to drain the pipeline for the last $s-1$ iterations.

Figure 18 (d) shows how software pipelining improves the performance of a simple loop. The depth of the pipeline is $s = 3$. Figure 19 illustrates the difference between unrolling and software pipelining: unrolling reduces overhead, while pipelining reduces the startup cost of each iteration.

Finally, Figure 18 (e) shows the result of combining software pipelining ($s = 3$) with unrolling ($u = 2$). The loop takes 5 cycles per iteration, or $2\frac{1}{2}$ cycles per result. Unrolling alone achieves $2\frac{2}{3}$ cycles per result. If software pipelining is combined with unrolling by $u = 3$, the resulting loop would take 6 cycles per iteration or two cycles per result, which is the best that can be done because only one memory operation can be initiated per cycle.

Perfect Pipelining combines unrolling by loop quantization with software pipelining [Aiken and Nicolau 1988a; Aiken and Nicolau 1988b]. A special case of software pipelining is *predictive commoning*, which is applied to memory operations. If an array element written in iteration $i-1$ is read in iteration i, then the first element is loaded outside of the loop and from then on there is one load and one store per iteration. This is done by the RS/6000 XL C/Fortran compiler [O'Brien et al. 1990].

If there are no loop-carried dependences, the length of the pipeline is the length of the dependence chain. If there are multiple, independent dependence chains they can be scheduled together subject to resource availability, or *loop distribution* can be applied to put each chain into its own loop (see the following section). The scheduling constraints in the presence of loop-carried dependences and conditionals are more complex; the details are discussed in [Aiken and Nicolau 1988a; Lam 1988].

6.2.8 *Loop Distribution*

Distribution (also called *loop fission* or *loop splitting*) breaks a single loop into multiple loops with the same iteration space but each enclosing only a subset of the statements of the original loop [Muraoka 1971; Kuck 1977; Kuck et al. 1981].

Distribution is used to

- create perfect loop nests;
- create sub-loops with fewer dependences;
- improve instruction cache and instruction TLB locality due to shorter loop bodies;
- reduce memory requirements by iterating over fewer arrays; and
- increase register re-use by decreasing register pressure;

Figure 20 is an example in which distribution removes dependences and allows part of a loop to be run in parallel.

```
do i=1, n
  a[i] = a[i]+c
  x[i+1] = x[i]*7 + x[i+1] + a[i]
end do
```

(a) original loop

```
do all i=1, n
  a[i] = a[i]+c
end do all
do i=1, n
  x[i+1] = x[i]*7 + x[i+1] + a[i]
end do
```

(b) after loop distribution

Figure 20: Loop Distribution

Distribution may be applied to any loop, but all statements belonging to a dependence cycle (called a π-block [Kuck 1977]) must be placed in the same loop, and if $S_1 \Rightarrow S_2$ in the original loop, then the loop containing S_1 must precede the loop that contains S_2. If the loop contains control flow, applying if-conversion (see Section 5.2) can expose greater opportunities for distribution. An alternative is to use a control-dependence graph [Kennedy and McKinley 1990].

A specialized version of this transformation is *distribution by name*, originally called *horizontal distribution of name partition* [Abu-Sufah et al. 1981]. Rather than performing full dependence analysis on the loop, the statements are partitioned into sets that reference mutually exclusive variables. These statements are guaranteed to be independent.

When the arrays in question are large, fission by name can increase cache locality. Note that the above loop can not be distributed using fission by name, since both statements reference a.

6.2.9 Loop Fusion

The inverse transformation of distribution is fusion (also called *jamming*) [Ershov 1966]. It can improve performance by

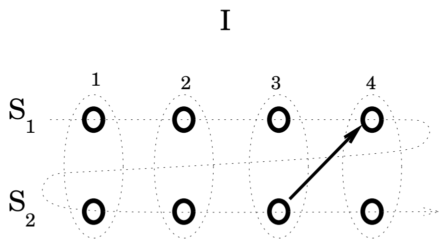


Figure 21: Two loops containing S_1 and S_2 can not be fused when $S_2 \Rightarrow S_1$ in the fused loop

- reducing loop overhead;
- increasing instruction parallelism;
- improving register, vector [Wolfe 1989b], data cache, TLB, or page [Abu-Sufah 1979] locality; and
- improving the load balance of parallel loops.

In the previous section distribution has enabled parallelization of part of the loop. However, fusing the two loops back together will improve register and cache locality since a[i] need only be loaded once; it will increase instruction parallelism by increasing the ratio of floating point operations to integer operations in the loop; and it will reduce loop overhead by a factor of two. With large n, the distributed loop will run fastest on a vector machine, and the fused loop will run fastest on a superscalar machine.

To fuse two loops, they must have the same loop bounds; when the bounds are not identical, it is sometimes possible to make them identical by *peeling* (described in Section 6.2.15) or by introducing conditional expressions into the body of the loop. Two loops with the same bounds may be fused if there does not exist statement S_1 in the first loop and S_2 in the second such that they have a dependence $S_2 \overset{(<)}{\Longrightarrow} S_1$ in the fused loop. The reason this would be incorrect is that before fusing, all instances of S_1 execute before any S_2. After fusing, corresponding instances are executed together. If any instance of S_1 has a dependence on (i.e. must be executed after) any subsequent instance of S_2, the fusion illegally alters execution order, as shown in Figure 21.

6.2.10 Array Statement Scalarization

When a loop is expressed in array notation, the compiler can either convert it into vector operations or *scalarize* it into one or more serial loops [Wolfe 1989b]. However, the conversion is not completely straightforward because array notation implies that every operation is performed concurrently.

The example in Figure 22 shows a computation in array notation (a), its obvious (and incorrect) conversion to serial form (b), and a correct conversion (c). The reason that (b) is not correct is that in the original code, every element of `a` is to be incremented by the value of the previous element. This is to happen simultaneously; in the incorrect version (b), each element is incremented by the updated value of the previous element.

The general solution is to introduce a temporary array `T` and to have a separate loop that writes the values back into `a`, as shown in (c). The temporary array can then be eliminated if the two loops can be legally fused, namely when there is no anti-dependence $S_2 \overset{(<)}{\not\rightarrow} S_1$ in the fused loop, where S_1 is the assignment to the temporary and S_2 is the assignment to the original array. Note that flow and output dependences can not arise from array language statements.

In this case there is an anti-dependence, but it can be removed by reversing the loops, enabling fusion and eliminating the temporary, as shown in (d). However, the array language statement in (e) requires a temporary since an anti-dependence exists regardless of the direction of the loop.

```
a[2:n-1] = a[2:n-1] + a[1:n-2]
```
(a) initial array language expression

```
do i = 2, n-1
  a[i] = a[i] + a[i-1]
end do
```
(b) incorrect scalarization

```
  do i = 2, n-1
1   T[i] = a[i] + a[i-1]
  end do
  do i = 2, n-1
2   a[i] = T[i]
  end do
```
(c) correct scalarization

```
do i = n-1, 2, -1
  a[i] = a[i] + a[i-1]
end do
```
(d) reversing both loops allows fusion and eliminates need for temporary array `t`

```
a[2:n-1] = a[2:n-1] + a[1:n-2] + a[3:n]
```
(e) array expression requiring a temporary

Figure 22: Array Statement Scalarization

6.2.11 Loop Coalescing

Coalescing combines a loop nest into a single loop, with the original indices computed from the resulting single induction variable [Polychronopoulos 1987b; Polychronopoulos 1988]. This can improve the scheduling of the loop on a parallel machine, and may also reduce loop overhead.

For example, if `n` and `m` are slightly larger than the number of processors P, then neither of the loops schedules well as the outer parallel loop,

```
do i=1, n
  do j=1, m
    a[i,j] = a[i,j] + c
  end do
end do
```

(a) original loop

```
do all T=1, n*m
  i = ((T-1) / m)*m + 1
  j = MOD(T-1, m) + 1
  a[i,j] = a[i,j] + c
end do all
```

(b) coalesced loop

```
real TA[n*m]
equivalence (TA,a)
do all T = 1, n*m
  TA[T] = TA[T] + c
end do all
```

(c) collapsed loop

Figure 23: Loop Coalescing versus Collapsing

since executing the last $n-P$ iterations will take the same time as the first P. Coalescing the two loops ensures that we can execute P iterations every time except during the last $nm \bmod P$ iterations, as shown in Figure 23 (b).

Coalescing itself is always legal since it does not change the iteration order of the loop. If the iterations of the coalesced loop are parallelized, all dependences for the original loops must be $=$, or there be a positive dependence distance in an enclosing loop.

The complex subscript calculations can often be simplified to reduce the overhead that this introduces [Polychronopoulos 1987b].

6.2.12 Loop Collapsing

Collapsing is a simpler, more efficient, but less general version of coalescing in which the number of dimensions of the array is actually reduced. Collapsing eliminates the overhead of multiple nested loops and multi-dimensional array indexing.

Collapsing is used not only to increase the number of parallelizable loop iterations, but also to increase vector lengths and to eliminate the overhead of a nested loop (also called the *Carry* optimization [Allen and Cocke 1971; IBM 1991]).

The collapsed version of the loop discussed in the previous section is shown in Figure 23 (c).

Collapsing is best suited to loop nests that iterate over memory with a constant stride. When more complex indexing is involved, coalescing may be a better approach.

6.2.13 Loop Unswitching

Loop unswitching is applied when a loop contains a conditional with a loop-invariant test condition. The loop is then replicated inside each branch of the conditional, saving the overhead of conditional branching inside the loop, reducing the code size of the loop body, and possibly enabling the parallelization of a branch of the conditional [Allen and Cocke 1971].

Conditionals that are candidates for unswitching can be detected during code motion, which identifies loop-invariant values.

In Figure 24 the variable **x** is loop invariant, allowing the loop to be unswitched and the **true** branch to be executed in parallel, as shown in Figure 24. Note that as with loop-invariant code motion, if there is any chance that the condition evaluation will cause an exception, it must be guarded by a test that the loop will be executed.

In a loop nest where the inner loop has unknown bounds, if code is generated straightforwardly there will be a test before the body of the inner loop to determine if it should be executed at all. The test for the inner loop will be repeated every time the outer loop is executed. If the intermediate representation of the program is sufficiently detailed to reveal this, then unswitching can be performed to move this test outside of the outer loop. The RS/6000 XL C/Fortran compiler implements unswitching for this reason [O'Brien et al. 1990].

```
do i=2, n
  a[i] = a[i] + c
  if (x < 7) then
    b[i] = a[i] * c[i]
  else
    b[i] = a[i-1] * b[i-1]
  end if
end do
```

(a) original loop

```
if (n > 0) then
  if (x < 7) then
    do all i=2, n
      a[i] = a[i] + c
      b[i] = a[i] * c[i]
    end do all
  else
    do i=2, n
      a[i] = a[i] + c
      b[i] = a[i-1] * b[i-1]
    end do
  end if
end if
```

(b) after unswitching

Figure 24: Loop Unswitching

```
do i=1, n
  call f(x,n)
end do

subroutine f(a, j)
real a[*]
a[j] = a[j] + c
return
```

(a) original loop and procedure

```
call F_2(x)

subroutine F_2(a)
real a[*]
do all i=1, n
  a[i] = a[i] + c
end do all
return
```

(b) after loop pushing

Figure 25: Loop Pushing

6.2.14 *Loop Pushing*

Pushing moves a loop nest from the caller to a cloned version of the called procedure. Very few commercial compilers perform vectorization or parallelization across procedure calls directly; this transformation is a way of achieving a similar effect, although it is less general.

Pushing is done by the CMAX Fortran preprocessor for the Thinking Machines CM-5. CMAX converts Fortran-77 programs to Fortran-90, attempting to discover data-parallel operations in the process [Sabot and Wholey 1993].

Pushing not only allows the parallelization of the loop in Figure 25, it also eliminates the overhead of all but one of the procedure calls.

If there are other statements in the loop, distribution is a prerequisite to pushing. In this case, however, the dependence analysis for distribution must be inter-procedural. If there are no other statements in the loop, the transformation is always legal.

Procedure inlining (see Section 6.6.6) is a different way of achieving a very similar effect, and does not require interprocedural analysis.

6.2.15 *Loop Peeling*

Peeling has two uses: to remove dependences created by the first or last few loop iterations, thereby enabling parallelization, and to match the iteration control of adjacent loops to enable fusion.

The loop in Figure 26 (a) is not parallelizable because of a flow dependence between iteration `i = 2` and iterations `i = 3...n`. Peeling off the first iteration allows the rest of the loop to be parallelized and fused with the following loop (b).

Since peeling simply breaks a loop into sections without changing the iteration order, it can be applied to any loop.

```
do i = 2, n
  b[i] = b[i] + b[2]
end do
do all i = 3, n
  a[i] = a[i] + c
end do all
```

(a) original loops

```
if (2 <= n) then
  b[2] = b[2] + b[2]
end if
do all i=3, n
  b[i] = b[i] + b[2]
  a[i] = a[i] + c
end do all
```

(b) after peeling one iteration from first loop and fusing the resulting loops

Figure 26: Loop Peeling

```
do i = 1, n
  a[i] = a[i] + c
end do

do i = 2, n+1
  b[i] = a[i-1] * b[i]
end do
```

(a) original loops

```
do i = 1, n
  a[i] = a[i] + c
end do

do i = 1, n
  b[i+1] = a[i] * b[i+1]
end do
```

(b) after normalization, the two loops can be fused

Figure 27: Loop Normalization

6.2.16 Loop Normalization

Normalization converts all loops so that the induction variable is initially 1 (or 0), and is incremented by 1 on each iteration [Allen and Kennedy 1987]. This transformation can expose opportunities for fusion and simplify inter-loop dependence analysis, as shown in Figure 27. It can also help to reveal which loops are candidates for peeling followed by fusion.

The most important use of normalization, however, is to permit the compiler to apply subscript analysis tests, most of which work with normalized iteration ranges.

6.2.17 Reduction Recognition

A reduction is an operation that computes a scalar value from an array. Common reductions include computing either the sum or the maximum value of the elements in an array. In Figure 28 (a), the sum of the elements of `a` are accumulated in the scalar `s`. The dependence vector for the loop is (1), or ($<$). While a loop with direction vector ($<$) must normally be executed serially, reductions can be parallelized if the operation performed is associative. Commutativity provides additional opportunities for reordering.

When the compiler transforms code under the assumption that addition is an associative and commutative operation, the reordered program will not necessarily yield exactly the same result. Provided that bit-wise identical results are not required, the partial sums can be computed in parallel. In Figure 28 (b), the reduction has been vectorized by using vector adds (the inner **do all** loop) to compute `TS`; the final result is computed from `TS` using a scalar loop.

For semi-commutative and semi-associative operators like floating point multiplication, the validity of the transformation depends upon the language semantics and the programmer's intent (see Section 2.1).

The maximum parallelism can be achieved by computing the reduction with a tree: pairs of elements are summed, then pairs of these results are summed, and so on. This reduces the number of serial steps from $O(\mathtt{n})$ to $O(\log \mathtt{n})$.

Operations such as **and**, **or**, **min**, and **max** are truly associative and their reduction can be parallelized under all circumstances.

```
do i = 1, n
  s = s + a[i]
end do
```

(a) a sum reduction loop

```
real TS[64]

TS[1:64] = 0.0

do TI = 1, n, 64
  TS[1:64] = TS[1:64] + a[TI:TI+63]
end do
do TI = 1, 64
  s = s + TS[TI]
end do
```

(b) loop transformed for vectorization

Figure 28: Reduction Recognition

6.2.18 Loop Idiom Recognition

Parallel architectures often provide specialized hardware that the compiler can take advantage of. For example, SIMD machines frequently support reduction directly in the processor interconnection network. Some parallel machines, such as the Connection Machine [Thi 1989], include hardware not only for reduction but for parallel prefix operations, allowing a loop of the form `a[i] = a[i-1] + a[i]` to be parallelized. The parallelization of a more general class of linear recurrences is described by Chen and Kuck [1975], and discussed by Kuck [1977; 1978] and Wolfe [1989b]. Blelloch [1989] describes compilation strategies for recognizing and exploiting parallel prefix operations.

Other idioms that are specially supported by hardware or software can be recognized and converted by the compiler. For instance, the CMAX Fortran preprocessor converts loops implementing vector and matrix operations into calls to assembly-coded BLAS (Basic Linear Algebra Subroutines) [Sabot and Wholey 1993], and the VAX Fortran compiler converts string copy loops into block transfer instructions [Harris and Hobbs to appear].

```
    do i = 1, n/2
1     a[i+1] = a[i+1] + a[i]
    end do
    do i = 1, n-3
2     b[i+1] = b[i+1] + b[i] + a[i+3]
    end do
```

(a) original loops

```
do i = 1, n/2
  COBEGIN
    a[i+1] = a[i+1] + a[i]
    if (i > 3) then
      b[i-2] = b[i-2]+b[i-3]+a[i]
    end if
  COEND
end do
do i = n/2-3,n-3
  b[i+1] = b[i+1] + b[i] + a[i+3]
end do
```

(b) after spreading

Figure 29: Loop Spreading

6.2.19 Loop Spreading

Spreading takes two serial loops and moves some of the computation from the second to the first so that the bodies of both loops can be executed in parallel [Girkar and Polychronopoulos 1988a]. While similar to loop fusion in combining the bodies of two loops, in a fused loop there may be dependences between the original loop bodies *within* an iteration, which would prevent the two loop bodies from being executed in parallel.

An example is shown in Figure 29: the two loops in the original program (a) cannot be fused because they have different bounds and there would be a dependence $S_2 \xrightarrow{(2)} S_1$ in the fused loop due to the write to a in S_1 and the read of a in S_2. By executing the statement S_2 three iterations later within the new loop, it is possible to execute the two statements in parallel, which we have indicated textually with the COBEGIN/COEND compound statement (b).

The number of iterations by which the body of the second loop must be delayed is the maximum dependence distance between any statement in the second loop and any statement in the first loop, plus 1. Adding one ensures that there are no dependences within an iteration. For this reason, there must not be any scalar dependences between the two loop bodies.

Unless the loop bodies are large, spreading is primarily beneficial for exposing instruction-level parallelism. Depending on the amount of instruction parallelism achieved, the introduction of a conditional may negate the gain due to spreading. The conditional can be removed by peeling the first few (in this case 3) iterations off the first loop, but this introduces more loop overhead.

6.2.20 *Flattening*

Flattening is a load-balancing transformation for SIMD machines. An array may be allocated to the processors in such a way that each processor has approximately the same number of elements, but there is a significant variation in the number of elements in each row (or whatever dimension is stored entirely locally). Flattening checks after each iteration whether the end of the local row has been reached, and advances the row index if necessary [von Hanxleden and Kennedy 1992].

6.3 Memory Access Transformations

High-performance applications are as frequently memory-limited as they are compute-limited. In fact, for the last fifteen years CPU speeds have doubled every three to five years, while DRAM speeds have doubled in speed about once every decade (DRAMs, or Dynamic Random Access Memory chips, are the low-cost, low-power memory chips used for main memory in most computers).

As a result, optimization of the use of the memory system has become steadily more important. Factors affecting memory performance include:

- Re-use, denoted by Q, the ratio of uses of an item to the number of times it is loaded (see Section 3). This may be applied to any element of the memory hierarchy: scalar registers, vector registers, cache lines, etc.;

- Parallelism. Vector machines often divide memory into *banks*, allowing vector registers to be loaded in a parallel or pipelined fashion. Superscalar machines often support double- or quad-word load and store instructions;

- Working Set Size. If all of the memory elements accessed inside of a loop do not fit in the data cache, then items that will be accessed in later iterations may be flushed, decreasing Q. If more variables are simultaneously live than there are available registers (that is, the register pressure $\Pi > 1$), then loads and stores will have to spill values into memory, decreasing Q. If more pages are accessed in a loop than there are entries in the TLB, then the TLB will thrash.

- Page Locality. DRAMs often support "page-mode" access: a consecutive memory access to the same page allows much of the address decode logic to be by-passed, leading to a 3 or 4 times faster access (currently, 20ns instead of 70ns).

Since the registers are the top of the memory hierarchy, efficient register usage is absolutely crucial to high performance. Until the late seventies, register allocation was considered the single most important problem in compiler optimization for which there was no adequate solution. The introduction of techniques based on graph-coloring [Chaitin et al. 1981; Chaitin 1982; Chow and Hennessy 1990] yielded very efficient global (within a procedure) register allocation. Most compilers make use of some variant of graph coloring in their register allocator.

Optimizations covered in other sections that also can improve memory system performance are loop interchange (6.1.1), loop blocking (6.1.4), loop unrolling (6.2.6), loop fusion (6.2.9), and various optimizations which eliminate register saves at procedure calls (6.6).

```
real a[8,512]

do i=1, 8
  a[i,1] = a[i,1] + c
end do
```

Figure 30: Memory Alignment. If a is aligned on a cache-line boundary, the loop takes 64 cycles to execute on S-DLX; if not, it takes 80 cycles.

6.3.1 Memory Alignment

Aligning an array on cache line or page boundaries reduces the number of cache lines or TLB entries used for the array. This both increases Q and reduces the chance that the cache or TLB will thrash.

For example, each column of array a in Figure 30 has 8 elements, consuming 64 bytes – exactly one cache line. If the array is aligned on a 64-byte boundary, the loop over the first column will incur one cache miss instead of two. If the loop is compiled for S-DLX as in Figure 18 (b) on page 22, it will take 6 cycles per element, or 48 cycles for the entire loop, plus the cache miss costs. A cache miss costs 16 cycles, so if a is aligned there is one cache miss and the loop takes 64 cycles, but if it is not aligned there are two cache misses and the loop takes 80 cycles, or 25% more time.

6.3.2 Array Padding

It is often desirable to have every column of an array be cache-aligned; in this case the compiler can increase the size of the columns so that once the array itself is aligned, every column begins on a cache line boundary.

Padding is especially important for vector machines with banked memory, such as the Cray and our hypothetical DLX-V. If indexing through an array dimension that is not laid out contiguously in memory, memory may be accessed with a stride that is an exact multiple of the number of banks. The bandwidth of the memory system will be reduced by a factor of 8 because all memory accesses are to the same bank.

```
real a[8,512]

do i = 1, 512
  a[1,i] = a[1,i] + c
end do
```

(a) original code

```
real a[9,512]

do i = 1, 512
  a[1,i] = a[1,i] + c
end do
```

(b) padding a eliminates memory bank conflicts on DLX-V

Figure 31: Array Padding

In general, if an array will be accessed with stride n, the array should be padded by the smallest p such that $\gcd(n+p, banks) = 1$. This will ensure that $banks$ successive stride $n+p$ accesses will all access different banks. An example is shown in Figure 31: the original loop accesses memory with stride 8, so all memory references will be to the first bank (a). After padding successive iterations access memory with stride 9, so they go to successive banks (b).

A similar effect can occur in cache-based machines when the stride is a high power of two. This can cause successively accessed elements to map into the same cache line set because the low bits of the address are identical. This would reduce the effective cache size of S-DLX to 4 lines (since the cache is 4-way set-associative).

The disadvantages of padding are that it increases memory consumption and makes the subscript calculations for operations over the whole array more complex, since the array has "holes". In particular, it reduces the benefits of loop collapsing (see Section 6.2.12).

6.3.3 Code Co-location

Code co-location is an optimization which improves instruction cache utilization by placing the most frequent successor to a basic block (or the most frequent callee of a procedure) immediately adjacent to it in instruction memory [Pettis and Hansen 1990; Hwu and Chang 1989].

An estimate is made of the frequency with which each arc in the control flow graph will be traversed during program execution (using either profiling information or static estimates). Procedures are grouped together using a greedy algorithm that always takes the pair of procedures (or procedure groups) with the largest number of calls between them.

Within a procedure, basic blocks can be grouped in the same way (although the direction of the control flow must be taken into account), or a top-down algorithm can be used which starts from the procedure entry node. Basic blocks with a frequency estimate of zero can be moved to a separate page to further increase locality. However, this may require long displacement jumps to be introduced (see the next subsection), creating the potential for performance loss if the basic blocks in question are actually executed.

Inlining can also affect code locality, and has been studied both in conjunction with [Hwu and Chang 1989] and independent of code positioning [McFarling 1991]. Inlining often improves performance by reducing overhead and increasing locality, but if a procedure is called more than once in a loop inlining will increase the number of cache misses because the procedure body will be loaded more than once.

6.3.4 Displacement Minimization

The target of a branch or a jump is usually specified relative to the current value of the program counter (PC). The largest offset that can be specified varies among architectures; it can be as little as 8 bits. If control is transferred to a location outside of the range of the offset, a multi-instruction sequence is required to synthesize the jump. For instance, on S-DLX

```
      BEQZ R4, error
```

if `error` is more than 2^{15} bytes away, the instruction must be replaced with:

```
      BNEZ R4, cont
      LI   R8, error ;get low bits
      LUI  R8, error>>16 ;get hi bits
      JR   R8         ;jump to target
cont:
```

This sequence requires three extra instructions but no memory reference. Given the cost of long-displacement jumps, the code should be organized to keep related sections close together in memory, in particular those sections referred to by the code which is executed most frequently [Szymanski 1978].

Displacement minimization can also be applied to data. For instance, a base register may be allocated for a Fortran common block or group of blocks:

```
common /foo/ q, r, a[20000], y, z
```

Common block `foo` is larger than the amount of memory indexable by the offset field in the load instruction (2^{16} bytes on S-DLX). To address `y` and `z`, multiple instruction sequences must be used in a manner analogous to the long jump sequences above. This can be avoided by laying out `foo` as

```
common /foo/ q, r, y, z, a[20000]
```

6.3.5 Scalar Expansion

Loops often contain variables that are used as temporaries within the loop body. Such variables will create an anti-dependence $S_2 \overset{(<)}{\longrightarrow\!\!\!\!\!\!\!+} S_1$ from one iteration to the next, and will have no other loop-carried dependencies. By allocating one temporary for each iteration, the dependence is removed, making the loop a candidate for parallelization [Padua et al. 1980; Wolfe 1989b], as shown in Figure 32. If the final value of `c` is used after the loop, `c` must be assigned the value of `T[n]`.

Scalar expansion is a fundamental technique for vectorizing compilers, and was performed by the Burroughs Scientific Processor and the Cray-1 compilers.

If the compiler vectorizes or parallelizes a loop, scalar expansion must be performed for any compiler-generated temporaries in a loop.

```
do i=1, n
  c = b[i]
  a[i] = a[i] + c
end do
```
(a) original loop

```
real T[n]

do all i=1, n
  T[i] = b[i]
  a[i] = a[i] + T[i]
end do all
```
(b) after scalar expansion

Figure 32: Scalar Expansion

Some languages and dialects allow variables to be declared whose scope is only the loop body, thereby allowing the programmer to declare variables for scalar expansion explicitly.

Scalar expansion can also increase instruction-level parallelism by removing dependences.

6.3.6 Array Expansion

An extension of scalar expansion is array expansion [Feautrier 1988]. The array is expanded by adding an additional dimension to it.

6.3.7 Array Contraction

After transformation of a loop nest, it may be possible to contract scalars or arrays that have previously been expanded. It may also be possible to contract other arrays due to interchange or the use of redundant storage allocation by the programmer [Wolfe 1989b].

If the iteration variable of a loop p is being used to index the k^{th} dimension of an array x, then dimension k may be removed from x if (1) loop p is not parallel, (2) all dependences V involving x have $V_p = 0$, and (3) x is not used subsequently (that is, x is *dead* after the loop). The latter two conditions are true for compiler-expanded variables unless the loop structure of the program was changed after expansion. In particular, loop distribution can inhibit array contraction by causing the second condition to be violated.

```
real T[n,n]

do i=1, n
  do all j=1, n
    T[i,j] = a[i,j]*3
    b[i,j] = T[i,j] + b[i,j]/T[i,j]
  end do all
end do
```
(a) original code

```
real T[n]

do i=1, n
  do all j=1, n
    T[j] = a[i,j]*3
    b[i,j] = T[j] + b[i,j]/T[j]
  end do all
end do
```
(b) after array contraction

Figure 33: Array Contraction

Contraction reduces the amount of storage consumed by compiler-generated temporaries, as well as reducing the number of cache lines referenced. Other methods for reducing storage consumption by temporaries are strip mining (see Section 6.2.1) and dynamic allocation of temporaries, either from the heap or from a static block of memory reserved for temporaries.

6.3.8 Scalar Replacement

Even when it is not possible to contract an array into a scalar, a similar optimization can be performed when a frequently referenced array element is invariant within the innermost loop or loops. In this case, the array element can be loaded into a scalar (and presumably therefore a register) before the inner loop and, if it is modified, stored after the inner loop.

This multiplies Q for the scalar-replaced array element by the number of iterations in the inner loop(s). It can also eliminate unneces-

```
do i = 1,n
  do j = 1,n
    total[i] = total[i] + a[i,j]
  end do
end do
```
(a) original loop nest

```
do i = 1,n
  T = total[i]
  do j = 1,n
    T = T + a[i,j]
  end do
  total[i] = T
end do
```
(b) after scalar replacement

Figure 34: Scalar Replacement

sary subscript calculations, although this is often done by loop-invariant code motion (see Section 6.2.5). Loop interchange can be used to enable or improve scalar replacement, but other effects must be taken into consideration.

An example of scalar replacement is shown in Figure 34; for a discussion of its interactions with loop interchange, see Section 6.1.1.

6.4 Partial Evaluation

Partial evaluation refers to the general technique of performing part of a computation at compile time. Most of the classical dataflow analysis-based optimizations are either forms of partial evaluation or of redundancy elimination (described in Section 6.5).

6.4.1 Constant Propagation

Constant propagation [Kildall 1973; Wegman and Zadeck 1991; Callahan et al. 1986] is one of the most important optimizations that a compiler can perform and any optimizing compiler will do so aggressively. Programs typically contain many constants; by propagating them through the program, the compiler can do a significant amount of pre-computation. More importantly, the propagation reveals many opportunities for other optimizations. In addition to obvious possibilities like dead code elimination, loop optimizations are much affected because constants often appear in their induction ranges. Knowing the range of the loop, the compiler can much more accurately perform the loop optimizations which more than anything else determine performance on high-speed architectures.

```
n = 64
c = 3
do i = 1, n
  a[i] = a[i] + c
end do
```
(a) original code

```
do i = 1, 64
  a[i] = a[i] + 3
end do
```
(b) after constant propagation

Figure 35: Constant Propagation

Figure 35 shows a simple example of propagation. On DLX-V, the resulting loop can be converted into a single vector operation because the loop is the same length as the hardware vector registers. The original loop would have to be strip-mined before vectorization (see Section 6.2.1), increasing the overhead of the loop.

6.4.2 Constant Folding

Constant folding is a companion to constant propagation; when an expression contains a computation on constants, that computation is performed at compile time. For example, `x = 3.1**2` becomes `x = 9.61`. Typically constants are propagated and folded simultaneously [Aho et al. 1986].

6.4.3 Copy Propagation

Optimizations such as induction variable elimination (6.2.4) and common subexpression elimination (6.5.4) may cause the same value to be copied several times. Copy propagation propagates the original name of the value and elimi-

```
t = i*4
s = t
print *, a[s]
r = t
a[r] = a[r] + c
```

(a) original code

```
t = i*4
print *, a[t]
a[t] = a[t] + c
```

(b) after copy propagation

Figure 36: Copy Propagation

```
np1 = n+1
do i = 1, n
  a[np1] = a[np1] + a[i]
end do
```

(a) original code

```
do all i = 1, n
  a[n+1] = a[n+1] + a[i]
end do all
```

(b) after forward substitution

Figure 37: Statement Substitution

nates redundant copies [Aho et al. 1986].

Copy propagation reduces register pressure and eliminates redundant register-to-register move instructions. An example is shown in Figure 36.

6.4.4 Statement Substitution

Statement substitution is a generalization of copy propagation. The use of a variable is replaced by its defining expression which is live at that point. This can change the dependence relation between variables [Wolfe 1989b] or improve the analysis of subscript expressions in loops [Kuck et al. 1981; Allen and Kennedy 1987].

For instance, in Figure 37 (a) the loop cannot be parallelized because an unknown element of a is being written. After forward substitution (b), the subscript expression is in terms of the loop bound variable, and it is straightforward to determine that the loop can be implemented as a parallel reduction (described in Section 6.2.17).

The use of variables like np1 is a common Fortran idiom that was developed when compilers did not routinely perform code motion. This idiom is recommended as "good programming style" in a number of Fortran programming texts!

Statement substitution is generally performed on array subscript expressions at the same time as loop normalization (Section 6.2.16). For efficient subscript analysis techniques to work, the array subscripts must be linear functions of the induction variables.

6.4.5 Reassociation

Reassociation is a technique for increasing the number of common subexpressions in a program [Cocke and Markstein 1980; Markstein et al. to appear]. It is generally applied to address calculations within loops when performing strength reduction on induction variable expressions (see Section 6.2.3). Address calculations generated by array references consist of several multiplications and additions. Reassociation applies the associative, commutative, and distributive laws to rewrite these expressions in a canonical sum-of-products form.

Statement substitution is usually performed where possible in the address calculations to increase the number of potential common subexpressions.

6.4.6 Algebraic Simplification

The compiler can simplify arithmetic expressions by applying algebraic rules to them. A particularly useful example is the set of algebraic identities. For instance, the statement `x = (y*1+0)/1` can be transformed into `x = y`. Figure 38 illustrates some of the commonly applied rules.

While identity operations are generally guaranteed to leave their operands unchanged, other

$$\begin{aligned} x \times 0 &= 0 \\ 0/x &= 0 \\ x \times 1 &= x \\ x + 0 &= x \\ x/1 &= x \end{aligned}$$

Figure 38: Some algebraic identities used in expression simplification

simplifications can change the results of the expression as discussed in Section 2.1.

6.4.7 Function Cloning

When some of the arguments to a function are constants, the function can be *cloned*, with the parameters replaced by their constant values. Constant propagation can then be used to expose other optimizations, such as algebraic simplification, dead code elimination (6.5.1), and strength reduction (6.7.1).

In Figure 39 cloning procedure `f` with `p` replaced by the constant 2 allows reduction in strength. The real-valued exponentiation is replaced by a multiplication, which is usually at least 10 times faster.

```
call f(a, n, 2)

procedure f(x, n, p)
real x[*]
integer n, p
do i = 1, n
  x[i] = x[i]**p
end do
```

(a) original code

```
call F_2(a, n)

procedure F_2(x, n)
real x[*]
integer n
do i = 1, n
  x[i] = x[i]*x[i]
end do
```

(b) after cloning

Figure 39: Function Cloning

6.4.8 I/O Format Compilation

Fortran provides complex format specification for character output that is in effect a formatting sublanguage. **format** statements are generally "interpreted" at run-time, with a correspondingly high cost for character I/O.

The same issue arises with C's `printf` and `scanf` functions, although it is complicated by the fact that they are library functions and may be redefined by the programmer.

Formatted writes can be converted almost directly into calls to the run-time routines that implement the various format styles. These calls are then likely candidates for inline substitution. Figure 40 shows two I/O statements and their compiled equivalents. The conversion of the implied **do** loop into an aggregate operation is actually a separate optimization which essentially performs strip mining of input/output operations.

Format compilation is done by the VAX Fortran compiler [Harris and Hobbs to appear] and by the Gnu C compiler [Fre 1992].

Note that in Fortran, a **format** statement is essentially a procedure definition, which may be invoked by any number of **read** or **write** statements. The same trade-off as with procedure inlining applies: the formatted I/O can be expanded inline for higher efficiency, or encapsulated as a procedure for code compactness.

6.5 Redundancy Elimination

There are a variety of optimizations which improve performance by identifying redundant computations and removing them [Morel and Renvoise 1979]. We have already covered one such transformation, loop-invariant code motion, in Section 6.2.5. There a computation was being performed many times when it could be done once.

Redundancy-eliminating transformations remove two kinds of computations: those that are

```
    write(6,100) c[i]
    read(7,100) (d(j),j=1,100)
100 format(A1)
```

(a) original code

```
    call putchar(c[i], 6)
    call fgets(d,100,7)
```

subcaption(b) after format compilation

Figure 40: Format Compilation

unreachable and those that are *useless*. A computation is unreachable if it is never executed; removing it from the program will have no effect on the instructions executed. Unreachable code is sometimes created by programmers (most frequently with conditional debugging code), but more often by previous transformations that left "orphan" code behind.

A computation is useless if none of the outputs of the program are dependent on it.

6.5.1 Unreachable Code Elimination

Most compilers perform unreachable code elimination [Allen and Cocke 1971; Aho et al. 1986]. In structured programs, there are two primary ways for code to become unreachable. If a conditional predicate is known to be true or false, one branch of the conditional is never taken and its code can be eliminated. The other common source of unreachable code is a loop that does not perform any iterations.

In an unstructured program that relies on **goto** statements to transfer control, unreachable code is not obvious from the program structure but can be found by traversing the control flow graph of the program.

Both unreachable and useless code are often created by *constant propagation*, described in Section 6.4.1. In Figure 41 (a), the variable `debug` is a constant. When its value is propagated, the conditional expression becomes `if (0 > 1)`; this expression is always false, so the body of the conditional is never executed and can be eliminated, as shown in (b). Similarly, the body of the **do** loop is never executed and is therefore removed.

Unreachable code elimination can in turn allow another iteration of constant propagation to discover more constants; for this reason some compilers perform constant propagation more than once.

Unreachable code is also known as *dead code* but that name is also applied to useless code, so we have chosen to use the more specific term.

Sometimes considered as a separate step is *redundant control elimination*, which removes control constructs like loops and conditionals when they become redundant (usually as a result of constant propagation). In Figure 41 (b), the loop and conditional control expressions are not used and we can remove them from the program, as shown in (c).

6.5.2 Useless Code Elimination

Useless code is often created by other optimizations, like unreachable code elimination. When the compiler discovers that the value being computed by a statement is not necessary, it can remove the code. This can be done for any nonglobal variable that is not *live* immediately after the defining statement. Live variable analysis is a well-know dataflow problem [Aho et al. 1986]. In Figure 41 (c), the values computed by the assignment statements are no longer used. They have been eliminated in (d).

6.5.3 Dead Variable Elimination

After a series of transformations, particularly loop optimizations, there are often variables whose value is never used. The unnecessary variables are called *dead* variables; eliminating them is a common optimization [Aho et al. 1986].

In Figure 41 (d), the variables `c`, `n`, and `debug` are no longer used and can be removed; (e) shows the code after the variables are pruned.

6.5.4 Common Subexpression Elimination

In many cases, a set of computations will contain identical sub-expressions. This is true both of user code and of address computations generated by the compiler. The compiler can store the value of the sub-expression rather than re-

```
integer c, n, debug
debug = 0
n = 0
a = b+7
if (debug > 1) then
  c = a + b + d
  print *, 'Warning -- total is ', c
end if
call foo(a)
do i = 1, n
  a[i] = a[i] + c
end do
```

(a) original code

```
integer c, n, debug
debug = 0
n = 0
a = b+7
if (0 > 1) then
end if
call foo(a)
do i = 1, 0
end do
```

(b) after constant propagation and unreachable code elimination

```
integer c, n, debug
debug = 0
n = 0
call foo(a)
```

(c) after redundant control elimination

```
integer c, n, debug
a = b+7
call foo(a)
```

(d) after useless code elimination

```
a = b+7
call foo(a)
```

(e) after dead variable elimination

Figure 41: Redundant Code and Variable Elimination

computing it [Cocke 1970; Aho et al. 1977; Aho et al. 1986]. Common sub-expression elimination is one of the most important transformations and is almost universally performed. While it is generally a good idea to perform common subexpression elimination wherever possible, the compiler must consider the current register pressure and the cost of recomputing. If storing the temporary value(s) forces additional spills to memory, the transformation can de-optimize.

6.5.5 Short-Circuiting

Short circuiting is an optimization that can be performed on Boolean expressions. It is based on the observation that the value of many binary Boolean operations can be determined from the value of the first operand [Arden et al. 1962]. For example, the expression

```
x = ((a=1) and (b=2))
```

is known to be false if `a` does not equal 1, regardless of the value of `b`. Short-circuiting would convert this expression to:

```
x = (a=1)
if (x) then x = (b=2)
```

Note that if any of the operands in the boolean expression have side-effects, short circuiting can change the results of the evaluation. The alteration may or may not be legal, depending upon the language semantics. The C language definition addressed this problem by *requiring* short circuiting of Boolean expressions.

6.6 Procedure Call Transformations

The optimizations described in the next several sections attempt to reduce the overhead of procedure calls in one of four ways:

- eliminating the call entirely
- eliminating execution of the called function's body
- eliminating some of the entry/exit overhead
- avoiding some steps in making a procedure call when the behavior of the called procedure is known or can be altered

6.6.1 A Calling Convention for S-DLX

To demonstrate the procedure call optimizations, we will first define a calling convention for S-DLX. Table 1 shows how the registers are used.

In general, each called procedure is responsible for ensuring that the values in registers `R16`–`R25` are preserved across the call. The stack begins at the top of memory and grows downwards. There is no explicit frame pointer; instead, the stack pointer is decremented by the size s of the procedure's frame at entry and left unchanged during the call. The value `R30`$+s$ serves as a *virtual frame pointer* that points to the base of the stack frame, avoiding the use of a second dedicated register. For languages that cannot predict the amount of stack space used during execution of a procedure, an additional general purpose register would be used as a frame pointer.

A similar convention is followed for floating point registers, except that only four are reserved for arguments.

On entering a procedure, the return address is in `R31`. The first six words of the procedure arguments appear in registers `R2`–`R7`, and the rest of the argument data is on the stack. Figure 42 shows the layout of the stack frame for a procedure invocation.

Execution of a procedure consists of six steps:

1. Space is allocated on the stack for the procedure invocation.

2. The values of registers that will be modified during procedure execution (and which must be preserved across the call) are saved on the stack. If the procedure calls any others, the saved registers should include the return address, `R31`.

3. The procedure body is executed.

4. The return value (if any) is stored in `R1` and the registers that were saved in step 2 are restored.

5. The frame is removed from the stack.

6. Control is transferred to the return address.

Calling a procedure is a four step process:

1. The values of any of the registers `R1`–`R15` that contain live values are saved. If the values of any global variables that might be used by the callee are in a register and have been modified, the copy of those variables in memory is updated.

2. The arguments are stored in the designated registers and, if necessary, on the stack.

3. A linked jump is made to the target procedure; the CPU leaves the address of the next instruction in `R31`.

4. Upon return, the saved registers are restored and the registers holding global variables are reloaded.

To demonstrate the structure of a procedure and the calling convention, Figure 43 shows a simple function and its compiled code. The function (`foo`) and the function that it calls (`max`) each take two integer arguments, so they do not need to pass arguments on the stack. The stack frame for `foo` is three words, which are used to save the return address (`R31`) and register `R16` during the call to `max`, and to hold the local variable `d`. `R31` must be preserved because it is overwritten by the jump and link (`JAL`) instruction; `R16` must be preserved because it is used to hold `a` across the call.

The procedure first allocates the 12 bytes for the stack frame and saves `R31` and `R16`. The value of `d` is calculated in the temporary register `R9`. Then the addresses of the arguments are stored in the argument registers and a jump to `max` is made. On return from `max`, the return value has been computed into the return value register (`R1`). After removing the stack frame and restoring the saved registers, the procedure jumps back to its caller through `R31`.

6.6.2 Leaf Procedure Optimization

A *leaf procedure* is one that does not call any other procedures; the name comes from the fact that these procedures are leaves in the call graph. The simplest optimization for leaf procedures is that they do not need to save and

Number	*Usage*
`R0`	Always zero; writes are ignored
`R1`	Return value when returning from a procedure call
`R2..R7`	The first six words of the arguments to the procedure call
`R8..R15`	8 caller save registers. Used as temporary registers by callee
`R16..R25`	10 callee save registers. These registers must be preserved across a call
`R26..R29`	Reserved for use by the operating system
`R30`	Stack pointer
`R31`	Return address during a procedure call
`F0..F3`	The first four floating point arguments to the procedure call
`F4..F17`	14 caller save floating point registers
`F18..F31`	14 callee save floating point registers

Table 1: S-DLX Registers and Their Usage

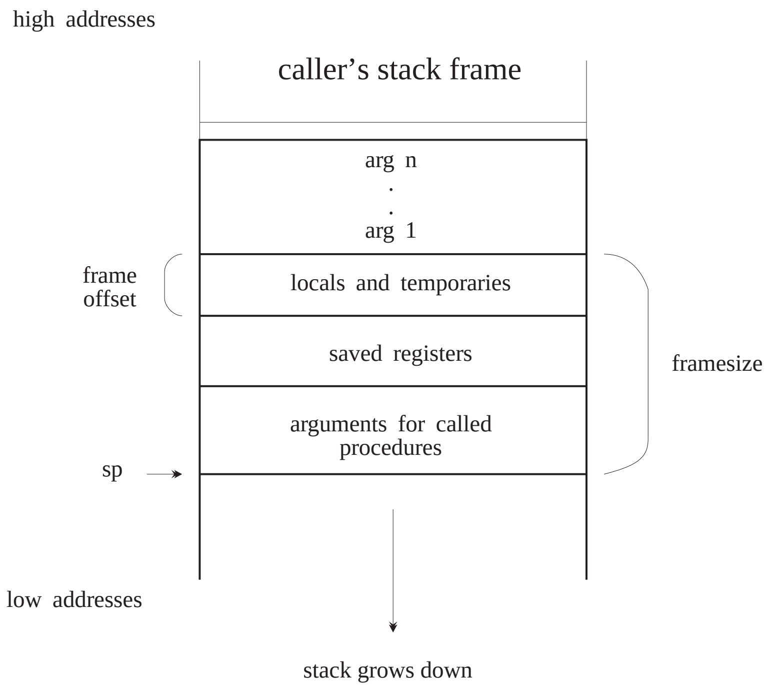


Figure 42: Stack Frame Layout

```
integer function foo(a, b)     foo: SUBI R30, #12    ;adjust SP
integer a, b                        SW   8(R30), R31 ;save retaddr
integer d, e                        SW   4(R30), R16 ;save R16
d = a+b                             LW   R16, (R2)   ;R16=a
                                    LW   R8, (R3)    ;R8=b
                                    ADD  R9, R16, R8 ;R9=d=a+b
e = max(b,d)                        SW   (R30), R9   ;save d
                                    MOV  R2, R3      ;arg1=addr(b)
                                    ADDI R3, R30, #0 ;arg2=addr(d)
                                    JAL  max         ;call max; R1=e
f = e+a                             ADD  R1, R1, R16 ;R1=e+a
return                              LW   R16, 4(R30) ;restore R16
                                    LW   R31, 8(R30) ;restore retaddr
                                    ADDI R30, #12    ;restore SP
end                                 JR   R31         ;return
```

Figure 43: Function `foo` and its Compiled Code

restore the return address (`R31`). In addition, if the procedure does not have any local variables allocated to memory, the compiler does not need to create a stack frame.

Figure 44 (a) shows the `max` function (called by the previous example function `foo`), its original compiled code (b), and the code after leaf procedure optimization (c). After eliminating the save/restore of `R31`, there is no need to allocate a stack frame. Eliminating the code that deallocates the frame also allows the function to return directly if `x < y`.

6.6.3 *Cross-call Register Allocation*

Separate compilation reduces the amount of information available to the compiler about called functions. However, when both callee and caller are available, the compiler can take advantage of the register usage of the callee to optimize the call.

If the callee does not need (or can be restricted not to use) all the temporary registers (`R8`–`R15`), the caller can leave values in the unused registers throughout execution of the callee. In addition, move instructions for parameters can be eliminated.

To perform this optimization, register allocation must be performed in a depth-first postorder traversal of the call graph, ensuring that each caller will know its callees' register usage [Chow 1988].

For example, in Figure 44 (c) `max` uses only `R8`–`R10`. Figure 45 (a) shows `foo` after cross-call register allocation. `R31` is saved in `R11` instead of on the stack, and the return is a jump to the saved address in `R11`. In addition, `R12` is used instead of `R16`, allowing the save and restore of `R16` to be eliminated. Only `d` remains in the stack frame.

6.6.4 *Parameter Promotion*

When a parameter is passed by reference, the address calculation is done by the caller, but the load of the parameter value is done by the callee. This wastes an instruction, since most address calculations can be handled with the `offset(Rn)` format of load instructions.

More importantly, if the operand is already in a register in the caller, it must be spilled to memory and reloaded by the callee. If the callee modifies the value, it must then be stored. Upon return to the caller, if the compiler can not prove that the callee did not modify the operand, it must be loaded again. Thus, as many as two unnecessary loads and two unnecessary stores can be introduced.

```
integer function max(x, y)
integer x, y
if (x > y) then
  max = x
else
  max = y
end if
return
end
```

(a) source code for function max

```
max: SUBI R30, #4     ;adjust SP
     SW   (R30), R3 1 ;save retaddr
     LW   R8, (R2)    ;R8=x
     LW   R9, (R3)    ;R9=y
     SGT  R10, R8, R9 ;R10=x > y
     BEQZ R10, Else   ;x <= y
     MOV  R1, R8      ;max=x
     J    Ret
Else:MOV  R1, R9      ;max=y
Ret: LW   R31, (R30)  ;restore R31
     ADDI R30, #4     ;restore SP
     JR   R31         ;return
```

(b) original compiled code of max

```
max: LW   R8, (R2)    ;R8=x
     LW   R9, (R3)    ;R9=y
     SGT  R10, R8, R9 ;R10=x > y
     BEQZ R10, Else   ;x <= y
     MOV  R1, R8      ;max=x
     JR   R31         ;return
Else:MOV  R1, R9      ;max=y
     JR   R31         ;return
```

(c) max after leaf optimization

Figure 44: Leaf Procedure Optimization

```
foo: SUBI R30, #4     ;adjust SP
     MOV  R11, R31    ;save retaddr
     LW   R12, (R2)   ;R12=a
     LW   R8, (R3)    ;R8=b
     ADD  R9, R12, R8 ;R9=d
     SW   (R30), R9   ;save d
     MOV  R2, R3      ;arg1=addr(b)
     ADDI R3, R30, #0 ;arg2=addr(d)
     JAL  max         ;call max
     ADD  R1, R1, R12 ;R1=e+a
     ADDI R30, #4     ;restore SP
     JR   R11         ;return
```

(a) foo after cross-call register allocation

```
max: SGT  R10, R2, R3 ;R10=x > y
     BEQZ R10, Else   ;x <= y
     MOV  R1, R8      ;max=x
     JR   R31         ;return
Else:MOV  R1, R9      ;max=y
     JR   R31         ;return
```

(b) max after parameter promotion: x and y are passed by value in R2 and R3.

```
foo: MOV  R11, R31    ;save retaddr
     LW   R12, (R2)   ;R12=a
     LW   R2, (R3)    ;R2=b
     ADD  R3, R12, R2 ;R3=d
     JAL  max         ;call max
     ADD  R1, R1, R12 ;R1=e+a
     JR   R11         ;return
```

(c) foo after parameter promotion on max and frame collapsing.

Figure 45: Further procedure call optimizations

An unmodified reference parameter can be passed by value, and a modified reference parameter can be passed by value-result. Figure 45 (b) shows `max` after this transformation has been applied. Figure 45 (c) shows the corresponding modified form of `foo`. Since `d` can now be held in a register, there is no longer a need for a stack frame.

Parameter promotion is particularly important for languages like Fortran, in which all argument passing is by reference.

6.6.5 Frame Collapsing

When a leaf procedure has only one call site, the compiler can expand the stack frame of the caller to include enough space for both procedures. Then the leaf procedure simply uses its caller's stack frame without doing any new allocation of its own.

If the source language is separately compiled, the compiler must be certain that the callee will not be invoked from a different module by a procedure that does not have the proper stack layout. One approach is to clone the procedure, creating a local version with a collapsed frame and an exported one without the optimization.

A special case of frame collapsing applies to procedures without stack variables. Both `foo` in Figure 43 and `max` in Figure 44 (a) are procedures that do not need a stack frame. Their frames can be collapsed without allocating extra space in their caller's stack frame, allowing the optimization to be applied without requiring inter-procedural analysis.

6.6.6 Procedure Inlining

Procedure inlining (also known as *procedure integration*) replaces a procedure call with a copy of the body of the called procedure, replacing each occurrence of a formal parameter with its corresponding actual parameter [Allen and Cocke 1971; Scheifler 1977; Ball 1979]. Inlining can almost always be performed, except when the procedure in question is recursive. For Fortran programs, incompatible common block usages between caller and callee can make inlining more complex and in practice often prevent it.

When a call is inlined, all the overhead for the invocation is eliminated. The stack frame for the caller and callee are allocated together and the transfer of control is eliminated. This is particularly important for the return (`J R31`), since a jump through a register may incur a higher pipeline penalty than a jump to a fixed address.

Another reason for inlining is to improve compiler analysis and optimization. In many compilers, a loop containing a procedure call can not be parallelized because its read-write behavior is unknown. After the call is inlined, the compiler may be able to prove loop independence, thereby allowing vectorization or parallelization. Additionally, register usage may be improved, constants propagated more accurately, and more redundant operations eliminated.

An alternative to inlining is to perform interprocedural analysis. The advantage of interprocedural analysis is that it can be applied uniformly, since it does not cause code expansion the way inlining does. However, many compilers perform little or no interprocedural analysis.

Inlining also affects the instruction cache behavior of the program [McFarling 1991]. The change can be favorable, because locality is improved by eliminating the transfer of control. On the other hand, if a loop body is made much larger, it may no longer fit in the cache and cause additional memory accesses. Further, if the loop contains multiple calls to the same procedure, multiple copies of the procedure will be loaded into the cache.

The primary disadvantage of inlining is that it increases code size, in the worst case exponentially. However, in practice it is simple to control the size increase by selective application of inlining (for example, to small leaf procedures, or to procedures that are only called a few times). The result can be a dramatic improvement in execution speed. Figure 46 shows a source-level example of inlining; Figure 47 shows the assembler output after the procedure `max` is inlined in `foo`.

Ignoring cache effects, if t_p is the time to execute the entire procedure and t_b is the time to execute just the body of the procedure, n is the number of times it is called, and T is the total

```
do i=1, n
  call f(x,n)
end do

subroutine f(a, j)
dimension a[*]
a[j] = a[j] + c
return
```

(a) original code

```
do all i=1, n
  x[i] = x[i] + c
end do all
```

(b) after inlining

Figure 46: Procedure Inlining.

```
foo: LW   R12, (R2)    ;R12=a
     LW   R2, (R3)     ;R2=b
     ADD  R3, R12, R2  ;R3=d

max: SGT  R10, R2, R3  ;R10=x > y
     BEQZ R10, Else    ;x <= y
     MOV  R1, R2       ;max=x
     J    Ret          ;"return" to f
Else:MOV  R1, R3       ;max=y

Ret: ADD  R1, R1, R12  ;R1=e+a
     JR   R31          ;return
```

Figure 47: `max` inlined into `foo`.

execution time of the program, then

$$t_s = n(t_p - t_b)$$

is the time saved by inlining and

$$I = \frac{t_s}{T}$$

is the fraction of the total run-time saved by inlining.

6.6.7 Tail Recursion Elimination

Tail recursion is a particularly common form of recursion. A function is recursive if it invokes itself, directly or indirectly. It is tail recursive if its last act is to call itself and return the value of the recursive call without performing any further processing.

When a function is tail recursive, it is unnecessary to invoke a separate instance with its own stack frame. The recursion can be eliminated; the current invocation will not be using its frame any longer, so the call can be replaced by a jump to the top of the procedure. Figure 48 shows an example of a tail recursive function (a) and the result after the recursion is eliminated (b). A function which is not tail-recursive is shown in (c): it uses the result of the recursive call as an operand to the addition, so there is computation that must be performed after the recursive call returns.

Some languages prevent tail recursion by requiring clean-up code to be executed after a procedure is finished. The semantics of C++, for example, demand that before a procedure returns it must call a deconstructor on each stack-allocated local object variable.

6.6.8 Function Memoization

Memoization is an optimization that is applied to side-effect free procedures (that is, procedures which do not change the state of the program, also called *referentially transparent*). In such cases it is possible to cache the results of recent invocations, and when the procedure is called again with the same arguments, the cached copy is used instead of re-executing the procedure [Michie 1968; Abelson and Sussman 1985].

```
recursive logical function inarray(a,x,i,n)
    real x, a[n]
    integer i, n

    if (i > n) then
      inarray = .FALSE.
    else if (a[i] = x) then
      inarray = .TRUE.
    else
      inarray = inarray(a, x, i+1, n)
    end if
    return
```

(a) A tail-recursive procedure

```
    logical function inarray(a, x, i, n)
    real x, a[n]
    integer i, n

1   if (i > n) then
      inarray = .FALSE.
    else if (a[i] = x) then
      inarray = .TRUE.
    else
      i = i+1
      goto 1
    end if
    return
```

(b) after tail recursion elimination

```
recursive integer function sumarray(a,x,i,n)
    real x, a[n]
    integer i, n

    if (i = n) then
      sumarray = a[i]
    else
      sumarray = a[i]+sumarray(a, x, i+1, n)
    end if
    return
```

(c) A procedure which is not tail-recursive

Figure 48: Tail Recursion Elimination

```
y = f(i)
```

(a) original function call

```
logical f_UNCACHED[n]
real    f_CACHE[n]

if (f_UNCACHED[i]) then
  f_CACHE[i] = f(i)
  f_UNCACHED[i] = .false.
end if
y = f_CACHE[i]
```

(b) code augmented for memoization

Figure 49: Function Memoization.

Figure 49 shows a simple example of memoization. If `f` is often called with the same arguments and `f` also takes a non-trivial amount of time to run, then memoization will substantially increase performance. If not, it will degrade performance and consume memory.

This example assumes that `f`'s parameter is confined to the range 1...`n`, and that `n` is not extremely large. A more sophisticated memoization scheme would hash the arguments and use a cache size that makes a sensible trade-off between re-use and memory consumption. For functions that return dynamically allocated objects, storage management must be considered as well.

For c calls and a hit rate of h, the time to execute the c calls is

$$T = cht_h + c(1-h)t_m$$

where t_h is the time to retrieve the memoized result from the cache (a hit), and t_m is the time to call `f` plus the overhead of discovering that it is not in the cache (a miss). With a high hit rate and a large difference between t_h and t_m, memoization can significantly improve performance.

6.7 Other Transformations

In this section we describe transformations which do not fit into any of the previous categories.

6.7.1 Strength Reductions

Reduction in strength replaces an expression with one that is equivalent but uses a less expensive operator [Allen 1969; Allen et al. 1981].

By far the most common strength reduction is conversion of multiplication to addition in induction variable expressions, as described in Section 6.2.3. However, there are numerous other strength reductions that can be performed, and that result in substantial speedups.

$$\begin{aligned} x \times 2 &= x + x \\ i \times 2^c &= i \ll c \\ i/2^c &= i \gg c \\ x/y &= x \times \frac{1}{y} \\ x^2 &= x \times x \\ x^{c.5} &= x^c \times \sqrt{x} \\ (a,0)+(b,0) &= (a+b,0) \\ (-1)^n & \quad \texttt{T} = -\texttt{T} \end{aligned}$$

Generalizing the first two, multiplication by any integer constant can be performed using only shift and add instructions [Bernstein 1986].

Other strength reductions are possible. For instance, strength reduction can be applied to the Ada string concatenation operator to transform `length(a & b)` to `length(a)+length(b)`. It is also possible to convert exponentiation to multiplication in the evaluation of polynomials, using the identity

$$a_n x^n + a_{n-1}x^{n-1} + \cdots + a_1 x + a_0 = (a_n x^{n-1} + a_{n-1}x^{n-2} + \cdots + a_1)x + a_0.$$

6.7.2 Superoptimizing

A *superoptimizer* [Massalin 1987] represents the extreme of optimization, seeking to replace a sequence of instructions with the optimal alternative. It does an exhaustive search, beginning with a single instruction. If all single instruction sequences fail, two instruction sequences are searched, and so on.

A randomly generated instruction sequence is checked by executing it with a small number of test inputs that were run through the original sequence. If it passes these tests, a thorough verification procedure is applied.

Superoptimization is only practical for short sequences (on the order of a dozen instructions). Superoptimization is particularly useful for eliminating conditional branches in short instruction sequences, where the pipeline stall penalty may be larger than the cost of executing the operations themselves [Granlund and Kenner 1992].

7 PARALLEL SCHEDULING

Many of the transformations in the previous section are used, directly or indirectly, to convert serial **do** loop into a parallel **do all** loop. However, a **do all** loop merely exposes parallelism; a more specific form of the loop is required to map it onto the parallelism available in a particular machine. These mappings are the principal focus of this section.

To execute a computation on multiple processors, both code and data must be distributed appropriately. Although a shared-memory machine may seem to eliminate the need to decompose data, in practice the programmer must pay careful attention to the way data is accessed by the different processors. Shared memory systems rely on various caching strategies to maintain consistency; if locality is ignored while decomposing an application, the overall performance will suffer badly. Shared memory machines do greatly simplify the coding task because the programmer need not specify interprocessor communication explicitly.

7.1 Scheduling Issues

There are basically two ways to schedule a loop on multiple processors: statically or dynamically. A static schedule is determined at compile time and typically uses a regular pattern for assigning iterations to processors. A dynamic strategy makes the decision at run time, allocating iterations or sets of iterations to processors that are free. There are also hybrid strategies which combine the two approaches.

Scheduling can be performed at various levels of *granularity*; the usual metric is the number of instructions executed per scheduling decision. If one decision is sufficient to allocate several thousand cycles of computation, the decision may require many cycles of computation and expensive message traffic. The trade-offs will be different when each decision leads to a few dozen cycles. Generally very fine-grained scheduling is only practical when it is automatically performed by the hardware.

Regardless of the level of granularity, there are three issues that determine the overall performance of a schedule:

- Load Balance. If processors are idle, the application is not taking full advantage of the machine. Scheduling algorithms try to spread the load on the machine evenly.

- Communication. Scheduling decisions result in communication between processors, which can force processors to wait. The schedule should try to minimize the delay by considering the communication pattern of the application. If the architecture of the machine supports simultaneous communication and computation, the schedule should try to take advantage of it.

- Overhead. Dynamic strategies make decisions at run time, based on the execution behavior of the application. They need to acquire information about the state of the machine, determine the schedule, and inform processors of the decisions. These tasks introduce overhead which can limit performance.

7.2 Static Scheduling

Both static and dynamic schedulers face the same problem: decomposing a loop onto a group of processors. Particularly in scientific applications, computations are tightly bound to the data they operate on. If loop bounds are known, and the amount of time needed to perform each iteration of the loop is roughly equal, the loop can be well scheduled statically.

Static decomposition strategies [Cytron 1987; Girkar and Polychronopoulos 1988b; Polychronopoulos and Banerjee 1986] can choose to allocate either data or computations to processors. The most common strategy, which is the one that we will be discussing throughout this section, is to decompose the data arrays, assigning parts of them to each processor. When a loop is executed, each processor computes the iterations that are using data owned locally. This general principle is known as *owner-computes* [Hiranandani et al. 1992]. The arrays being manipulated are allocated to processors by mapping each dimension to a set of processors in a pattern. The four commonly used patterns are discussed in the following sections.

7.2.1 Serial Decomposition

Serial decomposition is the degenerate case, where an entire dimension is allocated to a single processor. In Figure 50 (a), an array is allocated to four processors. Each column is mapped serially, meaning that all the data in that column will be placed on a single processor.

7.2.2 Block Decomposition

A *block* decomposition divides the elements into one group of adjacent elements per processor. Using the example in Figure 50 (a) again, each row of the array is divided between the four processors. Part (b) shows the decomposition if block scheduling is applied across both the horizontal and vertical dimensions.

Figure 51 shows a simple one-dimensional array computation and its block-scheduled implementation on MX-s. The loop simply adds a constant value to each element of the array. There are `n` processors, each of which executes an equal fraction of the computation. Figure 52 shows the same decomposition strategy on MX-d, where additional code is necessary to distribute the constant that is to be added. Note that the example assumes the array `a` has already been distributed across the processors and that processor 0 is controlling the computation.

The advantage of block decomposition is that adjacent elements are usually on the same processor. Because a loop that computes a value for a given element often uses the values of neighboring elements, the blocks have good locality and reduce the amount of communication. On

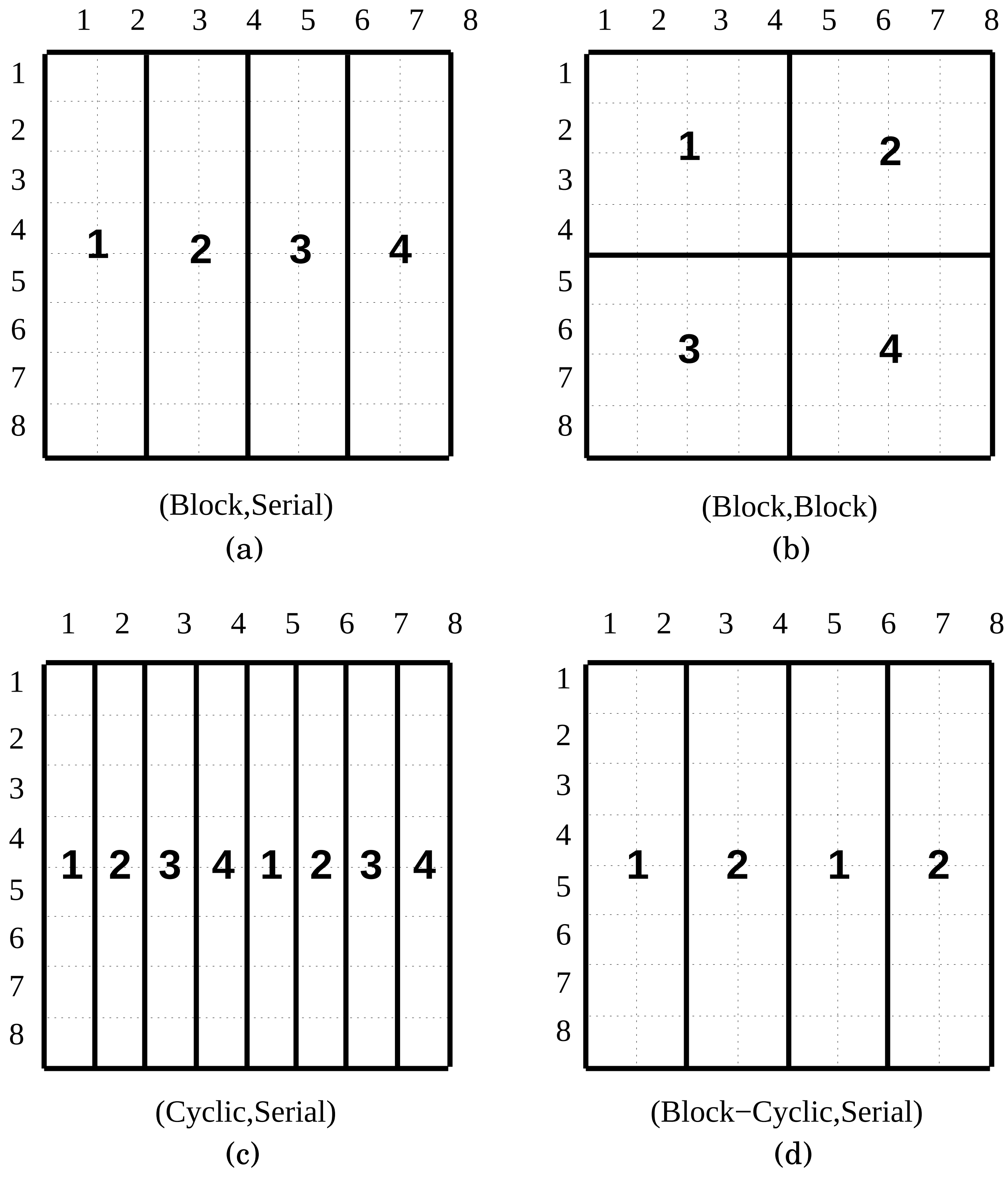


Figure 50: Data Organizations for Static Loop Scheduling. Numbers correspond to the processor assigned to that region of the array.

the other hand, execution time per iteration can also demonstrate locality. For a computation like the one in Figure 53, blocked allocation would work poorly because the most costly iterations to execute would be clustered together and computed by a single processor.

7.2.3 Cyclic Decomposition

A *cyclic* decomposition assigns successive elements to successive processors. Figure 50 (c) shows a cyclic allocation of columns to processors and Figure 51 (c) gives the code to implement such a mapping on MX-s.

Cyclic decomposition has the opposite effect of blocking; it has poor locality for neighbor-based communication, but spreads load more evenly.

7.2.4 Block-Cyclic Decomposition

A *block-cyclic* decomposition combines the two strategies; the elements are divided into many adjacent groups, usually an integer multiple of the number of processors available. Each group of elements is assigned to a processor cyclically. Figure 50 (d) shows an array whose columns have been allocated to two processors in a block-cyclic fashion and Figure 51 (c) gives the code to implement the mapping on MX-s.

Block-cyclic is a compromise between locality and load balancing.

7.3 Dynamic Iteration Scheduling

The two advantages to static decompositions are that they impose no dynamic scheduling overhead and they work well when iterations have roughly equal or easily predictable execution times. This is often the case in scientific code, which may perform some operation on each element in an array. When loop iterations vary in cost, the variation may be simple to compute from the induction expressions. Figure 53 shows a pair of nested loops where the inner depends on the outer's induction variable. This yields a triangularly shaped iteration space.

Static decomposition is less effective when execution times are irregular, because the load balance degrades. Irregularity is typically introduced by computed loop bounds and condition-

```
do all i=1, n
  a[i] = a[i] + c
end do all
```

(a) original loop

```
call fork(P)
do i = n/P*Pid+1, min(n/P*(Pid+1), n)
  a[i] = a[i] + c
end do
call join()
```

(b) block-scheduled loop (block size `n/P`)

```
call fork(P)
do i = Pid+1, n, P
  a[i] = a[i] + c
end do
call join()
```

(c) cyclic-scheduled loop

```
call fork(P)
do i = 8*Pid+1, min(8*(Pid+1), n)
  a[i] = a[i] + c
end do
call join()
```

(d) block-cyclic-scheduled loop (block size 8)

Figure 51: Scheduling a Parallel Loop on MX-s. `P` is the total number of processors, `Pid` is the local processor number.

```
if (my_pid() = 0) then
  broadcast(c,4)
else
  receive(c,4)
endif
do i = 1,n/numprocs()
  a[i] = a[i] + c
end do
```

Figure 52: Block-Scheduling a parallel loop on MX-d.

```
do i = 1, n
  do j = 1,i
    total = total + a[i,j]
  end do
end do
```

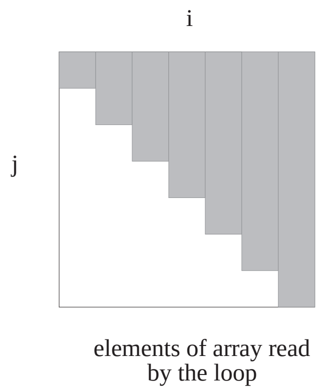


Figure 53: Triangular Iteration Space

als. A loop whose execution behavior cannot easily be predicted is shown in Figure 54.

Dynamic iteration scheduling can handle loops whose iterations have varying execution times. These strategies track the behavior of the loop; each scheduling decision allocates a group of iterations to a processor. The different algorithms differ in how they compute the number of iterations to put in a given group.

Self-scheduling [Tang and Yew 1990] is the simplest strategy and uses the *first-available* rule; the iterations are treated as a collection of tasks to perform, and each processor chooses a new task whenever it finishes its current one. A task can be a single iteration, or it can be a collection of them to reduce overhead. Self-scheduling is an easily implemented strategy, but it does not handle widely varied execution times well. Guided self-scheduling [Polychronopoulos and Kuck 1987] and factoring [Hummel et al. 1992] vary the number of iterations assigned based on the number left to compute and the number of processors. TAPER [Lucco 1992] uses a more sophisticated model of machine and computation state, taking into consideration the execution behavior of previous iterations and the level of parallelism currently available from all the computations being performed in the machine.

7.4 General Dynamic Scheduling

The previous section covered the dynamic scheduling of loop iterations; this section examines algorithms that seek to dynamically schedule more general types of code.

One form of generality is a loop that does not have predictable data usage. The following code fragment is simple, but it can encode any possible data interconnection pattern among a group of processors containing elements of the array:

```
do i = 1,n
   c[i] = a[b[i]]
end do
```

A different form of generality is to examine and decompose general blocks of code, not necessarily containing or contained by a loop, into large chunks which are assigned to processors. The task is either:

- to begin with a finely decomposed program and bundle the small computations together in an effort to reduce overhead or
- to begin with a monolithic block of sequential code, determine which parts are independent, and break it into pieces that are efficiently schedulable while also exposing sufficient parallelism

We will not cover the subject of general dynamic scheduling in detail, but the next sections briefly introduce transformations that address various aspects of the problem.

7.4.1 Graph Partitioning

Dataflow languages [Ackerman 1982; McGraw 1985; Nikhil 1988] expose parallelism explicitly. A program is converted into a graph that represents its basic computations as nodes and the movement of data as arrows between nodes. Figure 55 shows a simple program and its representation as a dataflow graph.

One of the major difficulties with dataflow graphs is that they expose parallelism at the level of individual operations. As it is impractical to use software to schedule such a small amount of work, early projects focused on developing architectures that embed dataflow execution policies into hardware [Dennis 1980; Arvind et al. 1980; Arvind and Culler 1986]. Such machines have not proven to be successful commercially, so researchers began to develop tech-

```
do i = 1,n
   if (mask[i] = 1) then
     a[i] = expensive_function(i)
   endif
enddo
```

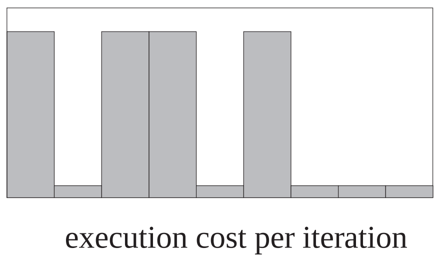


Figure 54: Irregular Execution Behavior

```
f(x,y)
    let a = sin(x)+cos(y)
         b = sin(y)+cos(x)
         c = a*b + pi/2
    in c*(a*b)
```

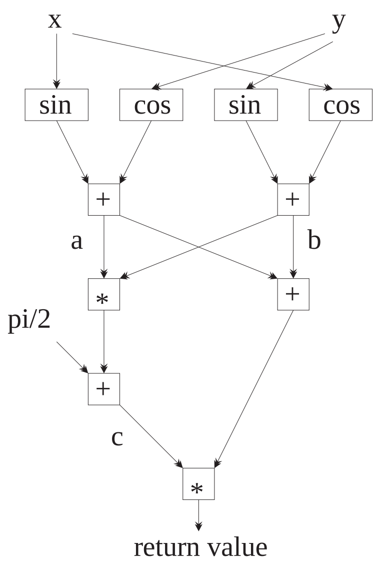


Figure 55: A Dataflow Language Fragment and its Dataflow Graph

niques for compiling dataflow languages on conventional architectures. The most common approach is to interpret the dataflow graph dynamically, executing a node representing a computation when all of its operands are available. To reduce scheduling overhead, the dataflow graph is generally transformed by gathering many simple computations into larger blocks that are executed atomically [Sarkar and Hennessey 1986a; Sarkar and Hennessey 1986b; Sarkar 1989b; Hudak and Goldberg 1985; Anderson and Hudak 1990; Mirchandaney et al. 1988].

7.4.2 *Multiple Call Parallelization*

Other work has been aimed at decomposing sequential programs to expose useful parallelism. Triolet et al. [1986] developed an approach that uses interprocedural dependence analysis to identify when the multiple invocations of a procedure in a loop can be executed in parallel. This is done by building descriptors of the regions of arrays accessed by the procedures.

7.4.3 *Split*

A more comprehensive approach to program decomposition summarizes the data usage behavior of a block of code in a *symbolic descriptor* [Graham et al. 1993]. The descriptor is used to identify independence between pieces of the program and to transform code to reveal additional opportunities for parallel execution. The process begins by dividing a sequential program into pieces and computing a descriptor for each piece. If the decomposition divided a procedure into two pieces, A and B, the next step is to determine whether they are independent. Even if they are not fully independent, it is often true that the compiler can expose partial parallelism and opportunities for pipelining by sub-dividing A further. The *split* transformation performs the sub-division, identifying the computations within A which are independent of B.

7.5 Fine-Grained Scheduling

Coarse-grained scheduling policies are usually embedded directly into a program or are carried out by a separate run-time system. Fine-grained scheduling between multiple function units within the chip is usually the responsibility of the hardware. However, by knowing the chip's scheduling policy, the compiler can target the code it outputs to take advantage of that policy.

In addition to applying the transformations discussed in Section 6, the compiler tries to exploit the scheduling policy in the target processor when it selects an instruction sequence. A widely used algorithm is called *list scheduling* [Adam et al. 1974]. Rau and Fisher [1993] present an excellent survey on the evolution of instruction-level parallelism in processors and the efforts of compiler designers to exploit it.

7.6 Explicitly Parallel Instruction Scheduling

With traditional processors, there is a clear distinction between low-level scheduling done via instruction selection and high-level scheduling between processors. This distinction is blurred in multi-threaded and Very Long Instruction Word (VLIW) architectures.

Unlike superscalar machines, which use hardware scheduling to discover and manage low level parallelism, multi-threaded and VLIW machines rely on the compiler to expose the parallel operations explicitly and sometimes to schedule them at compile time. Thus the task of generating object code incorporates high-level resource allocation and scheduling decisions that only occur between processors on a more conventional architecture.

7.6.1 *Multi-Threaded Architectures*

In a traditional architecture, there is a single thread of control executing at any moment, symbolized by the existence of a single program counter. On a multi-threaded architecture like the HEP [Smith 1978], Tera [Alverson et al. 1990], Monsoon [Papadopoulos and Culler 1990], and Fluent [Boothe and Ranade 1992], there are conceptually many different program counters. The processor allocates slices of execution time to the independent threads, using a variety of assignment policies. At the cost of extra processor state, multi-threaded machines can choose among many possible computations to support a large degree of instruction-level parallelism or to compensate for long memory access latency.

Multi-threaded architectures are generally not designed to support a traditional programming model. Machines like Monsoon are intended to be used for executing dataflow languages. The idea behind Fluent is to use multi-threading to support shared memory programming on many processors, hiding the latency of memory access by keeping more than one stream of control active. The HEP and Tera have their own programming models based on asynchronous variables and futures [Callahan and Smith 1990] respectively.

Because these alternative programming models are so different, their compilation strategies are not generally applicable to the optimization of conventional languages for conventional architectures.

7.6.2 VLIW

A VLIW processor [Colwell et al. 1988; Fisher et al. 1984; Rau et al. 1989; Flo 1979] exposes its multiple functional units explicitly; each machine instruction is very large (on the order of 512 bits) and controls many independent functional units. Typically the instruction is divided into 32 bit pieces, each of which is routed to one of the functional units.

Trace scheduling is the technique developed for compilation to a VLIW machine [Ellis 1986; Fisher et al. 1984; Fisher 1981]. When the target machine can only exploit very limited instruction-level parallelism, it is usually sufficient to constrain compiler analysis to a basic block. With a VLIW machine, basic block analysis will produce inefficient code because there is not enough parallelism available. Moving outside the basic block introduces complexity, however, because of branching. The multiple paths of control require the introduction of *speculative execution* — computing some results that may turn out to be unused.

The speculation can be handled dynamically by the hardware [Sohi and Vajapayem 1990], but that complicates the design significantly. An alternative is to have the compiler do it, hoisting code above branches based on the direction that the compiler expects the branch to take. Trace scheduling takes this approach; it identifies paths through the flow graph of a program that might be taken.

The compiler picks successive traces, assigning them to functional units. The choice starts with a *seed block*, the most frequently executed basic block that has not yet been scheduled. From the seed block, the compiler walks forwards and backwards in the flow graph to assemble a trace.

Superblock scheduling [Hwu et al. 1993] is an extension of trace scheduling that relies on program profile information to choose the traces.

8 TRANSFORMATION FRAMEWORKS

Given the many transformations that compiler writers have available, they face a daunting task in determining which ones to apply and in what order. There is no single best order of application; one transformation can permit or prevent a second from being applied, or it can change the effectiveness of subsequent changes. Current compilers generally use a combination of heuristics and a partially fixed order of applying transformations.

There are two basic approaches to this problem: unifying the transformations in a single mechanism, and applying search techniques to the transformation space. In fact there is often a degree of overlap between these two approaches.

8.1 Unified Transformation

A promising strategy is to encode both the characteristics of the code being transformed and the effect of each transformation; then the compiler can quickly search the space of possible sets of transformations to find an efficient solution.

One framework that is being actively investigated is based on unimodular matrix theory [Banerjee 1991; Wolf and Lam 1991]. It is applicable to any loop nest whose dependences can be described with a distance vector; a subset of the loops which require a direction vector can also be handled. The transformations that a unimodular matrix can describe are interchange, reversal, and skew.

The basic principle is to encode each transformation of the loop in a matrix, and apply it

```
do i = 2, 10
  do j = 1, 10
    a[i,j] = a[i-1,j] + a[i,j]
  end do
end do
```

Figure 56: Unimodular Transformation Example

to the dependence vectors of the loop. The effect on the dependence pattern of applying the transformation can be determined by multiplying the matrix and the vector. The form of the product vector reveals whether the transformation is legal. To model the effect of applying a sequence of transformations, the corresponding matrices are simply multiplied.

Figure 56 shows a loop nest that can be transformed with unimodular matrices. The distance vector that describes the loop is $D = (1, 0)$, representing the dependence of iteration i on $i-1$ in the outer loop.

Because of the dependence, it is not legal to reverse the outer loop of the nest. The reversal transformation is $R = \begin{bmatrix} -1 & 0 \\ 0 & 1 \end{bmatrix}$. The product $RD = P_1 = \begin{bmatrix} -1 \\ 0 \end{bmatrix}$. This demonstrates that the transformation is not legal, because P_1 is not *lexicographically positive*.

We can also test whether the two loops can be interchanged; the interchange transformation is $I = \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}$. Applying that to D yields $P_2 = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. In this case, the resulting vector is lexicographically positive showing that the transformation is legal.

Any loop nest whose dependences are all representable by a distance vector can be transformed into a canonical form called a *fully permutable loop nest*. In this form, any two loops in the nest can be interchanged without changing the loop semantics. Once in this canonical form, the compiler can decompose the loop into the granularity that matches the target architecture [Wolf and Lam 1991].

Sarkar and Thekkath [1992] describe a framework for transforming perfect loop nests which includes unimodular transformations, blocking, coalescing, and parallel loop execution. The transformations are encoded in an ordered sequence. Rules are provided for mapping the dependence vectors and loop bounds, and the transformation to the loop body is described by a template.

Pugh [1991] describes a more ambitious (and time-consuming) technique that can transform imperfectly nested loops and can do most of the transformations possible through a combination of statement reordering, interchange, fusion, skewing, reversal, distribution, and parallelization. It views the transformation problem as that of finding the best schedule for a set of operations in a loop nest. A method is given for generating and testing candidate schedules.

8.2 Searching the Transformation Space

Wang [1991] and Wang and Gannon [1989] describe a parallelization system which uses heuristic search techniques from artificial intelligence to find a program transformation sequence. The target machine is represented by a set of features that describe the type, size, and speed of the processors, memory, and interconnect. The heuristics are organized hierarchically. The main functions are description of parallelism in the program and in the machine; matching of program parallelism to machine parallelism; and control of restructuring.

9 COMPILER EVALUATION

Researchers are still trying to find a good way to evaluate the effectiveness of compilation. There is no generally agreed upon way to determine the best possible performance of a particular program on a particular machine, so it is difficult to determine how well a compiler is doing. Since some applications are better structured than others for a given architecture or a given compiler, measurements for a particular application or group of applications will not necessarily predict how well another application will fare.

Nevertheless, a wide variety of measurement studies do exist that seek to evaluate how applications behave, how well they are being compiled, and how well they could be compiled. We have divided these studies into several groups.

9.1 Benchmarks

Benchmarks have received by far the most attention since they measure delivered performance and the results are used to market machines. They were originally developed to measure machine speed, not compiler effectiveness. The Livermore Loops [McMahon 1986] is one of the early benchmark suites; it sought to compare the performance of supercomputers. The suite consists of a set of small loops based on the most time-consuming inner loops of scientific codes.

The SPEC benchmark suite [Dixit 1992] includes both scientific and general purpose applications intended to be representative of an engineering/scientific workload. The SPEC benchmarks are widely used as indicators of machine performance, but are essentially uniprocessor benchmarks.

As architectures have become more complex it has become obvious that the benchmarks measure the combined effectiveness of the compiler and the target machine. Thus two compilers that target the same machine or two versions of a compiler can be compared by using the SPEC ratings of the generated code.

For parallel machines, the contribution of the compiler is even more important, since the difference between naïve and optimized code can be many orders of magnitude. Parallel benchmarks include SPLASH (Stanford Parallel Applications for Shared Memory) [Singh et al. 1992], and the NASA Numerical Aerodynamic Simulation (NAS) benchmarks [Bailey et al. 1991].

The Perfect Club [Berry et al. 1989] is a benchmark suite of computationally intensive programs that is intended to help evaluate serial and parallel machines.

9.2 Code Characteristics

Other studies have focused on the applications themselves, examining their source code for programmer idioms or profiling the behavior of the compiled executable.

Shen et al. [1989] examined the subscript expressions that appeared in a set of Fortran applications and applied various types of dependency tests to these expressions. The results demonstrate how the various tests compare to one another, showing that one of the biggest problems was unknown variables. These variables were caused by procedure calls and by the use of an array element as an index value into another array. Coupled subscripts also caused problems for tests that examine a single array dimension at a time.

Knuth [1971] was an early and influential study of Fortran programs. He studied 440 programs comprising 250000 lines (punched cards). The most important effect of this study was to dramatize the fact that the majority of the execution time of a program is usually spent in a very small proportion of the code. Other interesting statistics are that 95% of all the **do** loops incremented their index variable by 1, and 40% of all **do** loops contained only one statement.

9.3 Compiler Effectiveness

As we mentioned above, researchers find it difficult to evaluate how well a compiler is doing. They have come up with four approaches to the problem:

- examine the compiler output by hand to evaluate its ability to transform code
- measure performance of one compiler against another
- compare the performance of executables compiled with full optimization against little or no optimization
- compare an application compiled for parallel execution against the sequential version running on one processor.

Nobayashi and Eoyang [1989] compare the performance of supercomputer compilers from Cray, Fujitsu, Alliant, Ardent, and NEC. The compilers were applied to various loops from the Livermore and Argonne test suites which required restructuring before they could be computed with vector operations.

Relatively few studies have been performed to test the effectiveness of real compilers in parallelizing real programs. However, the results of those that have are not encouraging.

One study of four Perfect benchmark programs compiled on the Alliant FX/8 produced speedups between 0.9 (that is, a slowdown) and 2.36 out of a potential 32; when the applications were tuned by hand, the speedups ranged from 5.1 to 13.2 [Eigenmann et al. 1991]. In another study of 12 Perfect benchmarks compiled with the KAP compiler for a simulated machine with unlimited parallelism, 7 applications had speedups of 1.4 or less, two applications had speedups of 2–4, and the rest were sped up 10.3, 66, and 77; all but three of these applications could have been sped up by a factor of 10 or more [Petersen and Padua 1990].

Lee at al. [1985] study the ability of the Parafrase compiler to parallelize a mixture of small programs written in Fortran. Before compilation, **while** loops were converted to **do** loops, and code for handling error conditions was removed. With 32 processors available, 4 out of 15 applications achieved 30% efficiency and 2 achieved 10% efficiency; the other 9 out of 15 achieved less than 10% efficiency. Out of 89 loops, 59 were parallelized, most of them loops that initialized array data structures. Some coding idioms that are amenable to improved analysis were identified.

9.4 Maximum Available Parallelism

In order to evaluate the potential gain from instruction level parallelism, researchers have engaged in a number of studies to measure an upper bound on how much parallelism is available assuming an unlimited number of functional units. Some of these studies are discussed and evaluated in detail in [Rau and Fisher 1993].

Early studies [Tjaden and Flynn 1970] were pessimistic in their findings, measuring a maximum level of parallelism on the order of two or three — a result that was called the *Flynn bottleneck*. The main reason for the low numbers was that these studies did not look beyond basic blocks.

Parallelism can be exploited across basic block boundaries, however, on machines that use *speculative execution*. Instead of waiting until the outcome of a conditional branch is known, these architectures begin executing the instructions at either or both potential branch targets; when the conditional is evaluated, any computations that are rendered invalid must be discarded.

When the basic block restriction is relaxed, there is much more parallelism available. [Riseman and Foster 1972] assumed replicated hardware to support speculative execution, while [Nicolau and Fisher 1984] sought to gain similar benefits from code transformation. Other studies investigated the effects of branch prediction [Butler et al. 1991] and other strategies for managing control flow [Wall 1991; Lam and Wilson 1992]

Larus studies the loop-level parallelism in a mixture of numeric and symbolic programs, and presents detailed measurements of loop-carried dependences [Larus 1993].

The updated studies may still understate the level of available parallelism, because they begin with an instruction trace from a compiled program. A compiler that used code transformations more aggressively could improve the results.

APPENDICES

A MACHINE MODELS

A.1 Superscalar DLX

A superscalar processor has multiple functional units and can issue more than one instruction per clock cycle. Current examples of superscalar machines are the DEC Alpha [Sites 1992], HP PA-RISC [Hew 1992], IBM RS/6000 [Oehler and Blasgen 1991], and Intel Pentium [Alpert and Avnon 1993].

S-DLX is a simplified superscalar RISC architecture. It has four independent functional units for integer, load/store, branch, and floating point operations. In every cycle, the next two instructions are considered for execution.

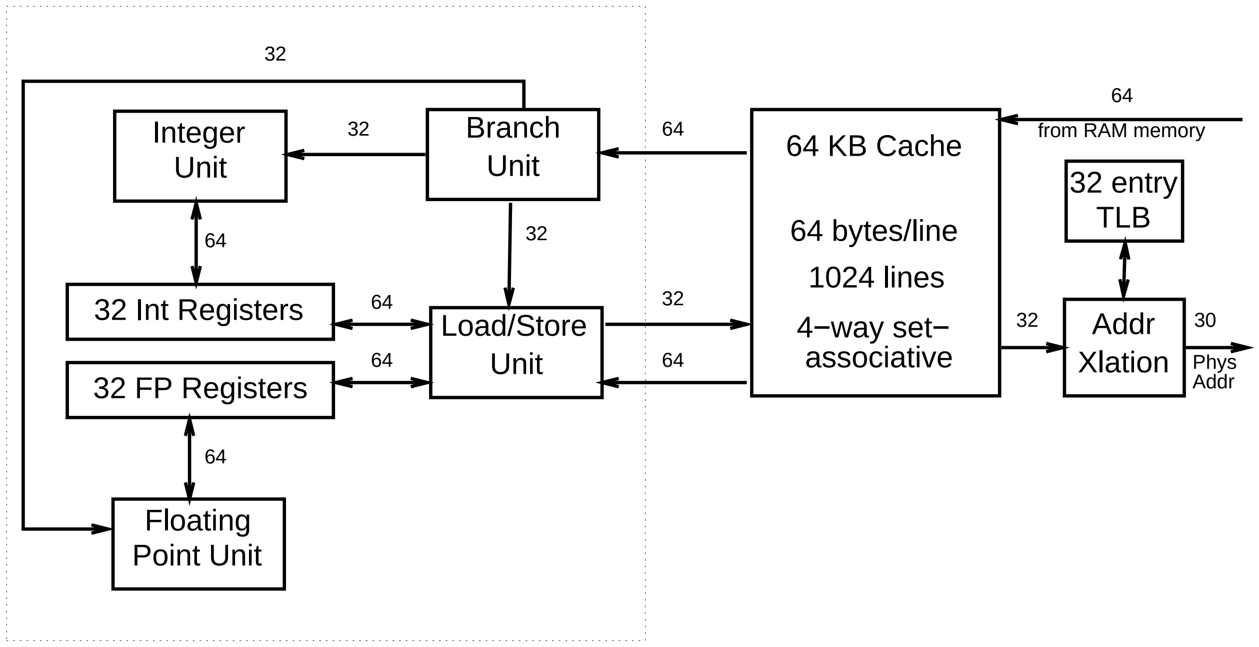


Figure 57: S-DLX Functional Diagram

Example Instr.	Name	Meaning	Similar instructions
`LW R1, 30(R2)`	Load word	`R1`←`Memory[30+R2]`	Load float (`LF`)
`SW 500(R4), R3`	Store word	`Memory[500+R4]`←`R3`	Store float (`SF`)
`LI R1, #666`	Load immediate	`R1`←666	
`LUI R1, #666`	Load upper immediate	$\texttt{R1}_{16..31}$ ←666	
`MOV R1, R2`	Move register	`R1`←`R2`	
`ADD R1, R2, R3`	Add	`R1`←`R2`+`R3`	Subtract (`SUB`)
`MULT R1, R2, R3`	Multiply	`R1`←`R2`×`R3`	Divide (`DIV`)
`ADDI R1, R2, #3`	Add immediate	`R1`←`R2`+3	`SUBI, MULTI, DIVI`
`SLL R1, R2, R3`	Shift left logical	`R1`←`R2`≪`R3`	Shift right logical (`SRL`)
`SLLI R1, R2, #3`	Shift left immediate	`R1`←`R2`≪ 3	`SRLI`
`SLT R1, R2, R3`	Set less than	if (`R2`<`R3`) `R1`←1 else `R1`←`0`	`SEQ, SNE, SLE, SGE, SGT` and immediate forms
`J label`	Jump	`PC`←`label`	
`JR R3`	Jump register	`PC`←`R3`	
`JAL label`	Jump and link	`R31`←`PC`+4; `PC`←`label`	
`JALR R2`	Jump and link register	`R31`←`PC`+4; `PC`←`R2`	
`BEQZ R4, label`	Branch if equal zero	if (`R4`=0) `PC`←`label`	Branch if not equal zero (`BNEZ`)
`BFPT label`	Branch if floating point true	if (`FPCR`) `PC`←`label`	Branch if floating point false (`BFPF`)
`ADDF F1, F2, F3`	Add float	`F1`←`F2`+`F3`	Subtract float (`SUBF`)
`MULTF F1,F2, F3`	Multiply float	`F1`←`F2`×`F3`	Divide float (`DIVF`)
`MAF F1,F2, F3`	Multiply-Add float	`F1`←`F1`+(`F2`×`F3`)	
`EQF F1, F2`	Test equal float	if (`F1`=`F2`) `FPCR` ←1 else `FPCR` ←0	`LTF, GTF, LEF, GEF, NEF`

Table 2: The S-DLX instruction set.

If they are for different functional units, and there are no dependences between the instructions, they are both initiated. Otherwise, the second instruction is deferred until the next cycle. S-DLX does not reorder the instructions.

Most operations complete in a single cycle. When an operation takes more than one cycle, subsequent instructions that use results from multi-cycle instructions are *stalled* until the result is available. Because there is no instruction reordering, when an instruction is stalled no instructions are issued to any of the functional units.

S-DLX has a 32-bit word, 32 general purpose registers (GPRs, denoted by `Rn`), and 32 floating point registers (FPRs, denoted by `Fn`). The value of `R0` is always 0. The FPRs can be used as double-precision (64-bit) register pairs. For the sake of simplicity we have not included the double-precision floating point instructions.

Memory is byte-addressable with a 32 bit virtual address. All memory references are made by load and store instructions between memory and the registers. Data is cached in a 64 kilobyte 4-way set-associative write-back cache composed of 1024 64-byte cache lines. Figure 57 is a block diagram of the architecture and its primary datapaths.

Table 2 describes the instruction set. All instructions are 32 bits and must be word-aligned. The immediate operand field is 16 bits. The address for the load and store instructions is computed by adding the 16-bit immediate operand to the register. To create a full 32-bit constant, the low 16 bits must first be set with a load immediate (`LI`) instruction which clears the high 16 bits; then the high 16 bits must be set with a load upper immediate (`LUI`) instruction.

The program counter is the special register `PC`. Jumps and branches are relative to `PC+4`; jumps have a 26-bit signed offset and branches have a 16-bit signed offset. Integer branches test the GPRs for zero; floating-point branches test a special floating point condition register (`FPCR`).

There is no *branch delay slot*. Because the number of instructions executed per cycle varies on a superscalar machine, it does not make sense to have a fixed number of delay slots. The number of delay slots is also heavily dependent on the pipeline depth, which may vary from one chip generation to the next.

Operation	Start-Up Time
Vector Add	6
Vector Multiply	7
Vector Divide	20
Vector Load	12

Table 3: Start-up Times in Cycles on DLX-V.

Instead, static branch prediction is used: forward branches are always predicted as not taken, and backward branches are predicted as taken. If the prediction is wrong, there is a one cycle delay.

Integer multiplication takes two cycles and all floating point operations take four cycles, except when the result is used by a store operation, in which case they take three cycles. A load that hits in the cache takes two cycles; if it misses in the cache it takes 16 cycles. All other instructions take one cycle.

When a load or store is followed by an integer instruction that modifies the address register, the instructions may be issued in the same cycle. If a floating point store is followed by an operation that modifies the register being stored, the instructions may also be issued in the same cycle.

A.2 Vector DLX

For vectorizing transformations, we will use a vector version of DLX, extended to include vector support. This new architecture, DLX-V, has eight vector registers each of which holds a vector consisting of up to 64 floating point numbers. The vector functional units perform all their operations on data in the vector and scalar registers.

We will only be discussing the functional units that perform floating point addition, multiplication, and division, though vector machines typically have units to perform integer and logical operations as well. DLX-V only issues one scalar or one vector instruction per cycle, but non-

Example Instr.	Name	Meaning	Similar
`LV V1, R1`	Load vector register	`V1`←`VLR` words at `M[R1]`	
`LVWS V1, (R1,R2)`	Load vector with stride	`V1`←every `R2`th word for `VLR` words at `M[R1]`	
`SV V1, R1`	Store vector register	`M[R1]`←`VLR` words from `V1`	
`SVWS V1, (R1,R2)`	Store vector with stride	`M[R1]`←`VLR` words from `V1` with stride `R2`	
`ADDV V1,V2,V3`	Vector-vector addition	`V1[1..VLR]`←`V2[1..VLR]+V3[1..VLR]`	`MULTV`
`ADDSV V1,F1,V2`	Vector-scalar addition	`V1[1..VLR]`←`F1+V2[1..VLR]`	`MULTSV`
`SUBV V1,V2,V3`	Vector-vector subtraction	`V1[1..VLR]`←`V2[1..VLR]-V3[1..VLR]`	`DIVV`
`SUBVS V1,V2,F1`	Vector-scalar subtraction	`V1[1..VLR]`←`V2[1..VLR]-F1`	`DIVVS`
`SUBSV V1,F1,V2`	Scalar-vector subtraction	`V1[1..VLR]`←`F1-V2[1..VLR]`	`DIVSV`
`SVLR R1`	Set vector length register	`VLR`←`R1`	

Table 4: The DLX-V vector instructions.

conflicting scalar and vector instructions can overlap each other.

A special register, the *vector length register* (`VLR`), controls the number of quantities that are loaded, operated upon, or stored in any vector instruction. The `VLR` is normally set to 64, except when handling the last few iterations of a loop.

The vector operations are described in Table 4. They include arithmetic operations on vector registers and load/store operations between vector registers and memory. Vector operations take place either between two vector registers or between a vector register and a scalar register. In the latter case, the scalar value is extended across the entire vector. All vector computations have a vector register as the destination.

The speed of a vector operation depends on the depth of the pipeline in its implementation. The first result appears after some number of cycles (called the *start-up time*). After the pipeline is full, one result is computed per clock cycle. In the meantime, the processor can continue to execute other instructions. Table 3 gives the startup times for the vector operations in DLX-V. These times should not be compared directly to the times in cycles for operations on S-DLX because vector machines typically have higher clock speeds than microprocessors, although this gap is closing.

A large factor in the speed of vector architectures is their memory system. DLX-V has 8 memory banks. After the load latency data is supplied at the rate of one word per clock cycle, provided that the stride with which the data is accessed does not cause bank conflicts (see Section 6.3.2).

Current examples of vector machines are the Cray C-90 [Oed 1992] and IBM ES 9000 Model 900 VF [Gibson and Rao 1992].

A.3 Multiprocessors

When multiple processors are employed to execute a program, many additional issues arise. The most obvious is how much of the underlying machine architecture to expose to the programmer. At one extreme is explicit use of hardware-supported operations by programming with locks, fork/join primitives, barrier synchronizations, and message send and receive. These operations are typically provided as system calls explicitly invoked by the programmer. System call semantics are usually not defined by the language, making it difficult for the compiler to automatically optimize them. We therefore do not discuss the transformation of such programs.

High-level languages for large-scale parallel processing provide primitives for expressing parallelism in one of two ways: *control parallel* or *data parallel*. Fortran-90 array section expressions are examples of explicitly data parallel operations. APL [Iverson 1962] also contains a wide variety of data parallel array operators. Examples of control parallel operations are **cobegin/coend** blocks and **doacross** loops [Cytron 1986].

Operation	Cycles
`fork(n)`	450
`join()`	90
`barrier()`	70
`SWAPW R1, 16(R8)`	42

Table 5: Minimum Time in Cycles for Parallel Operations on MX-s.

A.4 Shared-Memory DLX Multiprocessor

MX-s is our prototypical shared-memory parallel architecture. It consists of 16 processors and 8 banks of memory, each holding 64 megabytes. The processors and memories are connected together by a bus, as shown in Figure 58. Each processor has an intelligent cache controller that monitors the bus (a *snoopy cache*). The caches are the same as on S-DLX, except that they contain 256KB of data. The bandwidth of the bus is 128 megabytes/second.

The processors share the memory units; a program running on a processor can access any memory element and the system ensures that the values are maintained consistently across the machine. Without caching, consistency is easy to maintain; every memory reference is handled by the appropriate memory unit. However, performance would be very poor because memory latencies are already too high on sequential machines to run well without a cache; having many processors share a common bus would make the problem much worse by increasing memory access latency and introducing bandwidth restrictions. The solution is to give each processor a cache that is smart enough to resolve reference conflicts.

A snoopy cache implements a sharing protocol that maintains consistency while still minimizing bus traffic. A processor that modifies a cache line invalidates all other copies and is said to *own* that cache line. There are a variety of cache coherency protocols [Stenström 1990; Eggers and Katz 1989], but the details are not relevant to this study. From the compiler writer's perspective, the key issue is the time it takes to make a memory reference. Table 59 summarizes the latency of each kind of memory reference in MX-s.

Table 5 shows the additional operations provided on MX-s to support parallel operation. `fork(n)` starts the current program running on n processors; because it must copy the stack of the forking processor to a private memory for each of the $n-1$ other processors, it is a very expensive operation. `join()` re-synchronizes with the previous `fork`, and makes the processor available for other `fork` operations (except for the original forking processor, which proceeds serially).

`barrier()` performs a barrier synchronization – a *barrier* is a synchronization point in the program where each processor waits until all processors have arrived at that point. The `SWAPW` instruction exchanges the register with the specified address in memory; it is used to implement locks.

A.5 Distributed-Memory DLX Multiprocessor

MX-d is our distributed memory model. The machine consists of 64 S-DLX processors (indexed 0 through 63) connected in an 8×8 mesh, as shown in Figure 60. The network bandwidth of each link in the mesh is 5 MBytes per second. Each processor has an associated network processor that manages communication; the network processor has its own pool of memory and can communicate without involving the CPU. Having a separate processor manage the network allows applications to send a message asynchronously and continue executing while the message is sent. Messages that pass through a processor en route to some other destination in the mesh are handled by the network processor without interrupting the CPU.

The latency of a message transfer is ten microseconds plus 2 microseconds per mesh link traversed plus 1 microsecond per 5 bytes of message. Communication is supported by a message library that provides the following calls:

- `my_pid()` – returns the current processor's index.

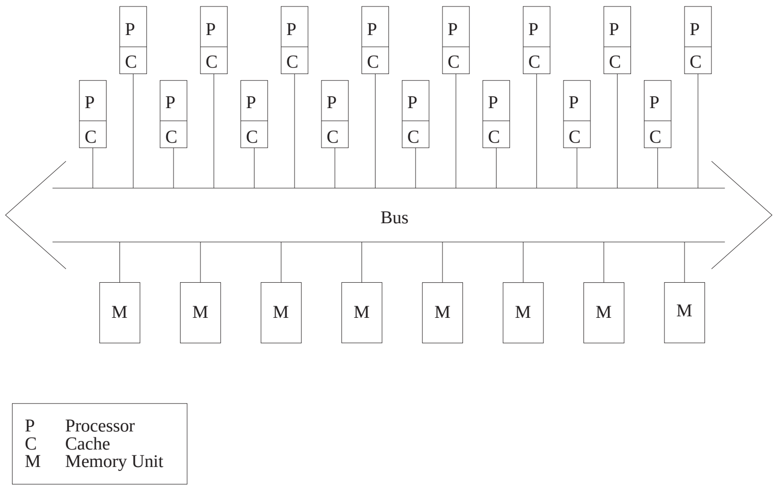


Figure 58: MX-s Shared Memory Architecture

Cycles	Type of Memory Reference
2	Read value available in local cache
16	Read value owned by other processor
20	Read value nobody owns
1	Write value owned locally
18	Write value owned by other processor
22	Write value nobody owns

Figure 59: Memory Reference Latency in MX-s

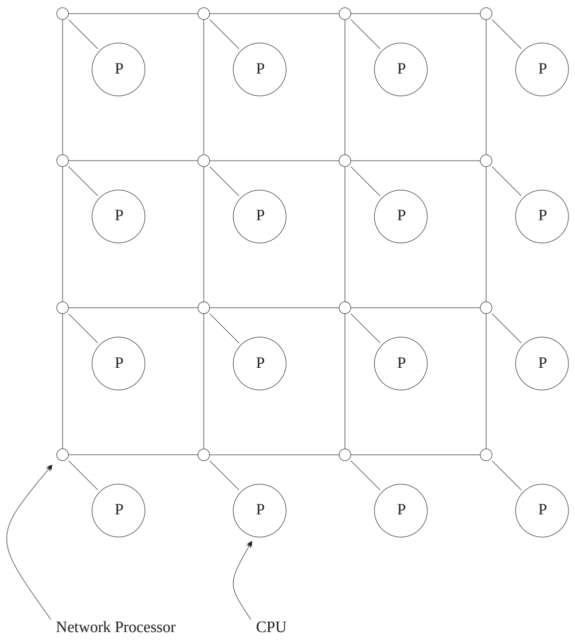


Figure 60: MX-d Architecture

- `num_procs()` – returns the number of processors in the machine.
- `send(buffer,nbytes,target)` – send `nbytes` from `buffer` to the processor `target`. If the message fits in the network processor's memory, the call returns after the copy (1 microsecond/5 bytes). If not, the call blocks until the message is sent. Note that this raises the potential for deadlock, but we will not concern ourselves with that in this simplified model.
- `broadcast(buffer,nbytes)` – send the message to every other processor in the machine. The communication library uses a fast broadcasting algorithm, so the maximum latency is roughly twice that of sending a message to the furthest edge of the mesh.
- `receive(buffer,nbytes)` – wait for a message to arrive; when it does, up to `nbytes` of it will be put in the buffer. The call returns the total number of bytes in the incoming message; if that value is greater than `nbytes`, `receive` guarantees that subsequent calls will return the rest of that message first.

B PROGRAM REPRESENTATION

There are a wide variety of representations that have been developed by compiler writers to simplify analysis. The most basic is the *control flow graph* (CFG) [Allen and Cocke 1976]. The CFG is a directed graph containing one node for each basic block in the program, plus two distinguished nodes called `Entry` and `Exit`. Each node has an edge to every node to which it can transfer control. The `Entry` node has an edge to every basic block that represents an entry point of the code; there is an edge to `Exit` from any basic block that can cause an exit. Figure 61 shows a procedure (a) and its control flow graph (b).

A second representation which is often used when managing transformations is the *program dependence graph* (PDG) [Ferrante et al. 1987]. A number of studies have shown how it can handle various optimizing transformations [Ottenstein and Ottenstein 1984; Baxter and Bauer III 1989; Selke 1989; Allen et al. 1988b].

A PDG is a directed graph in which the nodes represent statements and expressions and the edges represent data values passed between nodes or control conditions that determine whether the node will be evaluated. The root of a PDG is a distinguished node called `Entry`. There are two different kinds of edges to represent data and control dependences; if an edge is describing a control dependency, it will be labeled `true` or `false`. When there is a labeled edge from node A to node B, B is executed only if the evaluation of the predicate at node A matches the value of the label.

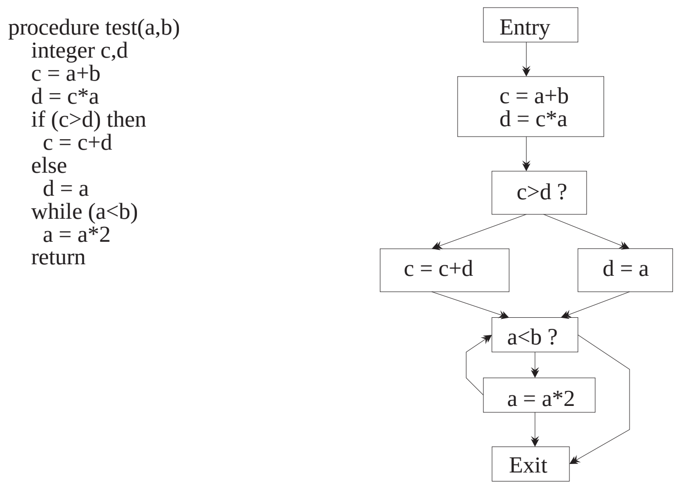
The procedure for building a PDG is described in detail by Ferrante et al. [1987]. In order to analyze the control dependences in a CFG, the compiler must determine the dominance relationship between pairs of nodes. Node X is said to *dominate* Y if X is always executed before Y. More formally, every path from the `Entry` node to Y passes through X. A similar notion is that of a *post-dominator*; B post-dominates A if and only if every path in the CFG from A to the `Exit` node goes through B.

Note that because it models data dependences instead of control dependences, there is no `Exit` node in the PDG.

Unlike control flow graphs, an edge between two nodes does not necessarily indicate a direct transfer of control between their corresponding code fragments. An edge from C to D means that D is control-dependent on C; D will only execute if C does.

In addition, the PDG supports summarization through the use of *region* nodes; if a set of nodes have the same control and data conditions on their execution, they can be grouped together and represented by a single node.

A third representation that is under active investigation is *static-single assignment form* (SSA) [Cytron et al. 1991]. SSA takes its inspiration from functional languages, which bind a value to a variable once and do not permit the value to be changed. SSA transforms an impera-



```
procedure test(a,b)
    integer c,d
    c = a+b
    d = c*a
    if (c>d) then
      c = c+d
    else
      d = a
    while (a<b)
      a = a*2
    return
```

Figure 61: Procedure and Corresponding CFG

tive program by introducing new variables; each variable is assigned a value once. When a particular name appears in two places, it refers to the same value. In functional languages, this principle is known as *referential transparency* [Abelson and Sussman 1985].

SSA is constructed by adding new variables and *merge* nodes. A merge node (also called a ϕ-function) is introduced when multiple control flow paths come together, yielding many possible values for a given variable. For example, Figure 62 shows a small fragment of code and the SSA conversion for it. On entry to the function `simple`, `a` has the value a_0. Each iteration of the loop changes the value of `a`, so at loop entry there are two possible values for `a`; control may be entering for the first time, or it may be in the middle of iteration. We create a name for the value of `a` at the top of the loop, calling it a_1. The assignment statement creates another version, which we call a_2. If the loop has completed execution, control goes to the node representing the last statement in the subroutine. It generates the final version of `a`, a_4, using the value a_3, because the loop will have been evaluated at least once.

The advantage to SSA is that it gives an explicit name to every value that a variable takes on. This makes comparison and value propagation much simpler; recasting optimizations traditionally done on flow graphs to use SSA is an ongoing area of interest.

C PROGRAM ANALYSIS

Compilers perform a variety of other types of analysis, aside from the dependence information discussed in Section5. This appendix discusses some of the other popular strategies for understanding the behavior of a program.

C.1 Dataflow Analysis

Dataflow analysis [Muchnick and Jones 1981] was one of the first strategies for analyzing program behavior. It is still widely used in modern compilers, although some of the information it is traditionally used to compute is subsumed by conversion into program representations like SSA and the PDG.

Dataflow analysis is usually performed on a control flow graph (CFG); it attempts to track the flow of data through the program's variables and to characterize the values of variables at various points of execution.

There are two primary strategies for performing dataflow analysis. The first is to develop a set of equations that are applied to basic blocks, yielding a list of variables that match some desired criteria. For example, suppose we wish to compute the set of values that are available at the exit of a given basic block B. Conventionally, this set is named *out*.

```
subroutine simple(a,b)
  a = a + 1
  do i = 1,10
    a = a + b
  end do
  a = a + 1
  return
```

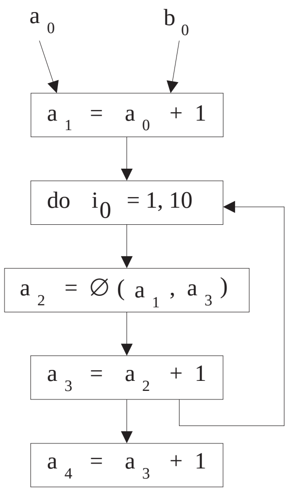


Figure 62: Simple Example of SSA

```
procedure simple(a,b,c,d)
if (a < 3) then
  a = a + 1
  b = b + a
else
  d = b
  b = c
  c = a
endif
```

Figure 63: May vs. Must: `simple` *may* modify `a,b,c,d` but it *must* only modify `b`.

The equation to compute $out(B)$ is

$$out(B) = gen(B) \cup (in(B) - kill(B))$$

where $gen(B)$ is the set of values generated inside the block, $in(B)$ is the set of values that *reach* the block from some other block, and $kill(B)$ is the set of values that were overwritten in the block (by an assignment statement, for example).

The second major strategy for doing dataflow analysis is based on *chains*. The idea is to identify where variables are assigned a value (called a *def*) and where the value is used (a *ref*). By connecting a particular def with every corresponding ref, the compiler can perform a wide variety of analysis including copy propagation, identification of loop invariance, and code motion. Depending on the analysis being performed, sometimes the compiler will connect uses to defs and sometimes the reverse.

Dataflow analysis is often conservative, because the compiler may not know what will happen at run time — whether a given conditional will be true or false, for example. Hence there is a distinction between *may* and *must* information. The set of variables modified by a procedure, for example, *may* include every variable that is the target of an assignment statement. However, if the goal is to find the set of variables that *must* be modified, the compiler needs to consider the effect of conditionals. In the procedure in Figure 63, the set of possibly modified variables is $\{a, b, c, d\}$, but the set of variables that must be modified is just $\{b\}$.

The may/must distinction can also be understood in terms of flow; may (or *flow-insensitive*) information occurs on at least one path through the CFG. Must (*flow-sensitive*) information is more expensive to compute, because the compiler must check to make sure that some condition holds for every path to or from a given node.

A straightforward approach to dataflow analysis is to apply the equations iteratively until reaching a fixed point [Kildall 1973; Hecht 1977; Ullman 1973; Aho et al. 1986]. This strategy can become quite expensive to compute, both in execution time and in memory required, so compilers use a variety of other strategies.

An early form of summarization was used by Allen and Cocke [1976]; it considered a loop in isolation and then moved the analysis outwards, collapsing the loop into a single node. For the purposes of the analysis, a loop is considered to be a strongly connected region.

Allen and Cocke then developed a simpler strategy for summarization called *interval analysis* [Allen and Cocke 1976], which works on *reducible* flow graphs. A graph is reducible if its edges can be partitioned into two disjoint sets *forward* and *back*, such that

- The *forward* edges make up an acyclic graph, in which every node can be reached from the `Entry` node.

```
subroutine irreduce(a,b)
    if (a<b) goto 2
1:  a = a + 1
2:  if (a < 0) goto 1
    return
```

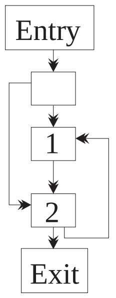


Figure 64: Procedure With an Irreducible Flow Graph

- For every *back* edge from node A to node B, B dominates A

Programs written in high level languages with structured constructs only generate reducible graphs unless the programmer uses goto statements. Figure 64 shows a fragment of code and its irreducible control flow graph. The code builds a loop with goto statements and violates structured programming style by having a jump into the middle of the loop – one of the simplest ways to generate an irreducible flow graph.

In practice, programs very rarely have irreducible flow graphs [Knuth 1971]. When such graphs do appear, they can always be converted to reducible graphs via *node splitting* [Cocke and Schwartz 1970]. Node splitting replicates nodes in the irreducible graph as necessary to yield a reducible one.

The idea behind interval analysis is to divide the graph into sets of nodes, called intervals, such that:

- the set is connected
- each set has a distinguished node h which dominates the nodes in the set
- all cycles within the set contain h

Once the graph has been divided into intervals, a data flow algorithm is performed in two passes. The first considers each of the intervals in isolation; the second computes the value for the entire graph using the summary information for each interval.

A variety of other optimizations have been developed as well, including path compression [Graham and Wegman 1976; Tarjan 1979; Ullman 1973].

C.2 Other Analysis Techniques

During analysis, most compilers also use a set of more specific techniques. We discuss several of these briefly.

C.2.1 Value Numbering

A compiler frequently needs to find identical sub-trees in a DAG, perhaps to avoid redundant computation by eliminating all but one of them. Value numbering [Reif and Lewis 1986; Alpern et al. 1988] is essentially a form of hashing which assigns a unique number to each quantity known at compile-time to be identical. Value numbering was originally developed by Cocke and Schwartz [1970].

Suppose that the computation involves a tree where each node has some value; assign a unique number to each value that can appear. Then compute the value of a sub-tree by taking some function of the value of each sub-tree and the value of the root node. The function could be addition, modulo some maximum value, or it might be sensitive to the ordering among the children. Enter each sub-tree into a hash table, using its value as the key. When two sub-trees collide, use a detailed comparison to determine whether they are truly identical.

C.2.2 Memory Usage Summarization

Dependency information is very useful for determining whether transformations are legal, but it is a low-level strategy that does not work well for summarizing the behavior of code. Summarization is often useful for high-level and interprocedural analysis; the usual goal is to describe the memory that is read or written by a code fragment. The key problem is to capture the behavior of arrays in a form that is easily manipulable. Researchers have proposed a variety of notations for array summaries, including Regular Sections [Callahan 1987], Data Access Descriptors [Balasundaram 1990], and atoms [Li and Yew 1988].

A somewhat related set of techniques try to describe the side-effects of a given piece of code so it can be scheduled without causing conflicts. Jade [Rinard et al. 1993] uses such a strategy for parallelism and what Gifford and Lucassen call *fluent* languages [Gifford and Lucassen 1986] use it to mix functional and imperative programming safely.

C.2.3 Feedback From Profiling

During optimization, a compiler is often forced to estimate the cost of various computations. A broad variety of models and heuristics are used to predict program behavior. An estimate of the number of iterations a loop is likely to execute is useful in determining scheduling granularity [Sarkar 1989a]. The direction a branch is likely to go [Bandyopadhyay et al. 1990; McFarling and Hennessy 1986] allows optimization of delayed branches. Global register allocation [Wall 1986] also depends on execution frequency information.

For high performance computing, these estimates can be very important in optimizing code. A number of studies have investigated the idea of profiling the program and feeding that information back into the optimizer [Sarkar and Hennessey 1986b; Hwu et al. 1993; Fisher and Freudenberger 1992; Fisher et al. 1984].

C.2.4 Alias Analysis

A crucial part of dataflow optimizations is computing the *kill* sets associated with each basic block. To do this, it must be possible to decide which variables (or parts of variables) can be written by each assignment. As long as there is a one-to-one correspondence between names and memory locations, this is relatively straightforward. Unfortunately, this is not the case for many languages.

In Fortran, aliasing can be created by parameter passing, and **equivalence** and **common** statements. In languages with pointers, there may be no information about a write through a pointer except the type of the object, and all *defs* of that type will need to be killed. The situation in C is even worse, since casting and unions allow a write through a pointer to write to any location in the address space. In the general case, a write through a pointer in C must be considered to kill *all* defs. This has a substantial impact on optimization.

Alias analysis uses a dataflow algorithm to propagate information about which variables may be aliased to one another. The problem of unrestricted pointers in C is often handled by providing an option that instructs the compiler to assume that the pointers are being used in a type-safe manner.

ACKNOWLEDGMENTS

We thank James Demmel, James Larus, John Hauser, and John Boyland for their valuable comments.

REFERENCES

ABELSON, H. AND SUSSMAN, G. J. 1985. *Structure and Interpretation of Computer Programs.* MIT Press, Cambridge, Massachusetts.

ABU-SUFAH, W. 1979. *Improving the Performance of Virtual Memory Computers.* PhD thesis, University of Illinois at Urbana-Champaign. Technical Report 78-945.

ABU-SUFAH, W., KUCK, D. J., AND LAWRIE, D. 1981. On the performance enhancement of paging systems through program analysis and transformations. *IEEE Trans. Comput., C-30,* 5 (May), 341–356.

ACKERMAN, W. B. 1982. Data flow languages. *Computer, 15,* 2 (Feb.), 15–25.

ADAM, T. L., CHANDY, K. M., AND DICKSON, J. R. 1974. A comparison of list schedules for parallel processing systems. *Commun. ACM, 17,* 12 (Dec.), 685–690.

AHO, A. V., JOHNSON, S. C., AND ULLMAN, J. D. 1977. Code generation for expressions with common subexpressions. *J. ACM, 24,* 1 (Jan.), 146–160.

Aho, A. V., Sethi, R., and Ullman, J. D. 1986. *Compilers : Principles, Techniques, and Tools*. Addison-Wesley, Reading, Massachusetts.

Aït-Kaci, H. 1991. *Warren's Abstract Machine: A Tutorial Reconstruction*. Logic Programming Series. MIT Press, Cambridge, Massachusetts.

Aiken, A. and Nicolau, A. 1988a. Optimal loop parallelization. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (Atlanta, Georgia, June). *SIGPLAN Notices*, *23*, 7, 308–317.

Aiken, A. and Nicolau, A. 1988b. Perfect pipelining: A new loop parallelization technique. In Ganzinger, H., Ed., *Proceedings of the Second European Symposium on Programming*, volume 300 of *Lecture Notes in Computer Science*, (Nancy, France, Mar.). Springer Verlag, Berlin, Germany, pp. 221–235. Also available as Cornell University Technical Report TR 87-873.

Allen, F. E. 1969. Program optimization. In *Annual Review in Automatic Programming 5*, volume 13 of *International Tracts in Computer Science and Technology and Their Application*, pp. 239–307. Pergammon Press, Oxford, England.

Allen, F. E., Burke, M., Charles, P., Cytron, R., and Ferrante, J. 1988a. An overview of the PTRAN analysis system for multiprocessing. *J. Parallel Distrib. Comput.*, *5*, 5 (Oct.), 617–640.

Allen, F. E., Burke, M., Cytron, R., Ferrante, J., Hsieh, W., and Sarkar, V. 1988b. A framework for determining useful parallelism. In *Proceedings of the ACM International Conference on Supercomputing*, (St. Malo, France, July). ACM Press, New York, pp. 207–215.

Allen, F. E. and Cocke, J. 1971. A catalogue of optimizing transformations. In Rustin, R., Ed., *Design and Optimization of Compilers*, pp. 1–30. Prentice-Hall, Englewood Cliffs, New Jersey.

Allen, F. E. and Cocke, J. 1976. A program data flow analysis procedure. *Commun. ACM*, *19*, 3 (Mar.), 137–146.

Allen, F. E., Cocke, J., and Kennedy, K. 1981. Reduction of operator strength. In Muchnick, S. S. and Jones, N. D., Eds., *Program Flow Analysis*, chapter 3, pp. 79–101. Prentice-Hall, Englewood Cliffs, New Jersey.

Allen, J. R. 1983. *Dependence Analysis for Subscripted Variables and its Application to Program Transformations*. PhD thesis, Dept. of Computer Science, Rice University.

Allen, J. R. and Kennedy, K. 1984. Automatic loop interchange. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, (Montreal, Canada, June). *SIGPLAN Notices*, *19*, 6, 233–246.

Allen, J. R., Kennedy, K., Porterfield, C., and Warren, J. 1983. Conversion of control dependence to data dependence. In *Proceedings of the Tenth Annual ACM Symposium on Principles of Programming Languages*, (Austin, Texas, Jan.). ACM Press, New York, pp. 177–189.

Allen, R. and Kennedy, K. 1987. Automatic translation of FORTRAN programs to vector form. *ACM Trans. Program. Lang. Syst.*, *9*, 4 (Oct.), 491–542.

Alpern, B., Wegman, M. N., and Zadeck, F. 1988. Detecting equality of values in programs. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, (San Diego, California, Jan.). ACM Press, New York, pp. 1–11.

Alpert, D. and Avnon, D. 1993. Architecture of the Pentium microprocessor. *IEEE Micro*, *13*, 3 (June), 11–21.

Alverson, R., Callahan, D., Cummings, D., Koblenz, B., Porterfield, A., and Smith, B. 1990. The Tera computer system. In *Proceedings of the ACM International Conference on Supercomputing*, (Amsterdam, The Netherlands, Sept.). *Computer Architecture News*, *18*, 3, 1–6.

Anderson, S. and Hudak, P. 1990. Compilation of Haskell array comprehensions for scientific computing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (White Plains, New York, June). *SIGPLAN Notices*, *25*, 6, 137–149.

Appel, A. W. 1992. *Compiling with Continuations*. Cambridge University Press, Cambridge, England.

Arden, B. W., Galler, B. A., and Graham, R. M. 1962. An algorithm for translating Boolean expressions. *J. ACM*, *9*, 2 (Apr.), 222–239.

Arvind and Culler, D. E. 1986. Dataflow architectures. In Traub, J. F., Grosz, B. J., Lampson, B. W., and Nilsson, N. J., Eds., *Annual Review* *of Computer Science*, volume 1, pp. 225–253. Annual Reviews, Palo Alto, California.

Arvind, Kathail, V., and Pingali, K. 1980. A dataflow architecture with tagged tokens. Technical Report TM-174, MIT Laboratory for Computer Science.

Bailey, D. H., Barszcz, E., Barton, J. T., Browning, D. S., Carter, R. L., Dagum, L., Fatoohi, R. A., Frederickson, P. O., Lasinski, T. A., Schreiber, R. S., Simon, H. D., Venkatakrishnan, V., and Weeratunga, S. K. 1991. The NAS parallel benchmarks. *Int. J. Supercomp. Appl.*, *5*, 3 (Fall), 63–73.

Balasundaram, V. 1990. A mechanism for keeping useful internal information in parallel programming tools: The data access descriptor. *J. Parallel Distrib. Comput.*, *9*, 2 (June), 154–170.

Balasundaram, V., Kennedy, K., Kremer, U., McKinley, K., and Subhlok, J. 1989. The ParaScope editor: An interactive parallel programming tool. In *Proceedings of Supercomputing '89*, (Reno, Nevada, Nov.). ACM Press, New York, pp. 540–550.

Ball, J. E. 1979. Predicting the effects of optimization on a procedure body. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, (Denver, Colorado, Aug.). *SIGPLAN Notices*, *14*, 8, 214–220.

Bandyopadhyay, S., Begwani, V. S., and Murray, R. B. 1990. Compiling for the CRISP microprocessor. In *Digest of Papers, Spring COMPCON 1987, Thirty-Second IEEE Computer Society International Conference*, (San Francisco, California, Feb.). IEEE Computer Society Press, Washington, D.C., pp. 96–100.

Banerjee, U. 1979. *Speedup of Ordinary Programs.* PhD thesis, University of Illinois at Urbana-Champaign, (Oct.). Technical Report 79-989.

Banerjee, U. 1988a. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, Massachusetts.

Banerjee, U. 1988b. An introduction to a formal theory of dependence analysis. *J. Supercomp.*, *2*, 2 (Oct.), 133–149.

Banerjee, U. 1991. Unimodular transformations of double loops. In Nicolau, A. et al., Eds., *Advances in Languages and Compilers for Parallel Processing*, Research Monographs in Parallel and Distributed Computing, chapter 10. MIT Press, Cambridge, Massachusetts.

Banerjee, U., Chen, S. C., Kuck, D. J., and Towle, R. A. 1979. Time and parallel processor bounds for FORTRAN-like loops. *IEEE Trans. Comput.*, *C-28*, 9 (Sept.), 660–670.

Baxter, W. and Bauer III, H. R. 1989. The program dependence graph and vectorization. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, (Austin, Texas, Jan.). ACM Press, New York, pp. 1–11.

Bernstein, R. 1986. Multiplication by integer constants. *Software – Practice and Experience*, *16*, 7 (July), 641–652.

Berry, M., Chen, D., Koss, P., Kuck, D., Lo, S., Pang, Y., Pointer, L., Roloff, R., Sameh, A., Clementi, E., Chin, S., Schneider, D., Fox, G., Messina, P., Walker, D., Hsiung, C., Schwarzmeier, J., Lue, K., Orszag, S., Seidl, F., Johnson, O., Goodrum, R., and Martin, J. 1989. The Perfect Club benchmarks: Effective performance evaluation of supercomputers. *Int. J. Supercomp. Appl.*, *3*, 3 (Fall), 5–40.

Blelloch, G. 1989. Scans as primitive parallel operations. *IEEE Trans. Comput.*, *C-38*, 11 (Nov.), 1526–1538.

Boothe, B. and Ranade, A. 1992. Improved Multithreading Techniques for Hiding Communication Latency in Multiprocessors. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, (Gold Coast, Australia, May). *Computer Architecture News*, *20*, 2, 214–223.

Bromley, M., Heller, S., McNerney, T., and Steele Jr., G. L. 1991. Fortran at ten gigaflops: The Connection Machine convolution compiler. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (Toronto, Ontario, June). *SIGPLAN Notices*, *26*, 6, 145–156.

Butler, M., Yeh, T., Patt, Y., Alsup, M., Scales, H., and Shebanow, M. 1991. Single instruction stream parallelism is greater than two. In *Proceedings of the 18th Annual International Symposium on Computer Architecture*, (Toronto, Ontario, May). *Computer Architecture News*, *19*, 3, 276–286.

Callahan, D. 1987. *A Global Approach to Detection of Parallelism.* PhD thesis, Rice University, (Apr.). Technical Report 87-50.

Callahan, D., Cooper, K., Kennedy, K., and Torczon, L. 1986. Interprocedural constant propagation. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, (Palo Alto, California, June). *SIGPLAN Notices*, *21*, 7, 152–161.

Callahan, D. and Smith, B. 1990. A future-based language for a general-purpose highly-parallel computer. In Gelernter, D., Nicolau, A., and Padua, D., Eds., *Languages and Compilers for Parallel Computing*, pp. 95–113. MIT Press, Cambridge, Massachusetts.

Chaitin, G. J. 1982. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, (Boston, Massachusetts, June). *SIGPLAN Notices*, *17*, 6, 98–105.

Chaitin, G. J., Auslander, M. A., Chandra, A. K., Cocke, J., Hopkins, M. E., and Markstein, P. W. 1981. Register allocation via coloring. *Computer Languages*, *6*, 1 (Jan.), 47–57.

Chambers, C. and Ungar, D. 1989. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (Portland, Oregon, June). *SIGPLAN Notices*, *24*, 7, 146–160.

Chen, S. C. and Kuck, D. J. 1975. Time and parallel processor bounds for linear recurrence systems. *IEEE Trans. Comput.*, *C-24*, 7 (July), 701–717.

Chow, F. C. 1988. Minimizing register usage penalty at procedure calls. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (Atlanta, Georgia, June). *SIGPLAN Notices*, *23*, 7, 85–94.

Chow, F. C. and Hennessy, J. L. 1990. The priority-based coloring approach to register allocation. *ACM Trans. Program. Lang. Syst.*, *12*, 4 (Oct.), 501–536.

Clark, C. D. and Peyton-Jones, S. L. 1985. Strictness analysis – a practical approach. In Jouannaud, J.-P., Ed., *Proceedings of the Conference on Functional Programming and Computer Architecture*, (Nancy, France, Sept.), pp. 35–49.

Cocke, J. 1970. Global common subexpression elimination. In *Proceedings of the ACM Symposium on Compiler Optimization*, (July). *SIGPLAN Notices*, *5*, 7, 20–24.

Cocke, J. and Markstein, P. 1980. Measurement of program improvement algorithms. In *Proceedings of the IFIP Congress*, (Tokyo, Japan, Oct.). North-Holland, Amsterdam, The Netherlands, pp. 221–228. Also available as IBM Research Division technical report RC 8111 (#35193), February 1980.

Cocke, J. and Schwartz, J. T. 1970. *Programming Languages and Their Compilers (Preliminary Notes).* Courant Institute of Mathematical Sciences, New York University, New York, New York, second revised edition.

Colwell, R. P., Nix, R. P., O'Donnel, J. J., Papworth, D. B., and Rodman, P. K. 1988. A VLIW architecture for a trace scheduling compiler. *IEEE Trans. Comput.*, *C-37*, 8 (Aug.), 967–979.

Cray Research, Inc. 1988. *CFT77 Reference Manual*, (Oct.). Publication SR-0018-C.

Cytron, R. 1986. Doacross: Beyond vectorization for multiprocessors. In Hwang, K. et al., Eds., *Proceedings of the International Conference on Parallel Processing*, (St. Charles, Illinois, Aug.). IEEE Computer Society Press, Washington, D.C., pp. 836–844.

Cytron, R. 1987. Limited processor scheduling of Doacross loops. In Sahni, S. K., Ed., *Proceedings of the International Conference on Parallel Processing*, (University Park, Pennsylvania, Aug.). Pennsylvania State University Press, pp. 226–234.

Cytron, R., Ferrante, J., Rosen, B. K., Wegman, M. N., and Zadeck, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, *13*, 4 (Oct.), 451–490.

Davidson, J. W. and Fraser, C. W. 1984. Code selection through object code optimization. *ACM Trans. Program. Lang. Syst.*, *6*, 4 (Oct.), 505–526.

Dennis, J. B. 1980. Data flow supercomputers. *Computer*, *13*, 11 (Nov.), 48–56.

Dixit, K. M. 1992. New CPU benchmarks from SPEC. In *Digest of Papers, Spring COMPCON 1992, Thirty-Seventh IEEE Computer Society International Conference*, (San Francisco, California, Feb.). IEEE Computer Society Press, Los Alamitos, California, pp. 305–310.

Dongarra, J. and Hind, A. R. 1979. Unrolling loops in fortran. *Software – Practice and Experience, 9*, 3 (Mar.), 219–226.

Eggers, S. J. and Katz, R. H. 1989. Evaluating the performance of four snooping cache coherency protocols. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, (Jerusalem, Israel, May). *Computer Architecture News, 17*, 3, 2–15.

Eigenmann, R., Hoeflinger, J., Li, Z., and Padua, D. A. 1991. Experience in the automatic parallelization of four Perfect-benchmark programs. In Banerjee, U. et al., Eds., *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*, (Santa Clara, California, Aug.). Springer Verlag, Berlin, Germany, pp. 65–83. Also available as Center for Supercomputing Research and Development Technical Report 1193.

Ellis, J. R. 1986. *Bulldog: A Compiler for VLIW Architectures.* ACM Doctoral Dissertation Award. MIT Press, Cambridge, Massachusetts. Based on the author's Ph.D. thesis at Yale University, 1984.

Ershov, A. P. 1966. ALPHA – an automatic programming system of high efficiency. *J. ACM, 13*, 1 (Jan.), 17–24.

Feautrier, P. 1988. Array expansion. In *Proceedings of the ACM International Conference on Supercomputing*, (St. Malo, France, July). ACM Press, New York, pp. 429–441.

Ferrante, J., Ottenstein, K. J., and Warren, J. D. 1987. The program dependence graph and its use in optimization. *ACM Trans. Program. Lang. Syst., 9*, 3 (July), 319–349.

Fisher, J. A. 1981. Trace scheduling: A technique for global microcode compaction. *IEEE Trans. Comput., C-30*, 7 (July), 478–490.

Fisher, J. A., Ellis, J. R., Ruttenberg, J. C., and Nicolau, A. 1984. Parallel processing: A smart compiler and a dumb machine. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, (Montreal, Canada, June). *SIGPLAN Notices, 19*, 6, 37–47.

Fisher, J. A. and Freudenberger, S. M. 1992. Predicting conditional branch directions from previous runs of a program. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, (Boston, Massachusetts, Oct.). *SIGPLAN Notices, 27*, 9, 85–95.

Floating Point Systems, Inc. 1979. *FPS AP-120B Processor Handbook.* Beaverton, Oregon.

Free Software Foundation. 1992. *gcc 2.x Reference Manual.*

Freudenberger, S. M., Schwartz, J. T., and Sharir, M. 1983. Experience with the SETL optimizer. *ACM Trans. Program. Lang. Syst., 5*, 1 (Jan.), 26–45.

Gannon, D., Jalby, W., and Gallivan, K. 1988. Strategies for cache and local memory management by global program transformation. *J. Parallel Distrib. Comput., 5*, 5 (Oct.), 587–616.

Gibson, D. H. and Rao, G. S. 1992. Design of the IBM System/390 computer family for numerically intensive applications: An overview for engineers and scientists. *IBM J. Res. Dev., 36*, 4 (July), 695–711.

Gifford, D. K. and Lucassen, J. M. 1986. Integrating functional and imperative programming. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, (Cambridge, Massachusetts, Aug.). ACM Press, New York, pp. 28–38.

Girkar, M. and Polychronopoulos, C. D. 1988a. Compiling issues for supercomputers. In *Proceedings of Supercomputing '88*, (Orlando, Florida, Nov.). IEEE Computer Society Press, Washington, D.C., pp. 164–173.

Girkar, M. and Polychronopoulos, C. D. 1988b. Partitioning programs for parallel execution. In *Proceedings of the ACM International Conference on Supercomputing*, (St. Malo, France, July). ACM Press, New York, pp. 216–229.

Goff, G., Kennedy, K., and Tseng, C.-W. 1991. Practical dependence testing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (Toronto, Ontario, June). *SIGPLAN Notices, 26*, 6, 15–29.

Graham, S. L., Lucco, S., and Sharp, O. J. 1993. Orchestrating interactions among parallel computations. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (Albuquerque, New Mexico, June). *SIGPLAN Notices, 28*, 6, 100–111.

Graham, S. L. and Wegman, M. 1976. A fast and usually linear algorithm for global flow analysis. *J. ACM*, *23*, 1 (Jan.), 172–202.

Granlund, T. and Kenner, R. 1992. Eliminating branches using a superoptimizer and the GNU C compiler. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (San Francisco, California, June). *SIGPLAN Notices*, *27*, 7, 341–352.

Harris, K. and Hobbs, S. to appear. VAX fortran. In Allen, F. E., Cytron, R., Rosen, B. K., and Zadeck, K., Eds., *Optimization in Compilers*, chapter 16. ACM Press, New York, New York.

Hecht, M. S. 1977. *Flow Analysis of Computer Programs.* North-Holland, New York, New York.

Hennessy, J. L. and Patterson, D. A. 1990. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, San Mateo, California.

Hewlett Packard. 1992. *PA-RISC 1.1 Architecture and Instruction Manual, 2nd ed.*, (Sept.). Part Number 09740-90039.

Hiranandani, S., Kennedy, K., and Tseng, C.-W. 1991. Compiler support for machine-independent programming in fortran D. In Salz, J. and Mehrotra, P., Eds., *Compilers and Runtime Software for Scalable Multiprocessors.* American Elsevier Publishing Company, New York, New York. Also available as Rice University Technical Report COMP TR91-149.

Hiranandani, S., Kennedy, K., and Tseng, C.-W. 1992. Compiling fortran D for MIMD distributed-memory machines. *Commun. ACM*, *35*, 8 (Aug.), 66–80.

Hudak, P. and Goldberg, B. 1985. Distributed execution of functional programs using serial combinators. *IEEE Trans. Comput.*, *C-34*, 10 (Oct.), 881–890.

Hummel, S., Schonberg, E., and Flynn, L. 1992. Factoring: A practical and robust method for scheduling parallel loops. *Commun. ACM*, *35*, 8 (Aug.), 90–101.

Hwu, W. W. and Chang, P. P. 1989. Achieving high instruction cache performance with an optimizing compiler. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, (Jerusalem, Israel, May). *Computer Architecture News*, *17*, 3, 242–251.

Hwu, W. W., Mahlke, S. A., Chen, W. Y., Chang, P. P., Warter, N. J., Bringmann, R. A., Ouellette, R. G., Hank, R. E., Kiyohara, T., Haab, G. E., Holm, J. G., and Lavery, D. M. 1993. The superblock: An effective technique for VLIW and superscalar compilation. *J. Supercomp.*, *7*, 1/2 (May), 229–248.

IBM. 1992. *Optimization and Tuning Guide for the XL* Fortran *and XL C Compilers*, first edition, (Sept.). Publication SC09-1545-00.

IBM International Technical Support Centers. 1991. *IBM RISC System/6000 NIC Tuning Guide for* Fortran *and C*, (July). Publication GG24-3611-01.

Irigoin, F. and Triolet, R. 1988. Supernode partitioning. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, (San Diego, California, Jan.). ACM Press, New York, pp. 319–329.

Iverson, K. E. 1962. *A Programming Language.* John Wiley and Sons, New York, New York.

Kennedy, K. and McKinley, K. S. 1990. Loop distribution with arbitrary control flow. In *Proceedings of Supercomputing '90*, (New York, New York, Nov.). IEEE Computer Society Press, Los Alamitos, California, pp. 407–416.

Kildall, G. 1973. A unified approach to global program optimization. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, (Boston, Massachusetts, Oct.). ACM Press, New York, pp. 194–206.

Knuth, D. E. 1971. An empirical study of fortran programs. *Software – Practice and Experience*, *1*, 2 (April-June), 105–133.

Kranz, D., Kelsey, R., Rees, J., Hudak, P., Philbin, J., and Adams, N. 1986. ORBIT: An optimizing compiler for Scheme. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, (Palo Alto, California, June). *SIGPLAN Notices*, *21*, 7, 219–233.

Kuck, D. J. 1977. A survey of parallel machine organization and programming. *Computing Surveys*, *9*, 1 (Mar.), 29–59.

Kuck, D. J. 1978. *The Structure of Computers and Computations*, volume 1. John Wiley and Sons, New York, New York.

Kuck, D. J., Kuhn, R. H., Padua, D., Leasure, B., and Wolfe, M. 1981. Dependence graphs and compiler optimizations. In *Proceedings of the Eighth Annual ACM Symposium on Principles of Programming Languages*, (Williamsburg, Virginia, Jan.). ACM Press, New York, pp. 207–218.

Lam, M. S. 1988. Software pipelining: An effective scheduling technique for VLIW machines. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (Atlanta, Georgia, June). *SIGPLAN Notices*, *23*, 7, 318–328.

Lam, M. S., Rothberg, E. E., and Wolf, M. E. 1991. The cache performance and optimization of blocked algorithms. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, (Santa Clara, California, Apr.). *SIGPLAN Notices*, *26*, 4, 63–74.

Lam, M. S. and Wilson, R. P. 1992. Limits of control flow on parallelism. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, (Gold Coast, Australia, May). *Computer Architecture News*, *20*, 2, 46–57.

Lamport, L. 1974. The parallel execution of DO loops. *Commun. ACM*, *17*, 2 (Feb.), 83–93.

Larus, J. R. 1993. Loop-level parallelism in numeric and symbolic programs. *IEEE Trans. Parallel Distrib. Syst.*, *4*, 7 (July), 812–826.

Lee, G., Kruskal, C. P., and Kuck, D. J. 1985. An empirical study of automatic restructuring of nonnumerical programs for parallel processors. *IEEE Trans. Comput.*, *C-34*, 10 (Oct.), 927–933.

Li, Z. and Yew, P. 1988. Program parallelization with interprocedural analysis. *J. Supercomp.*, *2*, 2 (Oct.), 225–244.

Li, Z., Yew, P., and Zhu, C. 1990. Data dependence analysis on multi-dimensional array references. *IEEE Trans. Parallel Distrib. Syst.*, *1*, 1 (Jan.), 26–34.

Loveman, D. B. 1977. Program improvement by source-to-source transformation. *J. ACM*, *1*, 24 (Jan.), 121–145.

Lucco, S. 1992. A dynamic scheduling method for irregular parallel programs. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (San Francisco, California, June). *SIGPLAN Notices*, *27*, 7, 200–211.

Markstein, P., Markstein, V., and Zadeck, K. to appear. Strength reduction. In Allen, F. E., Cytron, R., Rosen, B. K., and Zadeck, K., Eds., *Optimization in Compilers*, chapter 9. ACM Press, New York, New York.

Massalin, H. 1987. Superoptimizer: A look at the smallest program. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems*, (Palo Alto, California, Oct.). *SIGPLAN Notices*, *22*, 10, 122–126.

McFarling, S. 1991. Procedure merging with instruction caches. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (Toronto, Ontario, June). *SIGPLAN Notices*, *26*, 6, 71–79.

McFarling, S. and Hennessy, J. 1986. Reducing the cost of branches. In *Proceedings of the 13th Annual International Symposium on Computer Architecture*, (Tokyo, Japan, June). IEEE Computer Society Press, Los Alamitos, California, pp. 396–403.

McGraw, J. R. 1985. SISAL: streams and iteration in a single assignment language. Technical Report M-146, Lawrence Livermore National Laboratory, (Mar.).

McMahon, F. M. 1986. The Livermore fortran kernels: A computer test of numerical performance range. Technical Report UCRL-55745, Lawrence Livermore National Laboratory, (Dec.).

Michie, D. 1968. "Memo" functions and machine learning. *Nature*, *218*, 19–22.

Mirchandaney, R., Saltz, J. H., Smith, R. M., Nicol, D. M., and Crowley, K. 1988. Principles of runtime support for parallel processors. In *Proceedings of the ACM International Conference on Supercomputing*, (St. Malo, France, July). ACM Press, New York, pp. 140–152.

Morel, E. and Renvoise, C. 1979. Global optimization by suppression of partial redundancies. *Commun. ACM*, *22*, 2 (Feb.), 96–103.

Muchnick, S. S. and Jones, N., Eds. 1981. *Program Flow Analysis*. Prentice-Hall, Englewood Cliffs, New Jersey.

MURAOKA, Y. 1971. *Parallelism Exposure and Exploitation in Programs.* PhD thesis, University of Illinois at Urbana-Champaign, (Feb.). Technical Report 71-424.

NICOLAU, A. 1988. Loop quantization: A generalized loop unwinding technique. *J. Parallel Distrib. Comput., 5*, 5 (Oct.), 568–586.

NICOLAU, A. AND FISHER, J. A. 1984. Measuring the parallelism available for very long instruction word architectures. *IEEE Trans. Comput., C-33*, 11 (Nov.), 968–976.

NIKHIL, R. S. 1988. ID reference manual, version 88.0. Technical Report 284, MIT Laboratory for Computer Science.

NOBAYASHI, H. AND EOYANG, C. 1989. A comparison study of automatically vectorizing FORTRAN compilers. In *Proceedings of Supercomputing '89*, (Reno, Nevada, Nov.). ACM Press, New York, pp. 820–825.

O'BRIEN, K., HAY, B., MINISH, J., SCHAFFER, H., SCHLOSS, B., SHEPHERD, A., AND ZALESKI, M. 1990. Advanced compiler technology for the RISC System/6000 architecture. In *IBM RISC System/6000 Technology.* IBM Corporation, Mechanicsburg, Pennsylvania. Publication SA23-2619.

OED, W. 1992. Cray Y-MP C90: System features and early benchmark results. *Parallel Computing, 18*, 8 (Aug.), 947–954.

OEHLER, R. R. AND BLASGEN, M. W. 1991. IBM RISC System/6000: Architecture and performance. *IEEE Micro, 11*, 3 (June), 14–24.

OTTENSTEIN, K. J. AND OTTENSTEIN, L. M. 1984. The program dependence graph in a software development environment. In HENDERSON, P., ED., *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, (Pittsburgh, Pennsylvania, May). *SIGPLAN Notices, 19*, 5, 177–184.

PADUA, D. AND WOLFE, M. J. 1986. Advanced compiler optimizations for supercomputers. *Commun. ACM, 29*, 12 (Dec.), 1184–1201.

PADUA, D. A., KUCK, D. J., AND LAWRIE, D. 1980. High-speed multiprocessors and compilation techniques. *IEEE Trans. Comput., C-29*, 9 (Sept.), 763–776.

PAPADOPOULOS, G. M. AND CULLER, D. E. 1990. Monsoon: an explicit token-store architecture. In *Proceedings of the 17th Annual International Symposium on Computer Architecture*, (Seattle, Washington, May). *Computer Architecture News, 18*, 2, 82–91.

PETERSEN, P. M. AND PADUA, D. A. 1990? Machine-independent evaluation of parallelizing compilers. Technical Report 1173, Center for Supercomputing Research and Development, University of Illinois at Urbana-Champaign.

PETTIS, K. AND HANSEN, R. C. 1990. Profile guided code positioning. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (White Plains, New York, June). *SIGPLAN Notices, 25*, 6, 16–27.

POLYCHRONOPOULOS, C. D. 1987a. Advanced loop optimizations for parallel computers. In HOUSTIS, E. N., PAPATHEODOROU, T. S., AND POLYCHRONOPOULOS, C. D., EDS., *Proceedings of the First International Conference on Supercomputing*, (Athens, Greece, June). Springer Verlag, Berlin, Germany, pp. 255–277.

POLYCHRONOPOULOS, C. D. 1987b. Loop coalescing: A compiler transformation for parallel machines. In SAHNI, S. K., ED., *Proceedings of the International Conference on Parallel Processing*, (University Park, Pennsylvania, Aug.). Pennsylvania State University Press, pp. 235–242.

POLYCHRONOPOULOS, C. D. 1988. *Parallel Programming and Compilers.* Kluwer Academic Publishers, Boston, Massachusetts.

POLYCHRONOPOULOS, C. D. AND BANERJEE, U. 1986. Speedup bounds and processor allocation for parallel programs on a multiprocessor. In HWANG, K. ET AL., EDS., *Proceedings of the International Conference on Parallel Processing*, (St. Charles, Illinois, Aug.). IEEE Computer Society Press, Washington, D.C., pp. 961–968.

POLYCHRONOPOULOS, C. D., GIRKAR, M., HAGHIGHAT, M. R., LEE, C. L., LEUNG, B., AND SCHOUTEN, D. 1989. Parafrase-2: An environment for parallelizing, partitioning, synchronizing, and scheduling programs on multiprocessors. In *Proceedings of the International Conference on Parallel Processing*, volume II, (University Park, Pennsylvania, Aug.). Pennsylvania State University Press, pp. 39–48.

Polychronopoulos, C. D. and Kuck, D. J. 1987. Guided self-scheduling: A practical scheduling scheme for parallel supercomputers. *IEEE Trans. Comput., C-36*, 12 (Dec.), 1425–1439.

Pugh, W. 1991. Uniform techniques for loop optimization. In *Proceedings of the ACM International Conference on Supercomputing*, (Cologne, Germany, June). ACM Press, New York.

Rau, B. and Fisher, J. A. 1993. Instruction-level parallel processing: History, overview, and perspective. *J. Supercomp., 7*, 1/2 (May), 9–50.

Rau, B., Yen, D. W. L., Yen, W., and Towle, R. A. 1989. The Cydra 5 departmental supercomputer: Design philosophies, decisions, and trade-offs. *Computer, 22*, 1 (Jan.), 12–34.

Reif, J. H. and Lewis, H. R. 1986. Efficient symbolic analysis of programs. *J. Comput. Syst. Sci., 32*, 3 (June), 280–313.

Rinard, M. C., Scales, D. J., and Lam, M. S. 1993. Jade: A high-level machine-independent language for parallel programming. *Computer, 26*, 6 (June), 28–38.

Riseman, E. M. and Foster, C. C. 1972. The inhibition of potential parallelism by conditional jumps. *IEEE Trans. Comput., C-21*, 12 (Dec.), 1405–1411.

Sabot, G. and Wholey, S. 1993. CMAX: A fortran translator for the Connection Machine system. In *Proceedings of the ACM International Conference on Supercomputing.*

Sarkar, V. 1989a. Determining average program execution times and their variance. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (Portland, Oregon, June). *SIGPLAN Notices, 24*, 7, 298–312.

Sarkar, V. 1989b. *Partitioning and Scheduling Parallel Programs for Multiprocessors.* Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Massachusetts.

Sarkar, V. and Hennessey, J. 1986a. Compile-time partitioning and scheduling of parallel programs. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, (Palo Alto, California, June). *SIGPLAN Notices, 21*, 7, 17–26.

Sarkar, V. and Hennessey, J. 1986b. Partitioning parallel programs for macro dataflow. In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, (Cambridge, Massachusetts, Aug.). ACM Press, New York, pp. 202–211.

Sarkar, V. and Thekkath, R. 1992. A general framework for iteration-reordering transformations. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (San Francisco, California, June). *SIGPLAN Notices, 27*, 7, 175–187.

Scheifler. 1977. An analysis of inline substitution for a structured programming language. *Commun. ACM, 20*, 9 (Sept.), 647–654.

Selke, R. P. 1989. A rewriting semantics for program dependence graphs. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, (Austin, Texas, Jan.). ACM Press, New York, pp. 12–24.

Shen, Z., Li, Z., and Yew, P. 1989. An empirical study on array subscripts and data dependences. In *Proceedings of the International Conference on Parallel Processing*, volume II, (University Park, Pennsylvania, Aug.). Pennsylvania State University Press, pp. 145–152.

Singh, J. P., Weber, W.-D., and Gupta, A. 1992. SPLASH: Stanford parallel applications for shared memory. *Computer Architecture News, 20*, 1 (Mar.), 5–44. Also available as Stanford University technical report CSL-TR-92-526.

Sites, R. L., Ed. 1992. *Alpha Architecture Reference Manual.* Digital Press, Bedford, Massachusetts.

Smith, B. 1978. A pipelined, shared resource MIMD computer. In *Proceedings of the International Conference on Parallel Processing*, (Bellaire, Michigan, Aug.). IEEE, New York, NY, pp. 6–8.

Sohi, G. S. and Vajapayem, S. 1990. Instruction issue logic for high-performance, interruptable, multiple functional unit, pipelined computers. *IEEE Trans. Comput., C-39*, 3 (Mar.), 349–359.

Stenström, P. 1990. A survey of cache coherence schemes for multiprocessors. *Computer, 23*, 6 (June), 12–24.

Sun Microsystems. 1991. *SPARC Architecture Manual, Version 8.* Part No. 800-1399-08.

Szymanski, T. G. 1978. Assembling code for machines with span-dependent instructions. *Commun. ACM, 21*, 4 (Apr.), 300–308.

Tang, P. and Yew, P. 1990. Dynamic processor self-scheduling for general parallel nested loops. *IEEE Trans. Comput.*, *C-39*, 7 (July), 919–929.

Tarjan, R. E. 1979. Applications of path compression on balanced trees. *J. ACM*, *26*, 4 (Oct.), 690–715.

Thinking Machines Corporation. 1989. *Connection Machine, Model CM-2 Technical Summary.*

Tjaden, G. S. and Flynn, M. J. 1970. Detection and parallel execution of parallel instructions. *IEEE Trans. Comput.*, *C-19*, 10 (Oct.), 889–895.

Towle, R. A. 1976. *Control and Data Dependence for Program Transformations.* PhD thesis, University of Illinois at Urbana-Champaign, (Mar.). Technical Report 76-788.

Triolet, R., Irigoin, F., and Feautrier, P. 1986. Direct parallelization of call statements. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, (Palo Alto, California, June). *SIGPLAN Notices*, *21*, 7, 176–185.

Ullman, J. D. 1973. Fast algorithms for the elimination of common sub-expressions. *Acta Informatica*, *2*, 3 (July), 191–213.

von Hanxleden, R. and Kennedy, K. 1992. Relaxing SIMD control flow constraints using loop transformations. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, (San Francisco, California, June). *SIGPLAN Notices*, *27*, 7, 188–199.

Wall, D. W. 1986. Global register allocation at link time. In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, (Palo Alto, California, June). *SIGPLAN Notices*, *21*, 7, 264–275.

Wall, D. W. 1991. Limits of instruction-level parallelism. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, (Santa Clara, California, Apr.). *SIGPLAN Notices*, *26*, 4, 176–188.

Wang, K. 1991. *Intelligent Program Optimization and Parallelization for Parallel Computers.* PhD thesis, Purdue University, (Apr.). Technical Report CSD-TR-91-030.

Wang, K. and Gannon, D. 1989. Applying AI techniques to program optimization for parallel computers. In Hwang, K. and Degroot, D., Eds., *Parallel Processing for Supercomputers and Artificial Intelligence*, chapter 12. McGraw Hill Book Company, New York, New York.

Wedel, D. 1975. Fortran for the Texas Instruments ASC system. In *Programming Languages and Compilers for Parallel and Vector Machines*, (New York, New York, Mar.). *SIGPLAN Notices*, *10*, 3, 119–132.

Wegman, M. N. and Zadeck, F. K. 1991. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, *13*, 2 (Apr.), 181–210.

Wolf, M. E. and Lam, M. S. 1991. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parallel Distrib. Syst.*, *2*, 4 (Oct.), 452–471.

Wolfe, M. J. 1989a. More iteration space tiling. In *Proceedings of Supercomputing '89*, (Reno, Nevada, Nov.). ACM Press, New York, pp. 655–664.

Wolfe, M. J. 1989b. *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Massachusetts. Based on the author's Ph.D. thesis at the University of Illinois at Urbana-Champaign, 1982.

Wolfe, M. J. and Tseng, C. 1992. The power test for data dependence. *IEEE Trans. Parallel Distrib. Syst.*, *3*, 5 (Sept.), 591–601.