# 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 using transformations based on the analysis of scalar quantities and data-flow techniques. In contrast, optimizations for high-performance superscalar, vector, and parallel processors maximize parallelism and memory locality with transformations that rely on tracking the properties of arrays using loop dependence analysis.

This survey is a comprehensive overview of the important high-level program restructuring techniques for imperative languages such as C and Fortran. Transformations for both sequential and various types of parallel architectures are covered in depth. We describe the purpose of each transformation, explain how to determine if it is legal, and give an example of its application.

Programmers wishing to enhance the performance of their code can use this survey to improve their understanding of the optimizations that compilers can perform, or as a reference for techniques to be applied manually. Students can obtain an overview of optimizing compiler technology. Compiler writers can use this survey as a reference for most of the important optimizations developed to date, and as a bibliographic reference for the details of each optimization. Readers are expected to be familiar with modern computer architecture and basic program compilation techniques.

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

General Terms: Languages, Performance

Additional Key Words and Phrases: Compilation, dependence analysis, locality, multiprocessors, optimization, parallelism, superscalar processors, vectorization

## 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 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

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, and by an IBM Resident Study Program Fellowship to David Bacon.

CONTENTS



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.

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.

Most of the transformations we describe can be applied to any of the Algol-family languages; many are applicable to functional, logic, distributed, and object-oriented languages as well. These other languages raise additional optimization issues that 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 [Appel 1992; Clark and Peyton-Jones 1985; Kranz et al. 1986], object-oriented languages [Chambers and Ungar 1989], and the set-based language SETL [Freudenberger et al. 1983].

We have also restricted the discussion to higher-level transformations that require some program analysis. Thus we exclude peephole optimizations and most machine-level optimizations. 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.

We make use of a number of different machine models, all based on a hypothetical superscalar processor called S-DLX. The Appendix 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. Readers unfamiliar with program flow analysis or other basic compilation techniques may consult Aho et al. [1986].

## 1. SOURCE LANGUAGE

All of the high-level examples in this survey are written in a language similar

to Fortran 90, with minor variations and extensions; most examples use only features found in Fortran 77. 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. It is not yet clear what effect the recent introduction of pointers into Fortran 90 will have on optimization.

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 point to the same storage locations is called *alias analysis* [Banning 1979; Choi et al. 1993; Cooper and Kennedy 1989; Landi et al. 1993].

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.

We follow the Fortran convention that arrays are stored contiguously in memory in *column-major form*. If the array is traversed so as to visit consecutive locations in the linear memory, the first (that is, leftmost) subscript varies the fastest. In traversing the two-dimensional array declared as a[n, m], the following array locations are contiguous: a[n − 1, 3], a[n, 3], a[1, 4], a[2, 4].

Programmers unfamiliar with Fortran should also note that all arguments are passed by reference.

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, assuming a machine with vector registers of length 64. 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 register 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 6.2.4.

## 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.
(3) *Transform* the program.

In this survey we concentrate on the last step: transformation of the program.

However, in Section 5 we introduce dependence analysis techniques that are used for deciding upon and verifying many loop transformations.

Step (1) is the most difficult and poorly understood; consequently compiler design is in many ways a black art. Because analysis is expensive, engineering issues often constrain the optimization strategies that the compiler can perform. Even for relatively straightforward uniprocessor target architectures, it is possible for a sequence of optimizations to slow a program down. For example, an attempt to reduce the number of instructions executed may actually degrade performance by making less efficient use of the cache.

As processor architectures become more complex, (1) the number of dimensions in which optimization is possible increases and (2) the decision process is greatly complicated. Some progress has been made in systematizing the application of certain families of transformations, as discussed in Section 8. However, all optimizing compilers embody a set of heuristic decisions as to the transformation orderings likely to work best for the target machine(s).

### 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. The following is a more practical definition.

*Definition 2.1.1* A transformation is *legal* if the original and the transformed programs produce exactly the same output for all *identical executions*.

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

Nondeterminism 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 that forces **select** statements to evaluate their guards in a deterministic order. As a last resort, the programmer may have to replace nondeterministic system calls temporarily 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(a) shows a subroutine. Figure 1(b) is a transformed version of it. The new version may seem to have the same semantics as the original, but it violates Definition 2.1.1 in the following ways:

- *Overflow.* If b[k] is a very large number, and a[1] is negative, then changing the order of the additions so that C = b[k] + 100000.0 is executed before the loop body could 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 values that are left in the array a may be slightly different because the order of the additions has been changed. The reason is that float-

```
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] + 100000.0
end do
return
```

(a) original program

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

C = b[k] + 100000.0
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.

ing-point numbers are approximations of real numbers, and the order in which the approximations are applied (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 that are algebraically but not computationally commutative (or associative) *semicommutative* (or *semiassociative*) operations. These issues do not arise with Boolean operations, 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) to the transformed code in Figure 1(b).
- *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 bitwise identical results are desired, the following definition is used:

*Definition 2.1.2* 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 generally 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.

Fortran solves the problem of parameter aliasing by declaring calls that alias scalars or parts of arrays illegal, as in Figure 1(d). In practice, many programs make use of aliasing anyway. While this may improve performance, it sometimes leads to very obscure bugs.

For languages and standards that define a specific semantics for exceptions, meeting Definition 2.1.2 tightly constrains the permissible transformations. If the compiler may assume that exceptions are only generated when the program is semantically incorrect, a transformed program can produce different results when an exception occurs and still be legal. When exceptions have a well-defined semantics, as in the IEEE floating-point standard [American National Standards Institute 1987], many transformations will not be legal under this definition.

However, demanding bitwise identical results may not be necessary. An alternative correctness criterion is:

*Definition 2.1.3* 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 semicommutative operations are considered equivalent.

Since we cannot predict the degree to which transformations of semicommutative operations change the output, we must use an operational rather than an observational definition of equivalence. Generally, in practice, programmers observe whether the numeric results differ by more than a certain tolerance, and if they do, force the compiler to employ Definition 2.1.2.

### 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 target high-performance architectures effectively, compilers need to focus on loops. Most of the transformations discussed in this survey are based on loop manipulation. Many of them have been studied or widely applied only 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 of only the next loop in the nest. Because a perfect nest is more easily summarized and reorganized, several transformations apply only 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*.
- *Interprocedural*. 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 interprocedural analysis.

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

## 3. TARGET ARCHITECTURES

In this survey we discuss compilation techniques for high-performance archi-

tectures, 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 the size of total memory required.

While optimization for scalar CPUs has concentrated on minimizing the dynamic instruction count (or more precisely, the number of machine cycles required), CPU-intensive 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 that 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. Other compound operations are sometimes implemented as well.

Use of such compound operations may allow an application to run up to twice as fast. It is therefore important to organize the code so as to use multiply-adds wherever possible.

### 3.1 Characterizing Performance

There are a number of variables that we have used to help describe the performance of the compiled code quantitatively:

- $S$ is the hardware speed of a single processor in operations per second. Typically, speed is measured either in millions of instructions per second (MIPS) or in millions of floating-point operations per second (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. A superscalar machine can issue several instructions per cycle, but if the program does not contain a mixture of operations that matches the mixture of functional units, utilization will be lower. Similarly, a vector machine cannot be utilized fully unless the sizes of all of the arrays in the program are an exact multiple of the vector length.
- $Q$ measures reuse of operands that are stored in memory. It is the ratio of the number of times the operand is referenced during computation to the number of times it is loaded into a register. As the value of $Q$ rises, more reuse is being achieved, and less memory bandwidth is consumed. Values of $Q$ below

1 indicate that redundant loads are occurring.

- $Q_C$ is an analogous quantity that measures reuse of a cache line in memory. It is the ratio of the number of words that are read out of the cache from that particular line to the number of times the cache fetches that line from memory.

### 3.2 Model Architectures

In the Appendix 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 (V-DLX), a shared-memory multiprocessor (sMX), and a distributed-memory multiprocessor (dMX). 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

Figure 2 shows the design of a hypothetical compiler for a superscalar or vector machine. It includes most of the general-purpose transformations covered in this survey. Because compilers for parallel machines are still a very active area of research, we have excluded these machines from the design.

The purpose of this compiler design is to give the reader (1) an idea of how the various types of transformations fit together and (2) some of the ordering issues that arise. This organization is by no means definitive: different architectures dictate different designs, and opinions differ on how best to order the transformations.

Optimization takes place in three distinct phases, corresponding to three different representations of the program: high-level intermediate language, low-level intermediate language, and object code. First, optimizations are applied to a high-level intermediate language (HIL) that is semantically very close to the original source program. Because all the semantic information of the source program is available, higher-level transformations are easier to apply. For instance, array references are clearly distinguishable, instead of being a sequence of low-level address calculations.

Next, the program is translated to low-level intermediate form (LIL), essentially an abstract machine language, and optimized at the LIL level. Address computations provide a major source of potential optimization at this level. For instance, two references to a[5, 3] and a[7, 3] both require the computation of the address of column 3 of array a.

Finally, the program is translated to object code, and machine-specific optimizations are performed. Some optimizations, like code colocation, rely on profile information obtained by running the program. These optimizations are often implemented as binary-to-binary translations.

The high-level optimization phase begins by doing all of the large-scale restructuring that will significantly change the organization of the program. This includes various procedure-restructuring optimizations, as well as scalarization, which converts data-parallel constructs into loops. Doing this restructuring at the beginning allows the rest of the compiler to work on individual procedures in relative isolation.

Next, high-level data-flow optimizations, partial evaluation, and redundancy elimination are performed. These optimizations simplify the program as much as possible, and remove extraneous code that could reduce the effectiveness of subsequent analysis.

The main focus in compilers for high-performance architectures is loop optimizations. A sequence of transformations is performed to convert the loops into a form that is more amenable to optimization: where possible, perfect loop nests are created; subscript expressions are

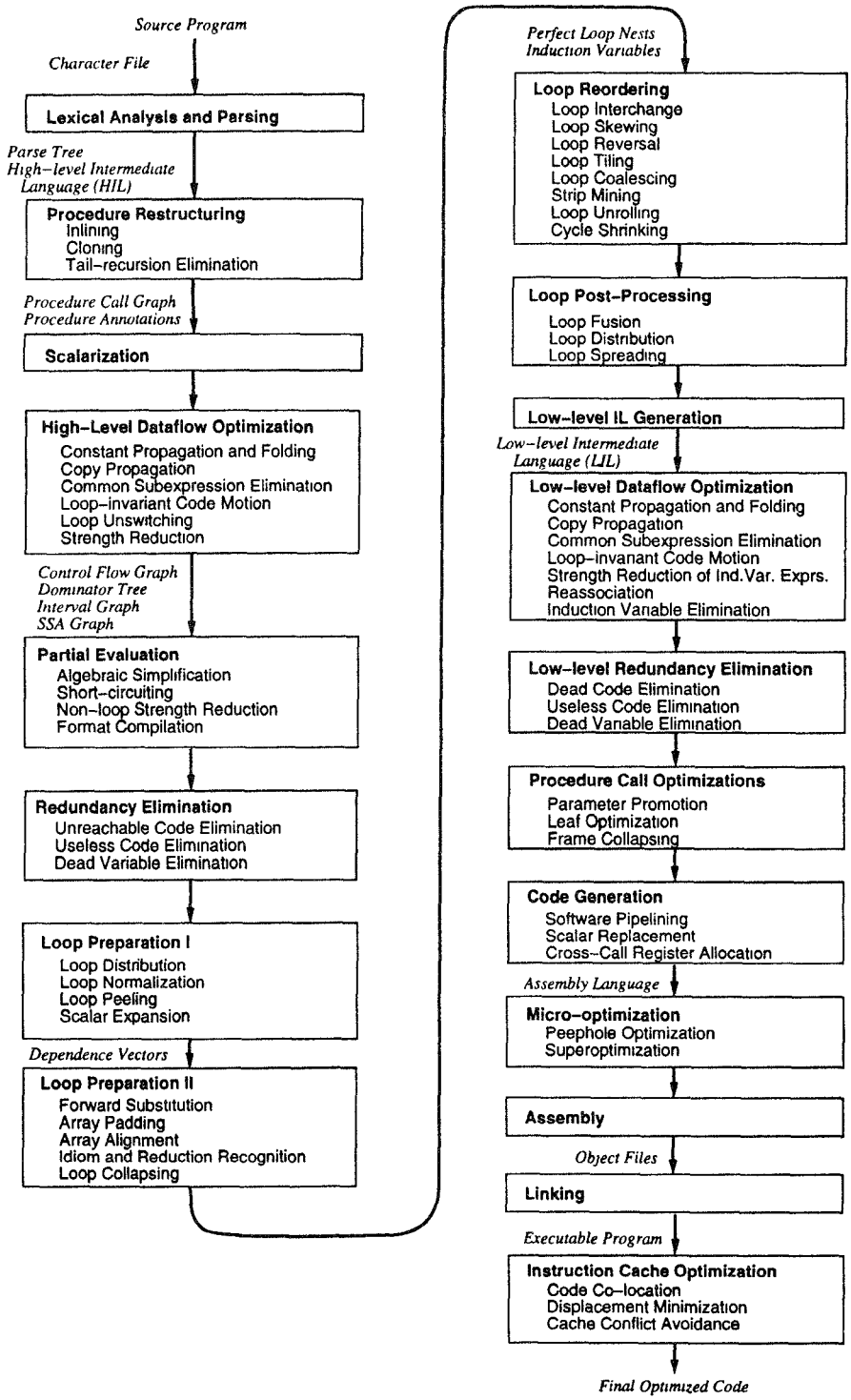



**Figure 2.** Organization of a hypothetical optimizing compiler.

rewritten in terms of the induction variables; and so on. Then the loop iterations are reordered to maximize parallelism and locality. A postprocessing phase organizes the resulting loops to minimize loop overhead.

After the loop optimizations, the program is converted to low-level intermediate language (LIL). Many of the data-flow and redundancy elimination optimizations that were applied to the HIL are reapplied to the LIL in order to eliminate inefficiencies in the component expressions generated from the high-level constructs. Additionally, induction variable optimizations are performed, which are often important for high performance on uniprocessor machines. A final LIL optimization pass applies procedure call optimizations.

Code generation converts LIL into assembly language. This phase of the compiler is responsible for instruction selection, instruction scheduling, and register allocation. The compiler applies low-level optimizations during this phase to improve further the performance of the code.

Finally, object code is generated by the assembler, and the object files are linked into an executable. After profiling, profile-based cache optimizations can be applied to the executable program itself.

## 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* [Banerjee 1988b; Wolfe 1989b]. This section introduces dependence analysis, its terminology, and the underlying theory.

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 control 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*), *antidependence*, 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 antidependence on $S_5$ (denoted by $S_5 \not\rightarrow S_6$) when $S_6$ writes a variable that is read by $S_5$:

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

An antidependence 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 $S_7 \not\leftrightarrow S_8$. Again, as with an antidependence, 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.

The fourth possible relationship, called an *input dependence*, holds when two accesses to the same memory location are both reads. Although an input dependence imposes no ordering constraints, the compiler can make note of it for the purposes of optimizing data placement on multiprocessors.

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 $S_3 \rightarrow S_4$, $S_5 \bar{\delta} S_6$ for $S_5 \not\rightarrow S_6$, and $S_7 \delta^0 S_8$ for $S_7 \not\leftrightarrow S_8$.

In the case of data dependences, when we write $X \Rightarrow Y$ we are being somewhat imprecise: the individual variable references within a statement generate the dependences, not the statement as a whole. 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. $S_7 \not\leftrightarrow S_8$ 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 dependence information for a piece of code, the compiler creates a *dependence graph*; typically each node in the graph represents one statement. An arc between two nodes indicates that there is a dependence between the computations they represent.

Because it can be cumbersome to account for both control and data dependence during analysis, sometimes compilers 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.

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

**Figure 3.** Loop-carried dependence.

### 5.3 Loop Dependence Analysis

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

A simple example of *loop-carried* dependence is shown in Figure 3. 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. When $i = k$, $S_2$ 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 to understand 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 dependence in the loop nest, it is sufficient to determine whether any of the iterations can write a value that is read or written by any of the other iterations.

Depending on the language, loop increments may be arbitrary expressions. However, the dependence analysis algorithms may require that the loops have only unit increments. When they do not, the compiler may be able to normalize them to fit the requirements of the analysis, as described in Section 6.3.6. For the remainder of this section we will assume that all loops are incremented by 1 for each iteration.

```
do i1 = l1, u1
   do i2 = l2, u2
      ...
         do id = ld, ud
1           a[f1(i1, ..., id), ..., fm(i1, ..., id)] = ...
2           ... = a[g1(i1, ..., id), ..., gm(i1, ..., id)]
         end do
      ...
   end do
end do
```

**Figure 4.** General loop nest.

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 function $f_i$ and $g_i$ map the current values of the loop iteration variables to integers that index 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 depend only on 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 on 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 caused by 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 5(a). 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 $J$. Extending the 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 5(a), 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 $0 \prec$

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

(a) $\{(1,-1)\}$

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

(b) $\{(0,1),(1,0),(1,-1)\}$

**Figure 5.** Distance vectors.

$V$ (the first nonzero 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 nonzero element was negative, this would indicate a dependence on a future iteration, which is impossible.

The reader should note that distance vectors describe dependences among *iterations*, not among *array elements*. The operations on array elements create the dependences, but the distance vectors describe dependences among iterations. For instance, the loop nest that updates the one-dimensional array a in Figure 5(b) has dependences described by the set of two-element distance vectors $\{(0, 1), (1, 0), (1, -1)\}$.

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 enough information to partially characterize the dependence. A *direction vector* (introduced by Wolfe [1989b]) is commonly used to describe such dependences.

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

$$w_p = \begin{cases} < & \text{if } i_p < j_p \\ = & \text{if } i_p = j_p \\ > & \text{if } i_p > j_p. \end{cases}$$

We will use the general term *dependence vector* to encompass both distance and direction vectors. In Figure 5 the direction vector for loop (a) is $(<, >)$, and the direction vectors for loop (b) are $\{(=, <), (<, =), (<, >)\}$. Note that a direction vector entry of $<$ corresponds to a distance vector entry that is *greater than zero*.

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 5(b) can be summarized as $(\leq, *)$. The symbol $\neq$ denotes both a $<$ and $>$ direction, and $*$ denotes $<$, $>$ and $=$.

A particular dependence between statements $S_1$ and $S_2$ is denoted by writing the dependence vector above the arrow, for example $S_1 \overset{(<)}{\rightarrow} S_2$.

Burke and Cytron [1986] present a framework for dependence analysis that defines a hierarchy of dependence vectors and allows flow and antidependences to be treated symmetrically. An antidependence is simply a flow dependence with an impossible dependence vector ($V \prec 0$).

### 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, first the compiler 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. When dependences are found, the compiler tries to describe them with a direction or distance vector. If the subscript expressions are too complex to analyze, the compiler assumes the statements are fully dependent on one another such that no change in execution order is permitted.

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].

Additionally, there are a large number of *exact* tests that exploit some subscript characteristics to determine whether a particular type of dependence exists. One of the less expensive exact tests is the Single-Index Test [Banerjee 1979; Wolfe 1989b]. The Delta Test [Goff et al. 1991] is a more general strategy that examines some combinations of dimensions. Other tests that are more expensive to evaluate consider the multiple subscript dimensions simultaneously, such as the $\lambda$-test [Li et al. 1990], multidimensional GCD [Banerjee 1988a], and the power test [Wolfe and Tseng 1992]. The omega test [Pugh 1992] uses a linear programming algorithm to solve the dependence equations. The SUIF compiler project at Stanford has had success applying a series of exact tests, starting with the cheaper ones [Maydan et al. 1991]. They use a general algorithm (Fourier-Motzkin variable elimination [Dantzig and Eaves 1974]) as a backup.

One advantage of the multiple-dimension exact tests is their ability to handle *coupled subscript* expressions [Goff et al. 1991]. Two expressions are coupled if the same variable appears in both of them. Figure 6 shows a loop that has no dependences between iterations. The statement reads the values to the right of the diagonal and updates the diagonal. A test that only examines one dimension of the subscript expressions at a time, however, will not detect the independence because there are many pairs of iterations where the expression i in one is equal to the expression i + 1 in the other. The more general exact tests would discover that the iterations are independent despite the presence of coupled subscript expressions.

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

**Figure 6.** Coupled subscripts.

Section 9.2 discusses a number of studies that examine the subscript expressions in scientific applications and evaluate the effectiveness of different dependence tests.

## 6. TRANSFORMATIONS

This section catalogs general-purpose program transformations; those transformations that are only applicable on parallel machines are discussed in Section 7. The primary emphasis is on loops, since that is generally where most of the execution time is spent. For each transformation, we provide an example, discuss the benefits and shortcomings, identify any variants, and provide citations.

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.

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 draw also on previous summaries [Allen and Cocke 1971; Kuck 1977; Padua and Wolfe 1986; Rau and Fisher 1993; Wolfe 1989b].

Because some transformations were already familiar to programmers who applied them manually, often we cite only the work of researchers who have systematized and automated the implementation of these transformations. Additionally, we omit citations to works that are restricted to basic blocks when global optimization techniques exist. Even so, the origin of some optimizations is murky. For instance, Ershov's ALPHA compiler

[Ershov 1966] performed interprocedural constant propagation, albeit in a limited form, in 1964!

### 6.1 Data-Flow-Based Loop Transformations

A number of classical loop optimizations are based on data-flow analysis, which tracks the flow of data through a program's variables [Muchnick and Jones 1981]. These optimizations are summarized by Aho et al. [1986].

#### *6.1.1 Loop-Based Strength Reduction*

Reduction in strength replaces an expression in a loop with one that is equivalent but uses a less expensive operator [Allen 1969; Allen et al. 1981]. Figure 7(a) shows a loop that contains a multiplication. Figure 7(b) is a transformed version of the loop where the multiplication has been replaced by an addition.

Table 1 shows how strength reduction can be applied to a number of operations. The operation in the first column is assumed to appear within a loop that iterates over $i$ from 1 to $n$. When the loop is transformed, the compiler initializes a temporary variable T with the expression in the second column. The operation within the loop is replaced by the expression in the third column, and the value of T is updated each iteration with the value in the fourth.

A variable whose value is derived from the number of iterations that have been executed by an enclosing 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.

The most common use of strength reduction, often implemented as a special case, is strength reduction of induction variable expressions [Aho et al. 1986; Allen 1969; Allen et al. 1981; Cocke and Schwartz 1970].

Strength reduction can be applied to products involving induction variables by converting them to references to an equivalent running sum, as shown in Figure 8(a–c). This special case 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.

The term *strength reduction* is also applied to operator substitution in a nonloop context, like replacing $x \times 2$ with $x + x$. These optimizations are covered in Section 6.6.7.

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

(a) original loop

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

(b) after strength reduction

**Figure 7.** Strength reduction example.

**Table 1.** Strength Reductions (the variable $c$ is loop invariant; $x$ may vary between iterations)

| Expression | Initialization | Use | Update |
|---|---|---|---|
| $c \times i$ | $T = c$ | $T$ | $T = T + c$ |
| $c^i$ | $T = c$ | $T$ | $T = T \times c$ |
| $(-1)^i$ | $T = -1$ | $T$ | $T = -T$ |
| $x/c$ | $T = 1/c$ | $x \times T$ | |

#### *6.1.2 Induction Variable Elimination*

Once strength reduction has been performed on induction variable expressions, the compiler can often eliminate the original variable entirely. The loop exit test must then be expressed in terms of one of the strength-reduced induction variables; the optimization is called *linear function test replacement* [Allen 1969; Aho et al. 1986]. The replacement not only reduces the number of operations in

```
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 8.** 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 9.** Loop-invariant code motion.

a loop, but frees the register used by the induction variable.

Figure 8(d) shows the result of applying induction variable elimination to the strength-reduced code from the previous section.

### 6.1.3 Loop-Invariant Code Motion

When a computation appears inside a loop, but its result does not change between iterations, the compiler can move that computation outside the loop [Aho et al. 1986; Cocke and Schwartz 1970].

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 multidimensional arrays or dereferencing pointers, as when the inner loop of a C program contains an expression like a.b → c.d[i].

Figure 9(a) shows an example in which an expensive transcendental function call is moved outside of the inner loop. The test in the transformed code in Figure 9(b) ensures that if the loop is never executed, the moved code is not executed either, lest it raise an exception.

The precomputed value is generally assigned to a register. If registers are scarce, and the expression moved is inexpensive to compute, code motion may actually deoptimize the code, since register spills will be introduced in the loop.

Although code motion is sometimes referred to as *code hoisting*, hoisting is a

```
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 > 1) 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 10.** Loop unswitching.

more general term referring to any transformation that moves a computation to an earlier point in the program [Aho et al. 1986]. While loop-invariant code motion is one particularly common form of hoisting, there are others: an expression can be evaluated earlier to reduce register pressure or avoid arithmetic unit latency, or a load instruction might be moved upward to reduce the effect of memory access latency.

#### *6.1.4 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 the analysis for code motion, which identifies loop-invariant values.

In Figure 10(a) 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 10(b). 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 whether it should be executed at all. The test for the inner loop will be repeated every time the outer loop is executed. When the compiler uses an intermediate representation of the program that exposes the test explicitly, unswitching can be applied to move the test outside of the outer loop. The RS/6000 XL C/Fortran compiler uses unswitching for this purpose [O'Brien et al. 1990].

### 6.2 Loop Reordering

In this section we describe transformations that change the relative order of execution of the iterations of a loop nest or nests. These transformations are primarily used to expose parallelism and improve memory locality.

Some compilers only apply reordering transformations to perfect loop nests (see Section 6.2.7). To increase the opportunities for optimization, such a compiler can sometimes apply *loop distribution* to extract perfect loop nests from an imperfect nesting.

The compiler determines whether a loop can be executed in parallel by examining the loop-carried dependences. 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. Figure 11(a) is an example; the distance vector for the loop is (0, 1), so the outer loop is parallelizable.

```
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 11.** Dependence conditions for parallelizing loops.

More generally, the $p$th loop in a loop nest is parallelizable if for every distance vector $V = (v_1, \ldots, v_p, \ldots, v_d)$,

$$v_p = 0 \vee \exists q < p : v_q > 0.$$

In Figure 11(b), the distance vectors are $\{(1,0),(1,-1)\}$, so the inner loop is parallelizable. 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.

### 6.2.1 Loop Interchange

Loop interchange exchanges the position of two loops in a perfect loop nest, generally moving one of the outer loops to the innermost position [Allen and Kennedy 1984; 1987; Wolfe 1989b]. Interchange is one of the most powerful transformations and can improve performance in many ways.

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 outward in a loop nest to increase the granularity of each iteration and reduce the number of barrier synchronizations;
- reduce stride, ideally to stride 1; and
- increase the number of loop-invariant expressions in the inner loop.

```
double precision a[*]

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

(a) loop with varying stride

| stride | Cache Misses | TLB Misses | Relative Speed (%) |
|---|---|---|---|
| 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 | 8 |

(b) effect of stride

**Figure 12.** 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 [IBM 1992].

Care must be taken that these benefits do not cancel each other out. For instance, an interchange that improves register reuse may change a stride-1 access pattern to a stride-$n$ access pattern with much lower overall performance due to increased cache misses. Figure 12 demonstrates the dramatic effect of different strides on an IBM RS/6000.

In Figure 13(a), the inner loop accesses array a with stride n (recall that we are assuming the Fortran convention of column-major storage order). By interchanging the loops, we convert the inner loop to stride-1 access, as shown in Figure 13(b).

For a large array in which less than one column fits in the cache, this opti-

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

(a) original loop nest

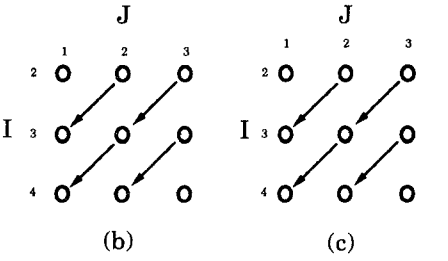
```
do j = 1,n
  do i = i,n
    total[i] = total[i] + a[i,j]
  end do
end do
```

(b) interchanged loop nest

**Figure 13.** Loop interchange.

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

(a)




(b) (c)

**Figure 14.** Original loop (a); original traversal order (b); traversal order after interchange (c).

mization reduces the number of cache misses on a from $n^2$ to $n^2 \times$ elementsize/linesize, or $n^2/16$ with 4-byte elements and the 64-byte lines of S-DLX. However, the original loop allows total[i] to be placed in a register, eliminating the load/store operations in the inner loop (see *scalar replacement* in Section 6.5.4). So the optimized version increases the number of load/store operations for total from $2n$ to $2n^2$. If a fits in the cache, the original loop is better.

On a vector architecture, 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 when 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.

A loop nest of just two loops can be interchanged unless it has a dependence vector of the form $(<, >)$. Figure 14(a) shows the loop nest with the dependence $(1, -1)$, giving rise to the loop-carried dependences shown in Figure 14(b). The order in which the iterations are executed is shown by the dotted line. The traversal order after interchange is shown in Figure 14(c): some iterations are executed before iterations that they depend on, so the interchange is illegal.

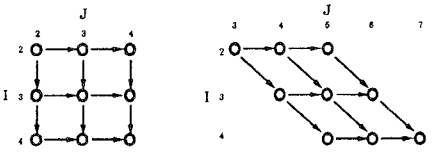
Switching the loop bounds is straightforward when the iteration space is rectangular, as in the loop nest in Figure 13. In this case the bounds of the inner loop are independent of the indices of the containing loop, and the two can simply be exchanged. When the iteration space is not rectangular, computing 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). Further techniques are necessary to manage imperfectly nested loops. Some of the variations are discussed in detail by Wolfe [1989b] and Wolf and Lam [1991].

### 6.2.2 Loop Skewing

Loop skewing is an enabling transformation that is primarily useful in combination with loop interchange [Lamport 1974; Muraoka 1971; Wolfe 1989b]. Skewing was invented to handle wavefront computations, so called because the updates to the array propagate like a wave across the iteration space.

```
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 $\{(1,0),(0,1)\}$




(b) original space (c) skewed space

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

(d) skewed code· dependences $\{(1,1),(0,1)\}$

```
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-1-i] +
                a[i+1,j-i] + a[i,j+1-i])/4
  end do
end do
```

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

**Figure 15.** Loop skewing.

In Figure 15(a) we show a typical wavefront computation. Each element is computed by averaging its four nearest neighbors. While neither of the loops is parallelizable in their original form, each array diagonal ("wavefront") could be computed in parallel. The iteration space and dependences are shown in Figure 15(b), with the dotted lines indicating the wavefronts.

Skewing is performed by 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. Because it alters the loop bounds but then alters the uses of the corresponding index variables to compensate, skewing does not change the meaning of the program and is always legal.

The original loop nest in Figure 15(a) can be interchanged, but neither loop can be parallelized, because there is a dependence on both the inner loop (0, 1) and on the outer loop (1, 0).

The result when $f = 1$ is shown in Figure 15(c–d). 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.

To expose this parallelism, the skewed loop nest must also be interchanged. After skew and interchange, the loop nest has distance vectors $\{(1,0),(1,1)\}$. 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.

Skewing can expose parallelism for a nest of two loops with a set of distance vectors $\{V_k\}$ only if

$$(\exists V_i = (v_{i1}, v_{i2}) : v_{i1} = 0 \wedge v_{i2} > 0) \wedge$$
$$(\exists V_j = (v_{j1}, v_{j2}) : v_{j1} > 0 \wedge v_{j2} \leq 0).$$

When skewing by $f$, the original distance vector $(v_1, v_2)$ becomes $(v_1, fv_1 + v_2)$. For any dependence where $v_2 \leq 0$, the goal is to find $f$ such that $fv_1 + v_2 \geq 1$. The correct skew factor $f$ is computed by taking the maximum of $f_i = \lceil (1 - v_{i2})/v_{i1} \rceil$ over all the dependences [Kennedy et al. 1993].

Interchanging of skewed loops is complicated by the fact that their bounds depend on the iteration variables of the enclosing loops. For two loops with bounds $i_1 = l_1, u_1$ and $i_2 = l_2, u_2$ where $l_2$ and $u_2$ are expressions independent of $i_1$, the skewed inner loop has bounds $i_2 = fi_1 + l_2, fi_1 + u_2$. After interchange, the bounds are

$$\text{do } i_2 = fl_1 + l_2, fu_1 + u_2$$
$$\quad \text{do } i_1 = \max(l_1, \lceil (i_2 - u_2)/f \rceil), \min(u_1, \lceil (i_2 - l_2)/f \rceil)$$

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

### 6.2.3 Loop Reversal

Reversal changes the direction in which the loop traverses its iteration range [Wedel 1975]. It is often used in conjunction with other 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 that the iteration variable runs down to zero, allowing the loop to end with a branch-if-not-equal-to-zero instruction (BNEZ, on S-DLX).

Reversal can also eliminate the need for temporary arrays in implementing Fortran 90 array statements (see Section 6.4.3).

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 direction vectors $\{(<, =), (<, >)\}$ can be reversed, because the resulting dependences are all still lexicographically positive.

Figure 16 shows how reversal can enable loop interchange: the original loop nest (a) has the distance vector $(1, -1)$ that prevents interchange because the resulting distance vector $(-1, 1)$ is not lexicographically positive; the reversed loop nest (b) can legally be interchanged.

```
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 16.** Loop reversal.

```
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 update of a[TI:TI+63]

**Figure 17.** Strip mining.

### 6.2.4 Strip Mining

Strip mining is a method of adjusting the granularity of an operation, especially a parallelizable operation [Abu-Sufah et al. 1981; Allen 1983; Loveman 1977].

An example is shown in Figure 17. 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.

One of the most common uses of strip mining is to choose the number of independent computations in the innermost

loop of a nest. On a vector machine, for example, the serial loop can then be converted into a series of vector operations, each vector comprising a single "strip" [Cray Research 1988]). Strip mining is also used for SIMD compilation [Weiss 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].

Strip mining often requires other transformations to be performed first. Loop distribution (see Section 6.2.7) can expose simple loops from within an original loop nest that is too complex to strip mine. Loop interchange can be used to move a parallelizable loop into the innermost position of a loop nest or to maximize the length of the strip.

The examples given above demonstrate how strip mining can create a larger unit of work out of smaller ones. The transformation can also be used in the reverse direction, as shown in Section 7.4.

### 6.2.5 Cycle Shrinking

Cycle shrinking is essentially a specialization of strip mining. When a loop has dependences that prevent it from being executed in parallel (that is, converted to a **do all**), the compiler may still be able to expose some parallelism if the dependence distance is greater than one. In this case cycle shrinking will convert a serial loop into an outer serial loop and an inner parallel loop [Polychronopoulos 1987a]. Cycle shrinking is primarily useful for exposing fine-grained parallelism.

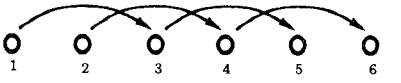
For instance, in Figure 18(a), a[i + k] is written in iteration i and read in iteration i + k; the dependence distance is k. Consequently the first k iterations can be performed in parallel provided that none of the subsequent iterations is allowed to begin until the first k are complete. The same is then done for the next k iterations, as shown in Figure 18(b). The iteration space dependences are shown in Figure 18(c): each group of k iterations is dependent only on the previous group.

```
  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 18.** Cycle shrinking.

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 limited to exposing parallelism that can be exploited at the instruction level (for instance by loop unrolling). Note that k must be constant within the loop and must at least be known to be positive at compile time.

### 6.2.6 Loop Tiling

Tiling is the multidimensional generalization of strip mining. Tiling (also called *blocking*) is primarily used to improve cache reuse ($Q_c$) by dividing an iteration space into tiles and transforming the loop nest to iterate over them [Abu-Sufah et al. 1981; Gannon et al. 1988; Lam et al. 1991; Wolfe 1989a]. However, it can also be used to improve processor, register, TLB, or page locality.

The need for tiling is illustrated by the loop in Figure 19(a) that assigns a the transpose of b. With the j loop innermost, access to b is stride-1, while access to a is

```
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) tiled loop

**Figure 19.** Loop tiling.

```
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.

stride-n. Interchanging does not help, since it makes access to b stride-n. By iterating over subrectangles of the iteration space, as shown in Figure 19(b), the loop uses every cache line fully.

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

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

### 6.2.7 Loop Distribution

Distribution (also called *loop fission* or *loop splitting*) breaks a single loop into many. Each of the new loops has the same iteration space as the original, but contains a subset of the statements of the original loop [Kuck 1977; Kuck et al. 1981; Muraoka 1971].

Distribution is used to

- create perfect loop nests;
- create subloops 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 reuse 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.

Distribution may be applied to any loop, but all statements belonging to a dependence cycle (called a $\pi$-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, distribution by name can increase cache

locality. Note that the above loop cannot be distributed using fission by name, since both statements reference a.

#### 6.2.8 Loop Fusion

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

- 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 Figure 20, distribution enables the parallelization of part of the loop. However, fusing the two loops improves register and cache locality since a[i] need only be loaded once. Fusion also increases instruction parallelism by increasing the ratio of floating-point operations to integer operations in the loop and reduces loop overhead by a factor of two. With large n, the distributed loop should run faster on a vector machine while the fused loop should be better on a superscalar machine.

For two loops to be fused, 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 6.3.5) or by introducing conditional expressions into the body of the loop. Two loops with the same bounds may be fused if there do not exist statements $S_1$ in the first loop and $S_2$ in the second such that they have a dependence $S_2 \overset{(<)}{\Rightarrow} S_1$ 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 alters execution order illegally, as shown in Figure 21.

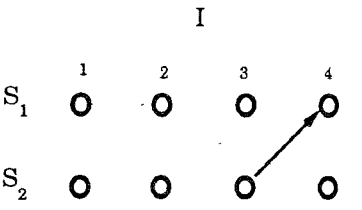



**Figure 21.** Two loops containing $S_1$ and $S_2$ cannot be fused when $S_2 \overset{<}{\Rightarrow} S_1$ in the fused loop.

### 6.3 Loop Restructuring

This section describes loop transformations that change the structure of the loop, but leave the computations performed by an iteration of the loop body and their relative order unchanged.

#### 6.3.1 Loop Unrolling

Unrolling replicates the body of a loop some number of times called the unrolling factor ($u$) and iterates by step $u$ instead of step 1. 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].

Unrolling can improve the performance by

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

In Figure 22, 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. 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 array elements are assigned to registers (either directly or by using *scalar replacement*, as described in Section 6.5.4), register locality will improve because a[i] and a[i + 1] are used twice in

```
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 22.** Loop unrolling.

the loop body, reducing the number of loads per iteration from 3 to 2.

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

The **if** statement at the end of Figure 22(b) is the *loop epilogue* that must be generated when it is not known at compile time whether the number of iterations of the loop will be an exact multiple of the unrolling factor $u$. If $u > 2$, the loop epilogue is itself a loop.

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

Figure 23(a–c) shows the effects of unrolling in more detail. The source code in Figure 23(a) is translated to the assembly code in Figure 23(b), which takes 6 cycles per result on S-DLX. After unrolling 3 times, the code requires 8 cycles per iteration or $2^2/_3$ cycles per result, as shown in Figure 23(c). 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. Outer loop unrolling is not as universal because it yields replicated instances of the inner loops. To avoid the additional control overhead, the compiler can often fuse the copies back together, yielding the same loop structure that appeared in the original code. This combination of transformations is sometimes referred to as *unroll-and-jam* [Callahan et al. 1988].

*Loop quantization* [Nicolau 1988] is another approach to unrolling that avoids replicated inner loops. Rather than creating multiple copies and then subsequently eliminating them, quantization adds additional statements to the innermost loop directly. The iteration ranges are changed, but the structure of the loop nest remains the same. However, unlike straightforward unrolling, quantization changes the order of execution of the loop and is not always a legal transformation.

### 6.3.2 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 23(d) shows how software pipelining improves the performance of a simple loop. The depth of the pipeline is $s = 3$. The software pipelined loop produces one new result every iteration, or 4 cycles per result.

```
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 8(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 and rescheduling (loop prologue and 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 and rescheduling, but without unrolling (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 then software pipelining (loop prologue and epilogue omitted)

**Figure 23.** Increasing instruction parallelism in loops.

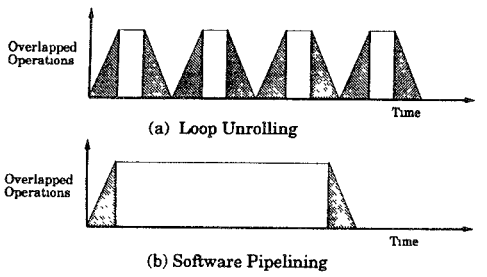



**Figure 24.** Loop unrolling vs. software pipelining.

Finally, Figure 23(e) shows the result of combining software pipelining ($s = 3$) with unrolling ($u = 2$). The loop takes 5 cycles per iteration, or $2^1/_2$ cycles per result. Unrolling alone achieves $2^2/_3$ 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 optimal because only one memory operation can be initiated per cycle.

Figure 24 illustrates the difference between unrolling and software pipelining: unrolling reduces overhead, while pipelining reduces the startup cost of each iteration.

*Perfect pipelining* combines unrolling by loop quantization with software pipelining [Aiken and Nicolau 1988a; 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 each iteration contains one load and one store. The RS/6000 XL C/Fortran compiler [O'Brien et al. 1990] performs this optimization.

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. The scheduling constraints in the presence of loop-carried dependences and conditionals are more complex; the details are discussed in Aiken and Nicolau [1988a] and Lam [1988].

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

(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 25.** Loop coalescing vs. collapsing.

### 6.3.3 Loop Coalescing

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

In Figure 25(a), for example, if n and m are slightly larger than the number of processors $P$, then neither of the loops schedules as well as the outer parallel loop, since executing the last $n - P$ iterations will take the same time as the first $P$. Coalescing the two loops ensures that $P$ iterations can be executed every time except during the last ($nm$ mod $P$) iterations, as shown in Figure 25(b).

Coalescing itself is always legal since it does not change the iteration order of the loop. The iterations of the coalesced loop may be parallelized if all the original loops were parallelizable. A criterion for

parallelizability in terms of dependence vectors is discussed in Section 6.2.

The complex subscript calculations introduced by coalescing can often be simplified to reduce the overhead of the coalesced loop [Polychronopoulos 1987b].

#### 6.3.4 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 multidimensional 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 25(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.

#### 6.3.5 Loop Peeling

When a loop is *peeled*, a small number of iterations are removed from the beginning or end of the loop and executed separately. If only one iteration is peeled, a common case, the code for that iteration can be enclosed within a conditional. For a larger number of iterations, a separate loop can be introduced. Peeling has two uses: for removing dependences created by the first or last few loop iterations, thereby enabling parallelization; and for matching 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 = $3 \cdots$ n. Peeling off the first iteration allows the rest of the loop to be parallelized and fused with the following loop, as shown in Figure 26(b).

```
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.

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

#### 6.3.6 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 is to permit the compiler to apply subscript analysis tests, many of which require normalized iteration ranges.

#### 6.3.7 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 1988].

An example is shown in Figure 28: the two loops in the original program (a) cannot be fused because they have different

```
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.

```
    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 28.** Loop spreading.

bounds, and there would be a dependence $S_2 \overset{(2)}{\nrightarrow} S_1$ 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 in Figure 28(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 from the first loop, at the cost of additional loop overhead.

### 6.4 Loop Replacement Transformations

This section describes loop transformations that operate on whole loops and completely alter their structure.

#### *6.4.1 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 29(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.

In Figure 29(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 semicommutative and semiassociative operators like floating-point multiplication, the validity of the transformation depends on the language se-

```
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 29.** Reduction recognition.

mantics and the programmer's intent, as described in Section 2.1. Provided that bitwise identical results are not required, the partial sums can be computed in parallel.

Maximum parallelism is achieved by computing the reduction with a tree: pairs of elements are summed, then pairs of these results are summed, and so on. The number of serial steps is reduced from $O(n)$ to $O(\log n)$.

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

#### 6.4.2 Loop Idiom Recognition

Parallel architectures often provide specialized hardware that the compiler can take advantage of. Frequently, for example, SIMD machines support reduction directly in the processor interconnection network. Some parallel machines, such as the Connection Machine CM-2 [Thinking Machines 1989], include hardware not only for reduction but for parallel prefix operations, allowing a loop with a body 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], by Kuck [1977; 1978], and Wolfe [1989b]. Blelloch [1989] describes compilation strategies for recognizing and exploiting parallel prefix operations.

The compiler can recognize and convert other idioms that are specially supported by hardware or software. 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]. The VAX Fortran compiler converts string copy loops into block transfer instructions [Harris and Hobbs 1994].

#### 6.4.3 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 requires that the operation be performed as if every value on the right-hand side and every subexpression on the left-hand side were computed before any assignments are performed.

The example in Figure 30 shows a computation in array notation (a), its "obvious" (but incorrect) conversion to serial form (b), and a correct conversion (c). The reason that Figure 30(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. The increments are to happen as if they were all performed simultaneously; in the incorrect version, 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 Figure 30(c). The temporary array can then be eliminated if the two loops can be legally fused, namely, when there is no dependence $S_2 \overset{(<)}{\Rightarrow} S_1$ in the fused loop, where $S_1$ is the assignment to the temporary, and $S_2$ is the assignment to the original array.

```
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 30.** Array statement scalarization.

In this case there is an antidependence, but it can be removed by reversing the loops, enabling fusion, and eliminating the temporary, as shown in Figure 30(d). However, the array language statement in Figure 30(e) requires a temporary since an antidependence exists regardless of the direction of the loop.

### 6.5 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 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:

- *Reuse*, denoted by $Q$ and $Q_C$, the ratio of uses of an item to the number of times it is loaded (described in Section 3);
- *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 quadword load and store instructions;
- *Working Set Size*. If all 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 $Q_C$. If more variables are simultaneously live than there are available registers (that is, the register pressure is high), 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 may thrash.

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 1982; Chaitin et al. 1981; Chow and Hennessy 1990] yielded very efficient global (within a procedure) register allocation. Most compilers now make use of some variant of graph coloring in their register allocator.

In general, memory optimizations are inhibited by low-level coding that relies on particular storage layouts. Fortran EQUIVALENCE statements are the most obvious example. C and Fortran 77 both specify the storage layout for each type of declaration. Programmers relying on a particular storage layout may defeat a wide range of important optimizations.

Optimizations covered in other sections that also can improve memory system performance are loop interchange (6.2.1), loop tiling (6.2.6), loop unrolling (6.3.1), loop fusion (6.2.8), and various optimizations that eliminate register saves at procedure calls (6.8).

### 6.5.1 Array Padding

Padding is a transformation whereby unused data locations are inserted between the columns of an array or between arrays. Padding is used to ameliorate a number of memory system conflicts, in particular:

- bank conflicts on vector machines with banked memory [Burnett and Coffman, Jr. 1970];
- cache set or TLB set conflicts;
- cache miss jamming [Bacon et al. 1994]; and
- false sharing of cache lines on shared-memory multiprocessors.

Bank conflicts on vector machines like the Cray and our hypothetical V-DLX can be caused when the program indexes through an array dimension that is not laid out contiguously in memory, leading to a nonunit stride whose value we will define to be $s$. If $s$ is an exact multiple of the number of banks ($B$), the bandwidth of the memory system will be reduced by a factor of $B$ (8, on V-DLX) because all memory accesses are to the same bank. The number of memory banks is usually a power of two.

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

Cached memory systems, especially those that are set-associative, are less sensitive to low power-of-two strides. However, large power-of-two strides will cause extremely poor performance due to cache set and TLB set conflicts. The problem is that addresses are mapped to sets simply by using a range of bits from the middle of the address. For instance, on S-DLX, the low 6 bits of an address are the byte offset within the line, and the next 8 bits are the cache set. Figure 12 shows the effect of stride on the performance of a cache-based superscalar machine.

```
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 V-DLX

**Figure 31.** Array padding.

A number of researchers have noted that other strides yield reduced cache performance. Bailey [1992] has quantified this by noting that because many words fit into a cache line, if the number of sets times the number of words per set is *almost* exactly divisible by the stride (say by $n \pm \epsilon$), then for a limited number of references $r$, every $n$th memory reference will map to the same set. If $\lfloor r/n \rfloor$ is greater than the associativity of the cache, then the later references will evict the lines loaded by the earlier references, precluding reuse.

Set and bank conflicts can be caused by a bad stride over a single array, or by a loop that accesses multiple arrays that all align to the same set or bank. Thus padding can be inserted between columns of an array (intraarray padding), or between arrays (interarray padding).

A further performance artifact, called *cache miss jamming*, can occur on machines that allow processing to continue during a cache miss: if cache misses are spread nonuniformly across the loop iterations, the asynchrony of the processor will not be exploited, and performance will be reduced. Jamming typically occurs when several arrays are accessed with the same stride and when all have the same alignment relative to cache line boundaries (that is, the same low address bits). Bacon et al. [1994] describe cache miss jamming in detail and present a unified framework for inter- and intraarray padding to handle set conflicts and jamming.

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, padding reduces the benefits of loop collapsing (see Section 6.3.4).

### 6.5.2 Scalar Expansion

Loops often contain variables that are used as temporaries within the loop body. Such variables will create an antidependence $S_2 \overset{(<)}{\nrightarrow} S_1$ from one iteration to the next, and will have no other loop-carried dependences. Allocating one temporary for each iteration removes the dependence and makes 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 [Kuck and Stokes 1982] and Cray-1 [Russell 1978] compilers. An alternative for parallel machines is to use *private variables*, where each processor has its own instance of the variable; these may be introduced by the compiler (see Section 7.1.3) or, if the language supports private variables, by the programmer.

If the compiler vectorizes or parallelizes a loop, scalar expansion must be performed for any compiler-generated temporaries in a loop. To avoid creating unnecessarily large temporary arrays, a vectorizing compiler can perform scalar expansion after strip mining, expanding the temporary to the size of the vector strip.

```
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.

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

### 6.5.3 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 the $p$th loop in a loop nest 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 distance vectors $V$ 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.

```
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.

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 6.2.4) and dynamic allocation of temporaries, either from the heap or from a static block of memory reserved for temporaries.

### 6.5.4 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 [Callahan et al. 1990].

Replacement multiplies $Q$ for the array element by the number of iterations in the inner loop(s). It can also eliminate unnecessary subscript calculations, although that optimization is often done by loop-invariant code motion (see Section 6.1.3). Loop interchange can be used to enable or improve scalar replacement; Carr [1993] examines the combination of scalar replacement, interchange, and unroll-and-jam in the context of cache optimization.

An example of scalar replacement is shown in Figure 34; for a discussion of the interaction between replacement and loop interchange, see Section 6.2.1.

### 6.5.5 Code Colocation

Code colocation improves memory access behavior by placing related code in close proximity. The earliest work rearranged code (often at the granularity of a procedure) to improve paging behavior [Ferrari 1976; Hatfield and Gerald 1971].

More recent strategies focus on improving cache behavior 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 [Hwu and Chang 1989; Pettis and Hansen 1990].

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 pro-

cedures (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 that starts from the procedure entry node. Basic blocks with a frequency estimate of zero can be moved to a separate page to increase locality further. However, accessing that page 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.

*Procedure inlining* (see Section 6.8.5) can also affect code locality, and has been studied both in conjunction with [Hwu and Chang 1989] and independent of [McFarling 1991] code positioning. Inlining improves performance often by reducing overhead and increasing locality, but if a procedure is called more than once in a loop, inlining will often increase the number of cache misses because the procedure body will be loaded more than once.

#### 6.5.6 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 few as 4 bits. If control is transferred to a location outside of the range of the offset, a multi-instruction sequence or long-format instruction is required to perform the jump. For instance, the S-DLX instruction BEQZ R4, error is only legal if error is within $2^{15}$ bytes. Otherwise, the instruction must be replaced with:

```
    BNEZ   R4,  cont          ;reversed test
    LI     R8,  error         ;get low bits
    LUI    R8,  error ≫ 16    ;get high bits
    JR     R8                 ;jump to target
cont:
```

This sequence requires three extra instructions. Given the cost of long displacement jumps, the code should be organized to keep related sections close together in memory, in particular those sections 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 /big / q, r, x[20000], y, z
```

If the array x contains word-sized elements, the common block 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. The problem is avoided if the layout of big is:

```
common /big / q, r, y, z, x[20000]
```

### 6.6 Partial Evaluation

Partial evaluation refers to the general technique of performing part of a computation at compile time. Most of the classical optimizations based on data-flow analysis are either a form of partial evaluation or of redundancy elimination (described in Section 6.7). Loop-based data-flow optimizations are described in Section 6.1.

#### 6.6.1 Constant Propagation

Constant propagation [Callahan et al. 1986; Kildall 1973; Wegman and Zadeck 1991] is one of the most important optimizations that a compiler can perform, and a good optimizing compiler will apply it aggressively. Typically programs contain many constants; by propagating them through the program, the compiler can do a significant amount of precomputation. More important, the propagation reveals many opportunities for other optimizations. In addition to obvious possibilities such as dead-code elimination, loop optimizations are affected because constants often appear in their induction ranges. Knowing the range of the loop, the compiler can be much more accurate

```
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.

```
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.

in applying the loop optimizations that, more than anything else, determine performance on high-speed machines.

Figure 35 shows a simple example of constant propagation. On V-DLX, 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 6.2.4), increasing the overhead of the loop.

### 6.6.2 Constant Folding

Constant folding is a companion to constant propagation: when an expression contains an operation with constant values as operands, the compiler can replace the expression with the result. For example, `x = 3**2` becomes `x = 9`. Typically constants are propagated and folded simultaneously [Aho et al. 1986].

Note that constant folding may not be legal (under Definition 2.1.2 in Section 2.1) if the operations performed at compile time are not identical to those that would have been performed at run time; a common source of such problems is the rounding of floating-point numbers during input or output between phases of the compilation process [Clinger 1990; Steele and White 1990].

### 6.6.3 Copy Propagation

Optimizations such as induction variable elimination (6.1.2) and common-subexpression elimination (6.7.4) may cause the same value to be copied several times. The compiler can propagate the original name of the value and eliminate 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.

### 6.6.4 Forward Substitution

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

For instance, in Figure 37(a) the loop cannot be parallelized because an unknown element of `a` is being written. After forward substitution, as shown in Figure 37(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 6.4.1).

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

```
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.** Forward substitution.

Forward substitution is generally performed on array subscript expressions at the same time as loop normalization (Section 6.3.6). For efficient subscript analysis techniques to work, the array subscripts must be linear functions of the induction variables.

### 6.6.5 Reassociation

Reassociation is a technique for increasing the number of common subexpressions in a program [Cocke and Markstein 1980; Markstein et al. 1994]. It is generally applied to address calculations within loops when performing strength reduction on induction variable expressions (see Section 6.1.1). 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.

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

### 6.6.6 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` if `x` and `y` are integers. Figure 38 illustrates some of the commonly applied rules.

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

**Figure 38.** Algebraic identities used in expression simplification.

**Table 2.** Identities Used in Strength Reduction (& is the string concatenation operator)

| Expression | Reduced Expr. | Datatypes |
|---|---|---|
| $x \times 2$ | $x + x$ | integer, real |
| $x^2$ | $x \times x$ | integer, real |
| $x^{c.5}$ | $x^c \times \sqrt{x}$ | real |
| $i \times 2^c$ | $i \ll c$ | integer |
| $(a,0)+(b,0)$ | $(a+b,0)$ | complex |
| $\text{len}(s_1 \ \& \ s_2)$ | $\text{len}(s_1)+\text{len}(s_2)$ | string |

Floating-point arithmetic can be problematic to simplify; for example, if x is an IEEE floating-point number with value Nan (not a number), then $x \times 0 = x$, instead of 0.

### 6.6.7 Strength Reduction

The identities in Table 2 are called *strength reductions* because they replace an expensive operator with an equivalent less expensive operator. Section 6.1.1 discusses the application of strength reduction to operations that appear inside a loop.

The two entries in the table that refer to multiplication, $x \times 2 = x + x$ and $i \times 2^c = i \ll c$, can be generalized. Multiplication by any integer constant can be performed using only shift and add instructions [Bernstein 1986]. Other transformations are conditional on the values of the affected variables; for example, $i/2^c = i \gg c$ if and only if $i$ is nonnegative [Steele 1977].

It is also possible to convert exponentiation to multiplication in the evaluation

```
      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)
```

(b) after format compilation

**Figure 39.** Format compilation.

of polynomials, using the identity

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

#### 6.6.8 I/O Format Compilation

Most languages provide fairly elaborate facilities for formatting input and output. The formatting specifications are in effect a formatting sublanguage that is generally "interpreted" at run time, with a correspondingly high cost for character input and output.

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 39 shows two I/O statements and their compiled equivalents. Idiom recognition has been performed to convert the implied **do** loop into an aggregate operation.

Note that in Fortran, a **format** statement is analogous to 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 (inlining is described in Section 6.8.5).

Format compilation is done by the VAX Fortran compiler [Harris and Hobbs 1994] and by the Gnu C compiler [Free Software Foundation 1992].

Format compilation is further complicated in C by the fact that **printf** and **scanf** are library functions and may be redefined by the programmer.

#### 6.6.9 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 at compile time is practical only for short sequences (on the order of a dozen instructions). It can also be used by the compiler designer to find optimal sequences for commonly occurring idioms. It 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].

### 6.7 Redundancy Elimination

There are many optimizations that 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 6.1.3. There a computation was being performed repeatedly when it could be done once.

Redundancy-eliminating transformations remove two other kinds of computations: those that are *unreachable* and those that are *useless*. A computation is unreachable if it is never executed; removing it from the program will have no semantic effect on the instructions executed. Unreachable code is created by programmers (most frequently with conditional debugging code), or by transfor-

mations that have left "orphan" code behind.

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

### 6.7.1. 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 6.6.1. In Figure 40(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 Figure 40(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, *redundant-control elimination* removes control constructs such as loops and conditionals when they become redundant (usually as a result of constant propagation). In Figure 40(b), the loop and conditional control expressions are not used, and we can remove them from the program, as shown in (c).

```
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
a = b+7
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 40.** Redundancy elimination.

### 6.7.2 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 state-

ment 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-known data-flow problem [Aho et al. 1986]. In Figure 40(c), the values computed by the assignment statements are no longer used; they have been eliminated in (d).

#### 6.7.3 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 40(d), the variables c, n, and debug are no longer used and can be removed; (e) shows the code after the variables are pruned.

#### 6.7.4 Common-Subexpression Elimination

In many cases, a set of computations will contain identical subexpressions. The redundancy can arise in both user code and in address computations generated by the compiler. The compiler can compute the value of the subexpression once, store it, and reuse the stored result [Aho et al. 1977; 1986; Cocke 1970]. Common-subexpression elimination is an important transformation 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 actually deoptimize the program.

#### 6.7.5 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 control expression in

```
if ((a = 1) and (b = 2)) then
  c = 5
end if
```

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

```
    if (not (a = 1)) goto 10
    if (not (b = 2)) goto 10
    c = 5
10  continue
```

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 on the language semantics. Some language definitions, including C and many dialects of Lisp, address this problem by *requiring* short circuiting of boolean expressions.

### 6.8 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 procedure's body;
- eliminating some of the entry/exit overhead; and
- avoiding some steps in making a procedure call when the behavior of the called procedure is known or can be altered.

#### 6.8.1 A Calling Convention for S-DLX

To demonstrate the procedure call optimizations, we first define a calling convention for S-DLX. Table 3 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 downward. 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.

**Table 3.** S-DLX Registers and Their Usage

| *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 |

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 can be used as a frame pointer, or the size of the frame can be stored at the top of the stack (R30 + 0).

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 41 shows the layout of the stack frame for a procedure invocation.

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

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 that must be preserved across the call) are saved on the stack. If the procedure makes any procedure calls itself, 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 42 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

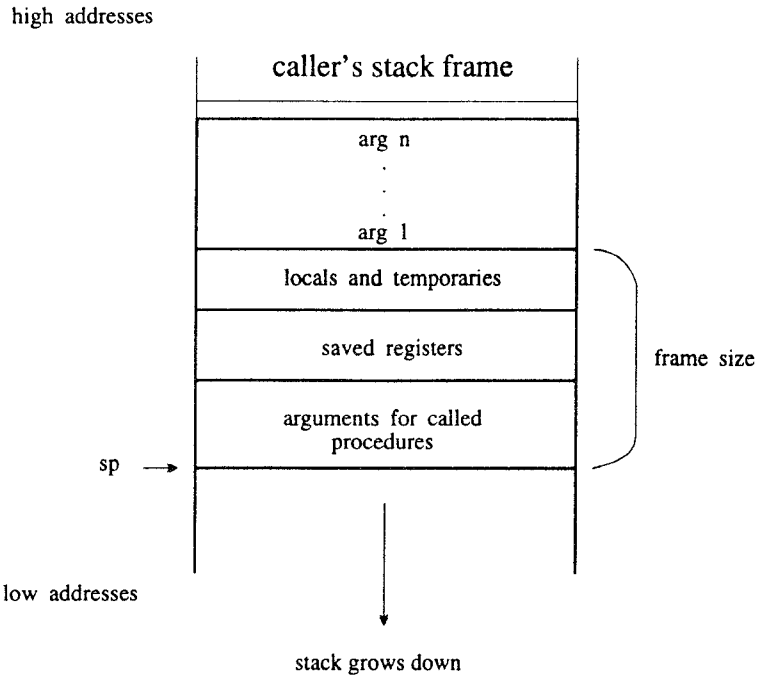



**Figure 41.** Stack frame layout.

must be preserved because it is overwritten by the jump-and-link (JAL) instruction; R16 must be preserved because it is used to hold c across the call.

The procedure first allocates the 12 bytes for the stack frame and saves R31 and R16; then the parameters are loaded into registers. 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.

### 6.8.2 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 static call graph. The simplest optimization for leaf procedures is that they do not need to save and restore the return address (R31). Additionally, if the procedure does not have any local variables allocated to memory, the compiler does not need to create a stack frame.

Figure 43(a) shows the function max (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.

### 6.8.3 Cross-Call Register Allocation

Separate compilation reduces the amount of information available to the compiler about called procedures. 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

```
integer function foo(c, b)
integer c, b
integer d, e
d = c+b
e = max(b, d)
foo = e+c
return
end
```

(a) source code for function foo

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

(b) compiled code for foo

**Figure 42.** Function foo and its compiled code.

registers (R8 – R15), the caller can leave values in the unused registers throughout execution of the callee. Additionally, move instructions for parameters can be eliminated.

To perform this optimization on a procedure f, registers must first have been allocated for each of the procedures that f calls. This can be done by performing register allocation on procedures ordered by a depth-first postorder traversal of the call graph [Chow 1988].

For example, in Figure 43(c) max uses only R8 – R10. Figure 44 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 addresses in R11. Additionally, R12 is used instead of R16, allowing the save and restore of R16 to be eliminated. Only d remains in the stack frame.

```
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), R31  ;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 43.** Leaf procedure optimization.

```
foo: SUBI R30, #4     ;adjust SP
     MOV  R11, R31    ;save retaddr
     LW   R12, (R2)   ;R12=c
     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+c
     ADDI R30, #4     ;restore SP
     JR   R11         ;return
```

**Figure 44.** Cross-call register allocation for foo.

#### 6.8.4 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*(R*n*) 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 cannot 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.

An unmodified reference parameter can be passed by value, and a modified reference parameter can be passed by value-result. Figure 45(a) shows max after this transformation has been applied. Figure 45(b) 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, so *frame collapsing* can be applied, and the stack frame is eliminated. In general, when the compiler can statically identify all the callers of a leaf procedure, it can expand their stack frames to include enough space for both procedures. The leaf procedure simply uses the caller's stack frame without doing any new allocation of its own.

Parameter promotion is particularly important for languages such as Fortran, in which all argument passing is by reference. However, the argument-passing semantics of value-result are not the same as for arguments passed by reference, in particular in the event of aliasing. Interprocedural analysis may be necessary to determine the legality of the transformation.

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

(a) 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=c
     LW   R2, (R3)    ;R2=b
     ADD  R3, R12, R2 ;R3=d
     JAL  max         ;call max
     ADD  R1, R1, R12 ;R1=e+c
     JR   R11         ;return
```

(b) foo after parameter promotion on max

**Figure 45.** Parameter promotion.

#### 6.8.5 Procedure Inlining

Procedure inlining (also known as *procedure integration*) replaces a procedure call with a copy of the body of the called procedure [Allen and Cocke 1971; Ball 1979; Scheifler 1977]. Each occurrence of a formal parameter is replaced with a version of the corresponding actual parameter, modified to reflect the calling convention of the language. Renaming may also be required for the local variables of the inlined procedure if they conflict with the calling procedure's variable names or if the procedure is inlined more than once in the same caller.

Inlining can almost always be performed, except when the procedure in question is recursive. Even recursive routines may benefit from a finite number of inlining steps, particularly when a constant argument allows the compiler to determine the number of recursive calls that will be performed. 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 frames 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.

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

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

(a) original code

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

(b) after inlining

**Figure 46.** Procedure inlining.

Another reason for inlining is to improve compiler analysis and optimization. In many compilers, a loop containing a procedure call cannot 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, interprocedural analysis can be costly and increases the complexity of the compiler.

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 therefore cause additional memory accesses. Furthermore, 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 from a few places). 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 into foo.

```
foo: LW   R12, (R2)    ;R12=c
     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+c
     JR   R31          ;return
```

**Figure 47.** max inlined into foo.

Ignoring cache effects, assume the following: 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 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.

Some recent studies [Cooper et al. 1991; 1992] have demonstrated that realizing the theoretical benefits of inlining may require some modification of the compiler, because inlined code has different characteristics than human-written code.

Peculiarities of a language or a compiler may reduce the effectiveness of inlining or produce unexpected results. Cooper et al. examined several commer-

cial Fortran compilers and uncovered the following potential problems: (1) increased demand for registers; (2) larger numbers of local variables, sometimes exceeding a compiler's limit for the number of analyzable variables; (3) loss of information due to the Fortran convention that procedure parameters may be assumed to be unaliased.

### 6.8.6 Procedure Cloning

Procedure cloning [Cooper et al. 1993] is a technique for improving optimization across procedure call boundaries. The call sites of the procedure being cloned are divided into groups, and a specialized version of the procedure is created for each group. The specialization often provides opportunities for better optimization, particularly due to improved constant propagation.

In Figure 48, 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)

subroutine 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)

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

(b) after cloning

**Figure 48.** Procedure cloning.

### 6.8.7 Loop Pushing

Loop pushing (also called *loop embedding* [Hall et al. 1991]) moves a loop nest from the caller to a cloned version of the called procedure. If a compiler does not perform vectorization or parallelization across procedure calls directly, pushing is a less general way of achieving a similar effect.

For example, 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 49, 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 interprocedural. If there are no other statements in the loop, the transformation is always legal.

*Procedure inlining* (see Section 6.8.5) is a different way of achieving a very

```
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 49.** Loop pushing.

similar effect, and does not require interprocedural analysis.

### 6.8.8 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 50 shows an example of a tail-recursive function (a) and the result after the recursion is eliminated (b). A function that is not tail recursive is shown in Figure 50(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.

Recursive programs can be transformed automatically into tail-recursive versions that can be executed iteratively [Burstall and Darlington 1977], but this is not commonly performed by existing compilers for imperative languages. Some languages prevent tail recursion elimination 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.

Other languages, like Scheme [Rees et al. 1986], require that all tail-recursive procedures be identified and that they be executed without consuming stack space proportional to the recursion depth.

```
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 that is not tail-recursive

**Figure 50.** Tail recursion elimination.

### 6.8.9 Function Memoization

Memoization is an optimization that is applied to side-effect free procedures (that is, procedures that 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. When the procedure is called again with the same arguments, the cached result is used instead of recomputing it [Abelson and Sussman 1985; Michie 1968].

Figure 51 shows a simple example of memoization. If f is often called with the same arguments and f also takes a nontrivial amount of time to run, then memoization will substantially increase

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

(a) original function call

```
logical f_UNCACHED[n]
real    f_CACHE[n]
do i = 1, n
  f_UNCACHED[i] = .true.
end do

...

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 51.** Function memoization

performance. If not, it will degrade performance and consume memory.

This example assumes that f's parameter is confined to the range $1 \cdots n$, 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 reuse 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 $r$, the time to execute the $c$ calls is

$$T = c(rt_h + (1 - r)t_m)$$

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.

## 7. TRANSFORMATIONS FOR PARALLEL MACHINES

All the transformations discussed so far are applicable to a wide variety of computer organizations. In this section we describe transformations that are specific to parallel architectures.

Highly efficient parallel applications have been developed by manually applying the transformations discussed here to sequential code. Duplicating these successes by automatically transforming the code within a compiler is an active area of research. While a great deal of work has been done, automatic parallelization of full-scale applications with existing compilers does not consistently yield significant speedups. Experiments that measure the effectiveness of parallelizing compilers are presented in Section 9.3.

Because the parallelization of sequential code is proving to be difficult, languages like HPF Fortran [High Performance Fortran Forum 1993] provide constructs that allow the programmer to declare data decomposition strategies and to expose parallelism. The compiler is then responsible for generating communication and synchronization code, using various optimizations (discussed below) to reduce the resulting overhead. As of yet, however, there are few real applications that have been written in these languages, and there is little experimental data.

In this section we make extensive use of our model shared-memory and distributed-memory multiprocessors sMX and dMX, which are described fully in the Appendix. Both machines are constructed using S-DLX processors. The programming environment initializes the global variable Pnum to be the number of processors in the machine and Pid to be the number of the local processor (starting with 0).

### 7.1 Data Layout

One of the key decisions that a compiler must make in targeting a multiprocessor is the decomposition of data across the processors. The need to distribute data and manage it is obvious in a distributed-memory programming model, where communication is explicit. The performance of code under a shared-memory model is also dependent on

avoiding unnecessary communication: the fact that communication is performed implicitly does not eliminate its cost.

### *7.1.1 Regular Array Decomposition*

The most common strategy for decomposing an array on a parallel machine is to map each dimension to a set of processors in a regular pattern. The four commonly used patterns are discussed in the following sections.

#### Serial Decomposition

Serial decomposition is the degenerate case, where an entire dimension is allocated to a single processor. Figure 52(a) shows a simple loop nest that adds c to each element of a two-dimensional array, the decomposition of the array onto processors (b), and the loop transformed into code for sMX (c). Each column is mapped serially, meaning that all the data in that column will be placed on a single processor. As a result, the loop that iterates over the columns (the inner loop) is unchanged.

#### Block Decomposition

A *block* decomposition divides the elements into one group of adjacent elements per processor. Using the previous example, the rows of array a are divided between the four processors, as shown in Figure 52(b). Because the rows have been divided across the processors, the outer loop in Figure 52(c) has been modified so that each processor iterates only over the local portion of each row.

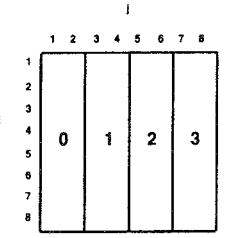
Figure 52(d) shows the decomposition if block scheduling is applied across both the horizontal and vertical dimensions.

A major difference between compilation for shared- and distributed-memory machines is that shared-memory machines provide a global name space, so that any particular array element can be named by any processor using the same subscripts as in the sequential code. On the other hand, on a distributed-memory machine, arrays must usually be divided across the processors, each of which has its own private address space.

As a result, naming a nonlocal array element requires a mapping function from the original, global name space to a processor number and local array indices within that processor. The examples shown here are all for shared memory, and therefore sidestep these complexities, which are discussed in Section 7.3. However, the same decomposition principles are used for distributed memory.

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

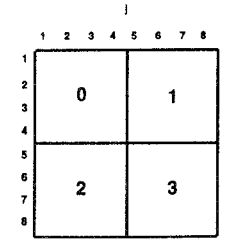
(a) original loop




(b) (*block,serial*) decomposition of a on four processors

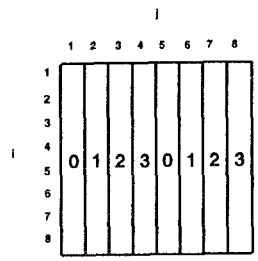
```
call FORK(Pnum)
do j = (n/Pnum)*Pid+1, min((n/Pnum)*(Pid+1), n)
  do i = 1, n
    a[i,j] = a[i,j] + c
  end do
end do
call JOIN()
```

(c) corresponding (*block, serial*) scheduling of the loop, using block size n/Pnum




(d) (*block,block*) decomposition of a on four processors

**Figure 52.** Serial and block decompositions of an array on the shared-memory multiprocessor sMX.




(a) (*cyclic,serial*) decomposition of a on four processors

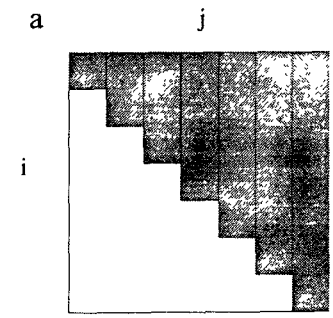
```
call FORK(Pnum)
do j = Pid+1, n, Pnum
  do i = 1, n
    a[i,j] = a[i,j] + c
  end do
end do
call JOIN()
```

(b) corresponding (*cyclic, serial*) scheduling of loop

**Figure 53.** Cyclic decomposition on sMX.

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

(a) loop




(b) elements of a read by the loop

**Figure 54.** Triangular iteration space.

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.

### Cyclic Decomposition

A *cyclic* decomposition assigns successive elements to successive processors. A cyclic allocation of columns to processors for the sequential code in Figure 52(a) is shown in Figure 53(a). Figure 53(b) shows the transformed code for sMX.

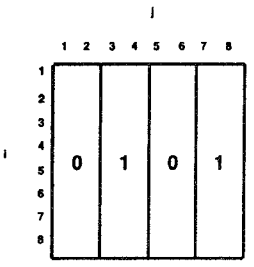
Cyclic decomposition has the opposite effect of blocking: it has poor locality for neighbor-based communication, but spreads load more evenly. Cyclic decomposition works well for loops like the one in Figure 54, as long as the array is large enough so that each processor handles many columns. A common algorithm that results in this type of load balance is LU factorization of matrices. A block decomposition would work poorly because the iterations most costly to execute would be clustered together and computed by a single processor. With a cyclic decomposition, the expensive iterations are spread across the processors.

### 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 55(a) shows an array whose columns have been allocated to two processors in a block-cyclic fashion, and Figure 55(b) gives the code to implement the mapping on sMX. Block-cyclic is a compromise between the locality of block decomposition and the load balancing of cyclic decomposition.

### Irregular Computations

Some loops, like the one shown in Figure 56, have highly irregular execution behavior. When the variance of execution times is high, none of the static regular decomposition strategies may be sufficient to yield good performance. A variety of dynamic algorithms make




(a) (*block-cyclic,serial*) decomposition of a on two processors

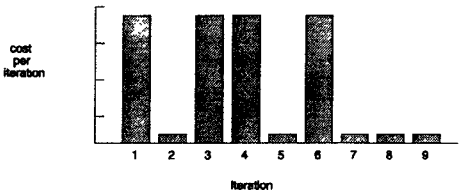
```
call FORK(Pnum)
do k = 1, n, 2*Pnum
  do j = k + 2*Pid, k + 2*Pid + 1
    do i = 1, n
      a[i,j] = a[i,j] + c
    end do
  end do
end do
call JOIN()
```

(b) corresponding (*block-cyclic, serial*) scheduling of loop, using block size 2

**Figure 55.** Block-cyclic decomposition on sMX.

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

(a) example loop




(b) iteration versus cost for the above loop

**Figure 56.** Irregular execution behavior.

scheduling decisions at run time based on the historical behavior of the computation [Lucco 1992; Polychronopoulos and Kuck 1987; Tang and Yew 1990].

### 7.1.2 Automatic Decomposition and Alignment

The decomposition of an array determines how the elements are distributed across a set of processors; in a block decomposition, for example, the array might be divided into contiguous $10 \times 10$ subarrays. The *alignment* of the array establishes the specific elements that are in each of those subarrays.

Languages like HPF Fortran rely on the programmer to declare how arrays are to be aligned and what style of decomposition to use; the language includes annotations like BLOCK and CYCLIC. A variety of research projects have investigated whether the strategy can be determined automatically without programmer assistance. Programmers can find it difficult to choose a good set of declarations, particularly since the optimal decomposition can change during the course of program execution.

The general strategy for automating decomposition and alignment is to express the behavior of the program in a representation that either captures communication explicitly or allows it to be computed. The compiler applies operations that reflect different decomposition and alignment choices; the goal is to maximize parallelism and minimize communication. The approaches include the *Alignment-Distribution Graph* [Chatterjee et al. 1993a; 1993b], the *Component Affinity Graph* [Li and Chen 1990; 1991], constraints [Gupta 1992], and affine transformations [Anderson and Lam 1993].

In addition to presenting an algorithm to find the optimal mapping of arrays to processors for one- and two-dimensional arrays, Mace [1985; 1987] shows that finding optimal decompositions is an NP-complete problem.

### 7.1.3 Scalar Privatization

The goal behind privatization is to increase the amount of parallelism and to avoid unnecessary communication. When a scalar is used within a loop solely as a scratch variable, each processor can be given a private copy so the use of the scalar need not involve any communication. The transformation is safe if there are no loop-carried dependences involv-

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

**Figure 57.** Scalar value suitable for privatization.

ing the scalar; before the scalar is used in the loop body, its value is always updated within the same iteration.

Figure 57 shows a loop that could benefit from privatization. The scalar value c is simply a scratch value; if it is privatized, the loop is revealed to have independent iterations. If the arrays are allocated to separate processors before the loop is executed, no further communication is necessary. In this very simple example the variable could also have been eliminated entirely, but in more complex loops temporary variables are repeatedly updated and referenced.

Scalar privatization is analogous to scalar expansion (see Section 6.5.2); in both cases, spurious dependences are eliminated by replicating the variable. The transformation is widely incorporated into parallel compilers [Allen et al. 1988b; Cytron and Ferrante 1987].

#### 7.1.4 Array Privatization

Array privatization is similar to scalar privatization; each processor is given its own copy of the entire array. The effect of privatizing an array is to reduce communication requirements and to expose additional parallelism by removing unnecessary dependences caused by storage reuse.

Parallelization studies of real application programs have found that privatization is necessary for high performance [Eigenmann et al. 1991; Singh and Hennessy 1991]. Despite its importance, verifying that privatization of an array is legal is much more difficult than the equivalent test on a scalar value.

Traditionally, data-flow analysis [Muchnick and Jones 1981] treats an entire array as a single value. This lack of precision prevents most loop optimizations discussed in this survey and led to the development of dependence analysis. Feautrier [1988; 1991] extends dependence analysis to support *array expansion*, essentially the same optimization as privatization. The analysis is expensive because it requires the use of parametric integer programming. Other privatization research has extended data-flow analysis to consider the behavior of individual array elements [Li 1992; Maydan et al. 1993; Tu and Padua 1993].

In addition to determining whether privatization is legal, the compiler must also check whether the values that are left in an array are used by subsequent computations. If so, after the array is privatized the compiler must add code to preserve those final values by performing a *copy-out* operation from the private copies to a global array.

#### 7.1.5 Cache Alignment

On shared-memory processors like sMX, when one processor modifies storage, the cache line in which it resides is flushed by any other processors holding a copy. The flush is necessary to ensure proper synchronization when multiple processors are updating a single shared variable. If several processors are continually updating different variables or array elements that reside in the same cache line, however, the constant flushing of cache lines may severely degrade performance. This problem is called *false sharing*.

When false sharing occurs between parallel loop iterations accessing different columns of an array or different structures, false sharing can be addressed by aligning each of the columns or structures to a cache line boundary. False sharing of scalars can be addressed by inserting padding between them so that each shared scalar resides in its own cache line (padding is discussed in Section 6.5.1). A further optimization is to place shared variables and their corresponding lock variables in the same cache line, which will cause the lock acquisition to act as a prefetch of the data [Torrellas et al. 1994].

### 7.2 Exposing Coarse-Grained Parallelism

Transferring data can be an expensive operation on a machine with asynchronously executing processors. One way to avoid the inefficiency of frequent communication is to divide the program into subcomputations that will execute for an extended period of time without generating message traffic. The following optimizations are designed to identify such computations or to help the compiler create them.

#### *7.2.1 Procedure Call Parallelization*

One potential source of coarse-grained parallelism in imperative languages is the procedure call. In order to determine whether a call can be performed as an independent task, the compiler must use interprocedural analysis to examine its behavior.

Because of its importance to all forms of optimization, interprocedural analysis has received a great deal of attention from compiler researchers. One approach is to propagate data-flow and dependence information through call sites [Banning 1979; Burke and Cytron 1986; Callahan et al. 1986; Myers 1981]. Another is to summarize the array access behavior of a procedure and use that summary to evaluate whether a call to it can be executed in parallel [Balasundaram 1990; Callahan and Kennedy 1988a; Li and Yew 1988; Triolet et al. 1986].

#### *7.2.2 Split*

A more comprehensive approach to program decomposition summarizes the data usage behavior of an arbitrary block of code in a *symbolic descriptor* [Graham et al. 1993]. As in the previous section, the summary identifies independence between computations—between procedure calls or between different loop nests. Additionally, the descriptor is used in applying the *split* transformation.

Split is unusual in that it is not applied in isolation to a computation like a loop nest. Instead, split considers the relationship of pairs of computations. Two loop nests, for example, may have dependences between them that prevent the compiler from executing both simultaneously on different processors. However, it is often the case that only some of the iterations conflict. Split uses memory summarization to identify the iterations in one loop nest that are independent of the other one, placing those iterations in a separate loop. Applying split to an application exposes additional sources of concurrency and pipelining.

#### *7.2.3 Graph Partitioning*

Data-flow languages [Ackerman 1982; McGraw 1985; Nikhil 1988] expose parallelism explicitly. The program is converted into a graph that represents basic computations as nodes and the movement of data as arcs between nodes. Data-flow graphs can also be used as an internal representation to express the parallel structure of programs written in imperative languages.

One of the major difficulties with data-flow languages is that they expose parallelism at the level of individual arithmetic operations. As it is impractical to use software to schedule such a small amount of work, early projects focused on developing architectures that embed data-flow execution policies into hardware [Arvind and Culler 1986; Arvind et al. 1980; Dennis 1980]. Such machines have not proven to be successful commercially, so researchers began to develop techniques for compiling data-flow languages on conventional architectures. The most common approach is to interpret the data-flow graph dynamically, executing a node representing a computation when all of its operands are available. To reduce scheduling overhead, the data-flow graph is generally transformed by gathering many simple computations into larger blocks that are executed atomically [Anderson and Hudak 1990; Hudak and Goldberg 1985; Mirchandaney et al. 1988; Sarkar 1989].

### 7.3 Computation Partitioning

In addition to decomposing data as discussed in Section 7.1, parallel compilers

must allocate the computations of a program to different processors. Each processor will generally execute a restricted set of iterations from a particular loop. The transformations in this section seek to enforce correct execution of the program without introducing unnecessary overhead.

### 7.3.1 Guard Introduction

After a loop is parallelized, not all the processors will compute the same iterations or send the same slices of data. The compiler can ensure that the correct computations are performed by introducing conditional statements (known as *guards*) into the program [Callahan and Kennedy 1988b].

Figure 58(a) shows an example of a sequential loop that we will transform for parallel execution. Figure 58(b) shows the parallel version of the loop. The code computes upper and lower bounds that the guards use to prevent computation on array elements that are not stored locally within the processor. The bounds depend on the size of the array, the number of processors, and the Pid of the processor.

We have made a number of simplifying assumptions: the value of n is an even multiple of Pnum, and the arrays are already distributed across the processors with a block decomposition. We have also chosen an example that does not require interprocessor communication. To parallelize the loops encountered in practice, compilers must introduce communication statements and much more complex guard and induction variable expressions.

dMX does not have a global namespace, so each processor's instance of an array is separate. In Figure 58(b), the compiler could take advantage of the isolation of each processor to remap the array elements. Instead of the original array of size n, each processor could declare a smaller array of n / Pnum elements and modify the loop to iterate from 1 to n / Pnum. Although the remapping would simplify the loop bounds, we preserve the original element indices to maintain a closer relationship between the sequential code and its transformed version.

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

(a) original loop

```
LBA = (n/Pnum)*Pid + 1
UBA = (n/Pnum)*(Pid + 1)
LBB = (n/Pnum)*Pid + 1
UBB = (n/Pnum)*(Pid + 1)
do i = 1, n
  if (LBA <= i .and. i <= UBA)
    a[i] = a[i] + c
  if (LBB <= i .and. i <= UBB)
    b[i] = b[i] + c
end do
```

(b) parallelized loop with guards introduced

```
LB = (n/Pnum)*Pid + 1
UB = (n/Pnum)*(Pid + 1)
do i = 1, n
  if (LB <= i .and. i <= UB) then
    a[i] = a[i] + c
    b[i] = b[i] + c
  end if
end do
```

(c) after guard combination

```
LB = (n/Pnum)*Pid + 1
UB = (n/Pnum)*(Pid + 1)
do i = LB, UB
  a[i] = a[i] + c
  b[i] = b[i] + c
end do
```

(d) after bounds reduction

**Figure 58.** Computation partitioning.

### 7.3.2 Redundant Guard Elimination

In the same way that the compiler can use code hoisting to optimize computation, it can reduce the number of guards

by hoisting them to the earliest point where they can be correctly computed. Hoisting often reveals that identical guards have been introduced and that all but one can be removed. In Figure 58(b), the two guard expressions are identical because the arrays have the same decomposition. When the guards are hoisted to the beginning of the loop, the compiler can eliminate the redundant guard. The second pair of bound computations becomes useless and is also removed. The final result is shown in Figure 58(c).

#### 7.3.3 Bounds Reduction

In Figure 58(c), the guards control which iterations of the loop perform computation. Since the desired set of iterations is a contiguous range, the compiler can achieve the same effect by changing the induction expressions to reduce the loop bounds [Koelbel 1990]. Figure 58(d) shows the result of the transformation.

### 7.4 Communication Optimization

An important part of compilation for distributed-memory machines is the analysis of an application's communication needs and introduction of explicit message-passing operations into it. Before a statement is executed on a given processor, any data it relies on that is not available locally must be sent. A simple-minded approach is to generate a message for each nonlocal data item referenced by the statement, but the resulting overhead will usually be unacceptably high. Compiler designers have developed a set of optimizations that reduce the time required for communication.

The same fundamental issues arise in communication optimization as in optimizations for memory access (described in Section 6.5): maximizing reuse, minimizing the working set, and making use of available parallelism in the communication system. However, the problems are magnified because while the ratio of one memory access to one computation on S-DLX is 16 : 1, the ratio of the number of cycles required to send one word to the number of cycles required for a single operation on dMX is about 500 : 1.

Like vectors on vector processors, communication operations on a distributed-memory machine are characterized by a *startup time*, $t_s$, and a *per-element cost*, $t_b$, the time required to send one byte once the message has been initiated. On dMX, $t_s = 10\mu s$ and $t_b = 100ns$, so sending one 10-byte message costs $11\mu s$ while sending two 5-byte messages costs $21\mu s$. To avoid paying the startup cost unnecessarily, the optimizations in this section combine data from multiple messages and send them in a single operation.

#### 7.4.1 Message Vectorization

Analysis can often determine the set of data items transferred in a loop. Rather than sending each element of an array in an individual message, the compiler can group many of them together and send them in a single block transfer. Because this is analogous to the way a vector processor interacts with memory, the optimization is called message vectorization [Balasundaram et al. 1990; Gerndt 1990; Zima et al. 1988].

Figure 59(a) shows a sample loop that multiplies each element of array a by the mirror element of array b. Figure 59(b) is an inefficient parallel version for processors 0 and 1 on dMX. To simplify the code, we assume that the arrays have already been allocated to the processors using a block decomposition: the lower half of each array is on processor 0 and the upper half on processor 1.

Each processor begins by computing the lower and upper bounds of the range that it is responsible for. During each iteration, it sends the element of b that the other processor will need and waits to receive the corresponding message. Note that Fortran's call-by-reference semantics convert the array reference implicitly into the address of the corresponding element, which is then used by the low-level communication routine to extract the bytes to be sent or received.

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

(a) original loop

```
LB = Pid * (n/2) + 1
UB = LB + (n/2)
otherPid = 1 - Pid
do i = LB, UB
  call SEND(b[i], 4, otherPid)
  call RECEIVE(b[n+1-i], 4)
  a[i] = a[i] + b[n+1-i]
end do
```

(b) parallel loop

```
LB = Pid * (n/2) + 1
UB = LB + (n/2)
otherPid = 1 - Pid
otherLB = otherPid * (n/2) + 1
otherUB = otherLB + (n/2)
call SEND(b[LB], (n/2)*4, otherPid)
call RECEIVE(b[otherLB], (n/2)*4, otherPid)
do i = LB, UB
  a[i] = a[i] + b[n+1-i]
end do
```

(c) parallel loop with vectorized messages

```
do j = LB, UB, 256
  call SEND(b[j], 256*4, otherPid)
  call RECEIVE(b[otherLB+(j-LB)],
         256*4, otherPid)
  do i = j, j+255
    a[i] = a[i] + b[n+1-i]
  end do
end do
```

(d) after strip mining messages (assuming array size is a multiple of 256)

**Figure 59.** Message vectorization.

When the message arrives, the iteration proceeds.

Figure 59(c) is a much more efficient version that handles all communication in a single message before the loop begins executing. Each processor computes the upper and lower bound for both itself and for the other processor so as to place the incoming elements of b properly.

Message-passing libraries and network hardware often perform poorly when their internal buffer sizes are exceeded. The compiler may be able to perform additional communication optimization by using strip mining to reduce the message length as shown in Figure 59(d).

#### 7.4.2 Message Coalescing

Once message vectorization has been performed, the compiler can further reduce the frequency of communication by grouping messages together that send overlapping or adjacent data. The Fortran D compiler [Tseng 1993] uses Regular Sections [Callahan and Kennedy 1988a], an array summarization strategy, to describe the array slice in each message. When two slices being sent to the same processor overlap or cover contiguous ranges of the array, the associated messages are combined.

#### 7.4.3 Message Aggregation

Sending a message is generally much more expensive than performing a block copy locally on a processor. Therefore it is worthwhile to aggregate messages being sent to the same processor even if the data they contain is unrelated [Tseng 1993]. A simple aggregation strategy is to identify all the messages directed at the same target processor and copy the various strips of data into a single buffer. The target processor performs the reverse operation.

#### 7.4.4 Collective Communication

Many parallel architectures and message-passing libraries offer special-purpose communication primitives such as broadcast, hardware reduction, and scatter-gather. Compilers can improve performance by recognizing opportunities to exploit these operations, which are often highly efficient. The idea is analogous to idiom and reduction recognition on sequential machines. Li and Chen [1991] present compiler techniques that rely on pattern matching to identify opportunities for using collective communication.

#### *7.4.5 Message Pipelining*

Another important optimization is to pipeline parallel computations by overlapping communication and computation. Studies have demonstrated that many applications perform very poorly without pipelining [Rogers 1991]. Many message-passing systems allow the processor to continue executing instructions while a message is being sent. Some support fully nonblocking send and receive operations. In either case, the compiler has the opportunity to arrange for useful computation to be performed while the network is delivering messages.

A variety of algorithms have been developed to discover opportunities for pipelining and to move message transfer operations so as to maximize the amount of resulting overlap [Rogers 1991; Koelbel and Mehrotra 1991; Tseng 1993].

#### *7.4.6 Redundant Communication Elimination*

To avoid sending messages wherever possible, the compiler can perform a variety of transformations to eliminate redundant communication. Many of the optimizations covered earlier can also be used on messages. If a message is sent within the body of a loop but the data does not change from one iteration to the next, the SEND can be hoisted out of the loop. When two messages contain the same data, only one need be sent.

Messages offer further opportunities for optimization. If the contents of a message are subsumed by a previous communication, the message need not be sent; this situation is often created when SENDs are hoisted in order to maximize pipelining opportunities. If a message contains data, some of which has already been sent, the overlap can be removed to reduce the amount of data transferred. Another possibility is that a message is being sent to a collection of processors, some of which previously received the data. The list of recipients can be pruned, reducing the amount of communication traffic. These optimizations are used by the PTRAN II compiler [Gupta et al. 1993] to reduce overall message traffic.

### 7.5 SIMD Transformations

SIMD architectures exhibit much more regular behavior than MIMD machines, eliminating many problematic synchronization issues. In addition to the alignment and decomposition strategies for distributed-memory systems (see Section 7.1.2), the regularity of SIMD interconnection networks offers additional opportunities for optimization. The compiler can use very accurate cost models to estimate the performance of a particular layout.

Early SIMD compilation work targeted IVTRAN [Millstein and Muntz 1975], a Fortran dialect for the Illiac IV that provided layout and alignment declarations. The compiler provided a parallelizing module called the Paralyzer [Presberg and Johnson 1975] that used an early form of dependence analysis to identify independent loops and applied linear transformations to optimize communication.

The Connection Machine Convolution Compiler [Bromley et al. 1991] targets the topology of the SIMD architecture explicitly with a pattern-matching strategy. The compiler focuses on computations that update array elements based on their neighbors. It finds the pattern of neighbors that are needed to compute a given element, called the *stencil*. The cross, a common stencil, represents a computation that updates each array element using the value of its neighbors to the north, east, west, and south. Stencils are aggregated into larger patterns, or *multistencils*, using optimizations analogous to loop unrolling and strip mining. The multistencils are mapped to the hardware so as to minimize communication.

SIMD compilers developed at Compass [Knobe and Natarajan 1993; Knobe et al. 1988; 1990] construct a preference graph based on the computations being performed. The graph represents alignment requests that would yield the least communication overhead. The compiler satisfies every request when possible, using a greedy algorithm to find a solution when preferences are in conflict.

Wholey [1992] begins by computing a detailed cost function for the computation. The function is the input to a hill-climbing algorithm that searches the space of possible allocations of data to processors. It begins with two processors, then four, and so forth; each time the number of processors is doubled, the algorithm uses the most efficient allocation from the previous stage as a starting point.

In addition to general-purpose mapping algorithms, the compiler can use specific transformations like loop flattening [von Hanxleden and Kennedy 1992] to avoid idle time due to nonuniform computations.

### 7.6 VLIW Transformations

The Very Long Instruction Word (VLIW) architecture is another strategy for executing instructions in parallel. A VLIW processor [Colwell et al. 1988; Fisher et al. 1984; Floating Point Systems 1979; Rau et al. 1989] 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.

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 VLIW architectures. They rely on the compiler to expose the parallel operations explicitly and schedule them at compile time. Thus the task of generating object code incorporates high-level resource allocation and scheduling decisions that occur only between processors on a more conventional parallel architecture.

Trace scheduling [Ellis 1986; Fisher 1981; Fisher et al. 1984] is an analysis strategy that was developed for VLIW architectures. Because VLIW machines require more parallelism than is typically available within a basic block, compilers for these machines must look through branches to find additional instructions to execute. The multiple paths of control require the introduction of *speculative execution*—computing some results that may turn out to be unused (ideally at no additional cost, by functional units that would otherwise have gone unused).

The speculation can be handled dynamically by the hardware [Sohi and Vajapayem 1990], but that complicates the design significantly. The trace-scheduling alternative is to have the compiler do it by identifying paths (or *traces*) through the control-flow graph that might be taken. The compiler guesses which way the branches are likely to go and hoists code above them accordingly. 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 that 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 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 valid. To model the effect of applying a sequence of transformations, the corresponding matrices are simply multiplied.

Figure 60 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 is

$$RD = P_1 = \begin{bmatrix} -1 \\ 0 \end{bmatrix}.$$

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.

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

**Figure 60.** Unimodular transformation example.

Any loop nest whose dependences are all representable by a distance vector can be transformed via skewing 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 that includes unimodular transformations, tiling, 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 that 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 in-

terconnect. 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 compilers. 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, yielding results that 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 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 naive 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

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

Knuth [1971] carried out an early and influential study of Fortran programs. He studied 440 programs comprising 250,000 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.

Shen et al. [1990] examined the subscript expressions that appeared in a set of Fortran mathematical libraries and scientific applications. They applied various dependence tests to these expressions. The results demonstrate how the 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 exam-

ine a single array dimension at a time (coupled subscripts are discussed in Section 5.4).

A study by Petersen and Padua [1993] evaluates approximate dependence tests against the omega test [Pugh 1992], finding that the latter does not expose substantially more parallelism in the Perfect Club benchmarks.

### 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; and
- 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 that required restructuring before they could be computed with vector operations.

Carr [1993] applies a set of loop transformations (unroll-and-jam, loop interchange, tiling, and scalar replacement) to various scientific applications to find a strategy that optimizes cache utilization.

Wolf [1992] uses inner loops from the Perfect Club and from one of the SPEC benchmarks to compare the performance of hand-optimized to compiler-optimized code. The focus is on improving locality to reduce traffic between memory and the cache.

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 [Padua and Petersen 1992].

Lee et 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.

Some preliminary results have also been reported for the Fortran D compiler [Hiranandani et al. 1993] on the Intel iPSC/860 and Thinking Machines CM-5 architectures.

### 9.4 Instruction-Level 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 when an unlimited number of functional units is assumed. Some of these studies are discussed and evaluated in detail by 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 or useless 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. They found that there was significant additional parallelism available, but that exploiting it would require a great deal of hardware. For the programs studied, on the order of $2^{16}$ arithmetic units would be necessary to expose 10-way parallelism.

A subsequent study by Nicolau and Fisher [1984] sought to measure the parallelism that could be exploited by a VLIW architecture using aggressive compiler analysis. The strategy was to compile a set of scientific programs into an intermediate form, simulate execution, and apply a scheduling algorithm retroactively that used branch and address information revealed by the simulation. The study revealed significant amounts of parallelism—a factor of tens or hundreds.

Wall [1991] took trace data from a number of applications and measured the amount of parallelism under a variety of models. His study considered the effect of architectural features such as varying numbers of registers, branch prediction, and jump prediction. It also considered the effects of alias analysis in the compiler. The results varied widely, yielding as much as 60-way parallelism under the most optimistic assumptions. The average parallelism on an ambitious combination of architecture and compiler support was 7.

Butler et al. [1991] used an optimizing compiler on a group of programs from the SPEC89 suite. The resulting code was executed on a variety of simulated machines. The range includes unrealistically omniscient architectures that combine data-flow strategies for scheduling with perfect branch prediciton. They also restricted models with limited hardware. Under the most optimistic assumptions, they were able to execute 17 instructions per cycle; more practical models yielded between 2.0 and 5.8 instructions per cycle. Without looking past branches, few applications could execute more than 2 instructions per cycle.

Lam and Wilson [1992] argue that one major reason why studies of instruction-level parallelism differ markedly in their results is branch prediction. The studies that perform speculative execution based on prediction yield much less parallelism than the ones that have an oracle with perfect knowledge about branches. The authors speculate that executing branches locally will not yield large degrees of parallelism; they conduct experiments on a number of applications and find similar results to other studies (between 4.2 and 9.2). They argue that compiler optimizations that consider the global flow of control are essential.

Finally, Larus [1993] takes a different approach than the others, focusing strictly on the parallelism available in loops. He examines six codes (three integer programs from SPEC89 and three scientific applications), collects a trace annotated with semantic information, and runs it through a simulator that assumes a uniform shared-memory architecture with infinite processors. Two of the scientific codes, the ones that were primarily array manipulators, showed a potential speedup in the range of 65–250. The potential speedup of the other codes was 1.7–3, suggesting that the techniques developed for parallelizing scientific codes will not work well in parallelizing other types of programs.

## CONCLUSION

We have described a large number of transformations that can be used to im-

prove program performance on sequential and parallel machines. Numerous studies have demonstrated that these transformations, when properly applied, are sufficient to yield high performance.

However, current optimizing compilers lack an organizing principle that allows them to choose how and when the transformations should be applied. Finding a framework that unifies many transformations is an active area of research. Such a framework could simplify the problem of searching the space of possible transformations, making it easier and therefore cheaper to build high-quality optimizing compilers. A key part of this problem is the development of a cost function that allows the compiler to evaluate the effectiveness of a particular set of transformations.

Despite the absence of a strategy for unifying transformations, compilers have proven to be quite successful in targeting sequential and superscalar architectures. On parallel architectures, however, most high-performance applications currently rely on the programmer rather than the compiler to manage parallelism. Compilers face a large space of possible transformations to apply, and parallel machines exact a very high cost of failure when optimization algorithms do not discover a good reorganization strategy for the application.

Because efforts to parallelize traditional languages automatically have met with little success, the focus of research has shifted to compiling languages in which the programmer shares the responsibility for exposing parallelism and choosing a data layout.

Another developing area is the optimization of nonscientific applications. Like most researchers working on high-performance architectures, compiler writers have focused on code that relies on loop-based manipulation of arrays. Other kinds of programs, particularly those written in an object-oriented programming style, rely heavily on pointers. Optimization of pointer manipulation and of object-oriented languages is emerging as another focus of compiler research.

## APPENDIX: 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 [Hewlett-Packard 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. 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 multicycle 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 61 is a block diagram of the architecture and its primary datapaths.

Table A.1 describes the instruction set. All instructions are 32 bits and must be word-aligned. The immediate operand

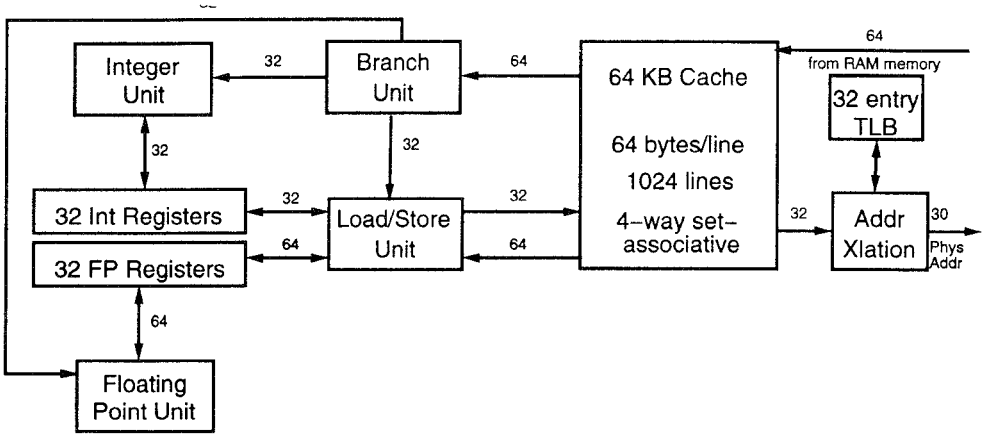



**Figure 61.** S-DLX functional diagram.

**Table A.1.** The S-DLX Instruction Set

| 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 | $R1_{16\ 31} \leftarrow 666$ | |
| `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` |

field is 16 bits. The address for 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 that 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

**Table A.2.** V-DLX Vector Instructions

| 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` | |

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.

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.

Floating-point divides take 14 cycles and all other 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. Integer multiplication takes two 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 version of DLX extended to include vector support. This new architecture, V-DLX, 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 discuss only 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. V-DLX issues only one scalar or one vector instruction per cycle, but nonconflicting 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. By software convention, the VLR is normally set to 64, except when handling the last few iterations of a loop.

The vector operations are described in Table A.2. 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 *startup time*). After the pipeline is full, one result is computed per clock cycle. In

**Table A.3.** Startup Times in Cycles on V-DLX

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

the meantime, the processor can continue to execute other instructions. Table A.3 gives the startup times for the vector operations in V-DLX. These times should not be compared directly to the cycle times 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. V-DLX has eight 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 6.5.1).

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 issue is how much of the underlying machine architecture to expose to the programmer. At one extreme, the programmer can make 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 or library routines. System call semantics are usually not defined by the language, making it difficult for the compiler to optimize them automatically.

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].

For both shared- and distributed-memory multiprocessors, we assume that the programming environment initializes the variables Pnum to be the total number of processors and Pid to be the number of the local processor. Processors are numbered starting with 0.

### *A.3.1 Shared-Memory DLX Multiprocessor*

sMX is our prototypical shared-memory parallel architecture. It consists of 16 S-DLX processors and 512MB of shared main memory. The processors and memory system are connected together by a bus. 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 main memory controller. 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;

**Table A.4.** Memory reference latency in sMX

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

Eggers and Katz 1989], but the details are not relevant to this survey. From the compiler writer's perspective, the key issue is the time it takes to make a memory reference. Table A.4 summarizes the latency of each kind of memory reference in sMX.

Typically, shared-memory multiprocessors implement a number of synchronization operations. FORK(n) starts identical copies of 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( ) resynchronizes 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, which is a synchronization point in the program where each processor waits until all processors have arrived at that point.

### *A.3.2 Distributed-Memory DLX Multiprocessor*

dMX is our hypothetical distributed-memory multiprocessor. The machine consists of 64 S-DLX processors (numbered 0 through 63) connected in an $8 \times 8$ mesh. The network bandwidth of each link in the mesh is 10 MB 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 1 microsecond per 10 bytes of message (we have simplified the machine by declaring that all remote processors have the same latency). Communication is supported by a message library that provides the following calls:

- 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/10 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.

## ACKNOWLEDGMENTS

We thank Michael Burke, Manish Gupta, Monica Lam, and Vivek Sarkar for their assistance with various parts of this survey. We thank John Boyland, James Demmel, Alex Dupuy, John Hauser, James Larus, Dick Muntz, Ken Stanley, the members of the IBM Advanced Development Technology Institute, and the anonymous referees for their valuable comments on earlier drafts.

## REFERENCES

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

ABU-SUFAH, W 1979. Improving the performance of virtual memory computers. Ph D., thesis, Tech. Rep. 78-945, Univ. of Illinois at Urbana-Champaign.

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.

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, Mass.

AIKEN, A. AND NICOLAU, A. 1988a Optimal loop parallelization. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (Atlanta, Ga., June). *SIGPLAN Not. 23*, 7 (July), 308–317.

AIKEN, A. AND NICOLAU, A. 1988b. Perfect pipelining: A new loop parallelization technique. In *Proceedings of the 2nd European Symposium on Programming* Lecture Notes in Computer Science, vol. 300. Springer-Verlag, Berlin, 221–235.

ALLEN, J. R. 1983. Dependence analysis for subscripted variables and its application to program transformations. Ph.D. thesis, Computer Science Dept., Rice University, Houston, Tex

ALLEN, F. E. 1969. Program optimization. In *Annual Review in Automatic Programming 5*. International Tracts in Computer Science and Technology and their Application, vol. 13. Pergamon Press, Oxford, England, 239–307.

ALLEN, F. E. AND COCKE, J. 1971. A catalogue of optimizing transformations. In *Design and Optimization of Compilers*, R Rustin, Ed. Prentice-Hall, Englewood Cliffs, N.J., 1–30.

ALLEN, J. R. AND KENNEDY, K. 1987. Automatic translation of Fortran programs to vector form. *ACM Trans. Program Lang. Syst. 9*, 4 (Oct.), 491–542.

ALLEN, J. R. AND KENNEDY, K. 1984. Automatic loop interchange. In *Proceedings of the SIGPLAN Symposium on Compiler Construction* (Montreal, Quebec, June). *SIGPLAN Not. 19*, 6, 233–246.

ALLEN, F. E., BURKE, M., CHARLES, P., CYTRON, R., AND FERRANTE, J. 1988a. An overview of the PTRAN analysis system for multiprocessing. *J. Parall. 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, 207–215.

ALLEN, F. E., COCKE, J., AND KENNEDY, K. 1981. Reduction of operator strength. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnik and N. D. Jones, Eds., Prentice-Hall, Englewood Cliffs, N.J., 79–101.

ALLEN, J R., KENNEDY, K., PORTERFIELD, C., AND WARREN, J. 1983. Conversion of control dependence to data dependence In *Conference Record of the 10th ACM Symposium on Principles of Programming Languages* (Austin, Tex., Jan.). ACM Press, New York, 177–189.

ALPERT, D. AND AVNON, D. 1993 Architecture of the Pentium microprocessor *IEEE Micro 13*, 3 (June), 11–21

AMERICAN NATIONAL STANDARDS INSTITUTE. 1987. An American National standard, IEEE standard for binary floating-point arithmetic *SIGPLAN Not. 22*, 2 (Feb.), 9–25.

ANDERSON, J. M. AND LAM, M. S. 1993 Global optimizations for parallelism and locality on scalable parallel machines. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, June). *SIGPLAN Not. 28*, 6, 112–125.

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, N.Y., June). *SIGPLAN Not. 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 *Annual Review of Computer Science.* Vol. 1. J. F. Traub, B. J. Grosz, B. W. Lampson, and N. J. Nilsson, Eds. Annual Reviews, Palo Alto, Calif, 225–253.

ARVIND, KATHAIL, V., AND PINGALI, K. 1980. A dataflow architecture with tagged tokens. Tech. Rep TM-174, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass.

BACON, D. F., CHOW, J.-H., JU, D. R , MUTHUKUMAR, K , AND SARKAR, V. 1994. A compiler framework for restructuring data declarations to enhance cache and TLB effectiveness. In *Proceedings of CASCON '94* (Toronto, Ontario, Oct.).

BAILEY, D. H. 1992. On the behavior of cache memories with strided data access. Tech. Rep. RNR-92-015, NASA Ames Research Center, Moffett Field, Calif., (May).

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. Supercomput. 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. Parall. Distrib. Comput. 9*, 2 (June), 154–170.

BALASUNDARAM, V., FOX, G., KENNEDY, K., AND KREMER, U. 1990. An interactive environment for data partitioning and distribution. In *Proceedings of the 5th Distributed Memory Computer Conference* (Charleston, South Carolina, Apr.). IEEE Computer Society Press, Los Alamitos, Calif.

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, Nev., Nov.). ACM Press, New York, 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, Color., Aug.). *SIGPLAN Not. 14*, 8, 214–220.

BANERJEE, U. 1991. Unimodular transformations of double loops. In *Advances in Languages and Compilers for Parallel Processing*, A. Nicolau, Ed. Research Monographs in Parallel and Distributed Computing, MIT Press, Cambridge, Mass., Chap. 10.

BANERJEE, U. 1988a. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, Mass.

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

BANERJEE, U. 1979. Speedup of ordinary programs, Ph.D. thesis, Tech. Rep. 79-989, Computer Science Dept., Univ. of Illinois at Urbana-Champaign.

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.

BANNING, J. P. 1979. An efficient way to find the side-effects of procedure calls and the aliases of variables. In *Conference Record of the 6th ACM Symposium on Principles of Programming Languages* (San Antonio, Tex., Jan.). ACM Press, 29–41.

BERNSTEIN, R. 1986. Multiplication by integer constants. *Softw. Pract. Exper. 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. Supercomput. Appl. 3*, 3 (Fall), 5–40.

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

BROMLEY, M., HELLER, S., MCNERNEY, T., AND STEELE, G. L., JR. 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 Not. 26*, 6, 145–156.

BURKE, M. AND CYTRON, R. 1986. Interprocedural dependence analysis and parallelization. In *Proceedings of the SIGPLAN Symposium on Compiler Construction* (Palo Alto, Calif., June). *SIGPLAN Not. 21*, 7 (July), 162–175. Extended version available as IBM Thomas J. Watson Research Center Tech. Rep. RC 11794.

BURNETT, G. J. AND COFFMAN, E. G., JR. 1970. A study of interleaved memory systems. In *Proceedings of the Spring Joint AFIPS Computer Conference*. vol. 36. AFIPS, Montvale, N.J., 467–474.

BURSTALL, R. M. AND DARLINGTON, J. 1977. A transformation system for developing recursive programs. *J. ACM 24*, 1 (Jan.), 44–67.

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). *SIGARCH Comput. Arch. News 19*, 3, 276–286.

CALLAHAN, D. AND KENNEDY, K. 1988a. Analysis of interprocedural side-effects in a parallel programming environment. *J. Parall. Distrib. Comput. 5*, 5 (Oct.), 517–550.

CALLAHAN, D. AND KENNEDY, K. 1988b. Compiling programs for distributed-memory multiprocessors. *J. Supercomput. 2*, 2 (Oct.), 151–169.

CALLAHAN, D., CARR, S., AND KENNEDY, K. 1990. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (White Plains, N.Y., June). *SIGPLAN Not. 25*, 6, 53–65.

CALLAHAN, D., COCKE, J., AND KENNEDY, K. 1988. Estimating interlock and improving balance for pipelined architectures. *J. Parall. Distrib. Comput. 5*, 4 (Aug.), 334–358.

CALLAHAN, D., COOPER, K., KENNEDY, K., AND TORCZON, L. 1986. Interprocedural constant propagation. In *Proceedings of the SIGPLAN Symposium on Compiler Construction* (Palo Alto, Calif., June). *SIGPLAN Not. 21*, 7 (July), 152–161.

CARR, S. 1993. Memory-hierarchy management. Ph.D. thesis, Rice University, Houston, Tex.

CHAITIN, G. J. 1982. Register allocation and spilling via graph coloring. In *Proceedings of the SIGPLAN Symposium on Compiler Construction* (Boston, Mass., June). *SIGPLAN Not 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. *Comput. Lang. 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, Ore., June). *SIGPLAN Not. 24*, 7 (July), 146–160.

CHATTERJEE, S., GILBERT, J. R., AND SCHREIBER, R. 1993a. The alignment-distribution graph. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, vol. 768. 234–252.

CHATTERJEE, S., GILBERT, J. R., SCHREIBER, R., AND TENG, S.-H. 1993b. Automatic array alignment in data-parallel programs. In *Conference Record of the 20th ACM Symposium on Principles of Programming Languages* (Charleston, S. Carolina, Jan.). ACM Press, New York, 16–28.

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.

CHOI, J.-D., BURKE, M., AND CARINI, P. 1993. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the 20th ACM Symposium on Principles of Programming Languages* (Charleston, S. Carolina, Jan.). ACM Press, New York, 232–245.

CHOW, F. C. 1988. Minimizing register usage penalty at procedure calls. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (Atlanta, Ga., June). *SIGPLAN Not. 23*, 7 (July), 85–94.

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

CLARKE, C. D. AND PEYTON-JONES, S. L. 1985. Strictness analysis—a practical approach. In *Functional Programming Languages and Computer Architecture*. Lecture Notes in Computer Science, vol. 201. Springer-Verlag, Berlin, 35–49.

CLINGER, W. D. 1990. How to read floating point numbers accurately. In *Proceedings of the SIGPLAN Conference on Programming Language Desiogn and Implementation* (White Plains, N.Y., June). *SIGPLAN Not. 25*, 6, 92–101.

COCKE, J. 1970. Global common subexpression elimination. In *Proceedings of the ACM Symposium on Compiler Optimization* (July). *SIGPLAN Not. 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, Netherlands, 221–228. Also available as IBM Tech. Rep. RC 8111, Feb. 1980.

COCKE, J. AND SCHWARTZ, J. T. 1970. Programming languages and their compilers (preliminary notes). 2nd ed. Courant Inst. of Mathematical Sciences, New York University, New York.

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.

COOPER, K. D. AND KENNEDY, K. 1989. Fast interprocedural alias analysis. In *Conference Record of the 16th ACM Symposium on Principles of Programming Languages* (Austin, Tex., Jan.). ACM Press, New York, 49–59.

COOPER, K. D., HALL, M. W., AND KENNEDY, K. 1993. A methodology for procedure cloning. *Comput. Lang. 19*, 2 (Apr.), 105–117.

COOPER, K. D., HALL, M. W., AND TORCZON, L. 1992. Unexpected side effects of inline substitution: A case study. *ACM Lett. Program. Lang. Syst 1*, 1 (Mar.), 22–32.

COOPER, K. D., HALL, M. W., AND TORCZON, L. 1991. An experiment with inline substitution. *Softw. Pract. Exper. 21*, 6 (June), 581–601

CRAY RESEARCH 1988 *CFT77 Reference Manual*. Publication SR-0018-C. Cray Research, Inc., Eagan, Minn.

CYTRON, R. 1986. Doacross: Beyond vectorization for multiprocessors. In *Proceedings of the International Conference on Parallel Processing* (St. Charles, Ill., Aug.). IEEE Computer Society, Washington, D.C., 836–844.

CYTRON, R. AND FERRANTE, J. 1987. What's in a name? -or- the value of renaming for parallelism detection and storage allocation. *Proceedings of the International Conference on Parallel Processing* (University Park, Penn., Aug.). Pennsylvania State University Press, University Park, Pa., 19–27.

DANTZIG, G. B. AND EAVES, B. C. 1974. Fourier-Motzkin elimination and sits dual with application to integer programming. In *Combinatorial Programming: Methods and Applications* (Versailles, France, Sept.). B. D. Roy, Ed. D. Reidel, Boston, Mass., 93–102.

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, 37th IEEE Computer Society International Conference* (San Francisco, Calif., Feb.). IEEE Computer Society Press, Los Alamitos, Calif., 305–310.

DONGARRA, J. AND HIND, A. R. 1979. Unrolling loops in Fortran. *Softw. Pract. Exper. 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). *SIGARCH Comput. Arch. News 17*, 3 (June), 2–15.

EIGENMANN, R., HOEFLINGER, J., LI, Z., AND PADUA, D. A. 1991. Experience in the automatic parallelization of four Perfect-benchmark programs. In *Proceedings of the 4th International Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, vol. 589. Springer-Verlag, Berlin, 65–83. Also available as Tech. Rep. 1193, Center for Supercomputing Research and Development.

ELLIS, J. R. 1986. *Bulldog: A Compiler for VLIW Architectures*. ACM Doctoral Dissertation Award. MIT Press, Cambridge, Mass.

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

FEAUTRIER, P. 1991. Dataflow analysis of array and scalar references. *Int. J. Parall. Program. 20*, 1 (Feb.), 23–52.

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

FERRARI, D. 1976. The improvement of program behavior. *Computer 9*, 11 (Nov.), 39–47.

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, Quebec, June). *SIGPLAN Not. 19*, 6, 37–47.

FLOATING POINT SYSTEMS. 1979. *FPS AP-120B Processor Handbook*. Floating Point Systems, Inc., Beaverton, Ore.

FREE SOFTWARE FOUNDATION. 1992. *gcc 2.x Reference Manual*. Free Software Foundation, Cambridge, Mass.

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. Parall. Distrib. Comput. 5*, 5 (Oct.), 587–616.

GERNDT, M. 1990. Updating distributed variables in local computations. *Concurrency Pract. Exper. 2*, 3 (Sept.), 171–193.

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.

GIRKAR, M. AND POLYCHRONOPOULOS, C. D. 1988. Compiling issues for supercomputers. In *Proceedings of Supercomputing '88* (Orlando, Fla., Nov.). IEEE Computer Society, Washington, D.C., 164–173.

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 Not. 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 Not. 28*, 6, 100–111.

GRANLUND, T. AND KENNER, R. 1992. Eliminating branches using a superoptimizer and the GNU compiler. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (San Francisco, Calif., June). *SIGPLAN Not. 27*, 7 (July), 341–352.

GUPTA, M. 1992. Automatic data partitioning on distributed memory multicomputers. Ph.D. thesis, Tech. Rep. UILU-ENG-92-2237, Univ. of Illinois at Urbana-Champaign.

GUPTA, M., MIDKIFF, S., SCHONBERG, E., SWEENEY, P., WANG, K. Y., AND BURKE, M. 1993. PTRAN II: A compiler for High Performance Fortran. In *Proceedings of the 4th International Workshop on Compilers for Parallel Computers* (Delft, Netherlands, Dec.). 479–493.

HALL, M. W., KENNEDY, K., AND MCKINLEY, K. S. 1991. Interprocedural transformations for parallel code generation. In *Proceedings of Supercomputing '91* (Albuquerque, New Mexico, Nov.). IEEE Computer Society Press, os Alamitos, Calif., 424–434.

HARRIS, K. AND HOBBS, S. 1994. VAX Fortran. In *Optimization in Compilers*, F. E. Allen, R. Cytron, B. K. Rosen, and K. Zadeck, Eds. ACM Press, New York, chap. 16. To be published.

HATFIELD, D. J. AND GERALD, J. 1971. Program restructuring for virtual memory. *IBM Syst. J. 10*, 3, 168–192.

HENNESSY, J. L. AND PATTERSON, D. A. 1990. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, Calif.

HEWLETT-PACKARD. 1992. *PA-RISC 1.1 Architecture and Instruction Manual*. 2nd ed. Part Number 09740-90039. Hewlett-Packard, Palo Alto, Calif.

HIGH PERFORMANCE FORTRAN FORUM. 1993. High Performance Fortran language specification, version 1.0. Tech. Rep. CRPC-TR92225, Rice University, Houston, Tex.

HIRANANDANI, S., KENNEDY, K., AND TSENG, C.-W. 1993. Preliminary experiences with the Fortran D compiler. In *Proceedings of Supercomputing '93* (Portland, Ore., Nov.). IEEE Computer Society Press, Los Alamitos, Calif., 338–350.

HIRANANDANI, S., KENNEDY, K., AND TSENG, C.-W. 1992. Compiling Fortran D for MIMD dis-

tributed-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.

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). *SIGARCH Comput. Arch. News 17*, 3 (June), 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. Supercomput. 7*, 1/2 (May), 229–248.

IBM. 1992. *Optimization and Tuning Guide for the XL Fortran and XL C Compilers*. 1st ed. Publication SC09-1545-00, IBM, Armonk, N.Y.

IBM. 1991. *IBM RISC System/6000 NIC Tuning Guide for Fortran and C* Publication GG24-3611-01, IBM, Armonk, N.Y.

IRIGOIN, F. AND TRIOLET, R. 1988. Supernode partitioning. In *Conference Record of the 15th ACM Symposium on Principles of Programming Languages* (San Diego, Calif., Jan.). ACM Press, New York, 319–329.

IVERSON, K. E. 1962. *A Programming Language*. John Wiley and Sons, New York.

KENNEDY, K. AND MCKINLEY, K. S. 1990. Loop distribution with arbitrary control flow. In *Proceedings of Supercomputing '90* (New York, N.Y., Nov.). IEEE Computer Society Press, Los Alamitos, Calif., 407–416.

KENNEDY, K., MCKINLEY, K. S., AND TSENG, C.-W. 1993. Analysis and transformation in an interactive parallel programming tool. *Concurrency Pract. Exper 5*, 7 (Oct.), 575–602.

KILDALL, G. 1973. A unified approach to global program optimization In *Conference Record of the ACM Symposium on Principles of Programming Languages* (Boston, Mass , Oct.). ACM Press, New York, 194–206.

KNOBE, K. AND NATARAJAN, V. 1993. Automatic data allocation to minimize communication on SIMD machines. *J. Supercomput. 7*, 4 (Dec.), 387–415.

KNOBE, K., LUKAS, J. D., AND STEELE, G. L., JR. 1990. Data optimization: Allocation of arrays to reduce communication on SIMD machines. *J. Parall Distrib. Comput. 8*, 2 (Feb.), 102–118.

KNOBE, K., LUKAS, J D., AND STEELE, G. L., JR 1988. Massively parallel data optimization. In *The 2nd. Symposium on the Frontiers of Massively Parallel Computation* (Fairfax, Va., Oct.). IEEE, Piscataway, N.J., 551–558.

KNUTH, D. E. 1971. An empirical study of Fortran programs. *Softw. Pract. Exper. 1*, 2 (Apr.-June), 105–133.

KOELBEL, C. 1990. Compiling programs for nonshared memory machines. Ph.D. thesis, Purdue University, West Lafayette, Ind.

KOELBEL, C AND MEHROTRA, P. 1991. Compiling global name-space parallel loops for distributed execution. *IEEE Trans. Parall. Distrib. Syst. 2*, 4 (Oct.), 440–451.

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, Calif., June). *SIGPLAN Not. 21*, 7 (July), 219–233.

KUCK, D. J. 1978. *The Structure of Computers and Computations* Vol. 1. John Wiley and Sons, New York.

KUCK, D. J. 1977. A survey of parallel machine organization and programming. *ACM Comput. Surv. 9*, 1 (Mar.), 29–59.

KUCK, D. J. AND STOKES, R. 1982. The Burroughs Scientific Processor (BSP). *IEEE Trans. Comput. C-31*, 5 (May), 363–376.

KUCK, D. J., KUHN, R. H., PADUA, D., LEASURE, B., AND WOLFE, M. 1981. Dependence graphs and compiler optimizations. In *Conference Record of the 8th ACM Symposium on Principles of Programming Languages* (Williamsburg, Va., Jan.). ACM Press, New York, 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, Ga., June). *SIGPLAN Not. 23*, 7 (July), 318–328.

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). *SIGARCH Comput. Arch. News 20*, 2, 46–57

LAM, M. S., ROTHBERG, E. E., AND WOLF, M. E. 1991. The cache performance and optimization of blocked algorithms. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, Calif., Apr.). *SIGPLAN Not. 26*, 4, 63–74.

LAMPORT, L. 1974. The parallel execution of DO loops. *Commun. ACM 17*, 2 (Feb.), 83–93.

LANDI, W., RYDER, B. G., AND ZHANG, S. 1993. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (Albuquerque, New Mexico, June). *SIGPLAN Not. 28*, 6, 56–67.

LARUS, J. R. 1993 Loop-level parallelism in numeric and symbolic programs. *IEEE Trans. Parall. 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. 1992. Array privatization for parallel execution of loops. In *Proceedings of the ACM International Conference on Supercomputing* (Washington, D.C., July). ACM Press, New York, 313–322.

LI, J. AND CHEN, M. 1991. Compiling communication-efficient programs for massively parallel machines. *IEEE Trans. Parall. Distrib. Syst. 2*, 3 (July), 361–376.

LI, J. AND CHEN, M. 1990. Index domain alignment: Minimizing cost of cross-referencing between distributed arrays. In *The 3rd Symposium on the Frontiers of Massively Parallel Computation*, J. Jaja, Ed. IEEE Computer Society Press, Los Alamitos, Calif., 424–433.

LI, Z. AND YEW, P. 1988. Interprocedural analysis for parallel computing. In *Proceedings of the International Conference on Parallel Processing*. F. A. Briggs, Ed. Vol. 2. Pennsylvania State University Press, University Park, Pa., 221–228.

LI, Z., YEW, P., AND ZHU, C. 1990. Data dependence analysis on multi-dimensional array references. *IEEE Trans. Parall. 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, Calif., June). *SIGPLAN Not. 27*, 7 (July), 200–211.

MACE, M. E. 1987. *Memory Storage Patterns in Parallel Processing*. Kluwer Academic Publishers, Norwell, Mass.

MACE, M. E. AND WAGNER, R. A. 1985. Globally optimum selection of memory storage patterns. In *Proceedings of the International Conference on Parallel Processing*, D. DeGrott, Ed. IEEE Computer Society, Washington, D.C., 264–271.

MARKSTEIN, P., MARKSTEIN, V., AND ZADECK, K. 1994. Strength reduction. In *Optimization in Compilers*. ACM Press, New York, Chap. 9. To be published.

MASSALIN, H. 1987. Superoptimizer: A look at the smallest program. In *Proceedings of the 2nd International Conference on Architectural Support for Programming Languages and Operating Systems* (Palo Alto, Calif., Oct.). *SIGPLAN Not. 22*, 10, 122–126.

MAYDAN, D. E., AMARASINGHE, S. P., AND LAM, M. S. 1993. Array data flow analysis and its use in array privatization. In *Conference Record of the 20th ACM Symposium on Principles of Programming Languages* (Charleston, S. Carolina, Jan.). ACM Press, New York, 1–15.

MAYDAN, D. E., HENNESSEY, J. L., AND LAM, M. S. 1991. Efficient and exact data dependence analysis. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, June). *SIGPLAN Not. 26*, 6, 1–14.

MCFARLING, S. 1991. Procedure merging with instruction caches. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (Toronto, Ontario, June). *SIGPLAN Not. 26*, 6, 71–79.

MCGRAW, J. R. 1985. SISAL: Streams and iteration in a single assignment language. Tech. Rep. M-146, Lawrence Livermore National Laboratory, Livermore, Calif.

MCMAHON, F. M. 1986. The Livermore Fortran kernels: A computer test of numerical performance range. Tech. Rep. UCRL-55745, Lawrence Livermore National Laboratory, Livermore, Calif.

MICHIE, D. 1968. "Memo" functions and machine learning. *Nature 218*, 19–22.

MILLSTEIN, R. E. AND MUNTZ, C. A. 1975. The ILLIAC-IV Fortran compiler. In *Programming Languages and Compilers for Parallel and Vector Machines* (New York, N.Y., Mar.). *SIGPLAN Not. 10*, 3, 1–8.

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, 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, N.J.

MURAOKA, Y. 1971. Parallelism exposure and exploitation in programs. Ph.D. thesis, Tech. Rep. 71-424, Univ. of Illinois at Urbana-Champaign.

MYERS, E. W. 1981. A precise interprocedural data flow algorithm. In *Conference Record of the 8th ACM Symposium on Principles of Programming Languages* (Williamsburg, Va., Jan.). ACM Press, New York, 219–230.

NICOLAU, A. 1988. Loop quantization: A generalized loop unwinding technique. *J. Parall. 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. Tech. Rep. 284, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, Mass.

NOBAYASHI, H. AND EOYANG, C. 1989. A comparison study of automatically vectorizing Fortran compilers. In *Proceedings of Supercomputing*

'89 (Reno, Nev., Nov.). ACM Press, New York, 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*. Publication SA23-2619. IBM Corporation, Mechanicsburg, Penn

OED, W. 1992. Cray Y-MP C90: System features and early benchmark results. *Parall. Comput. 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.

PADUA, D. A. AND PETERSEN, P. M. 1992. Evaluation of parallelizing compilers. *Parallel Computing and Transputer Applications*, (Barcelona, Spain, Sept.). CIMNE, 1505–1514. Also available as Center for Supercomputing Research and Development Tech Rep. 1173.

PADUA, D. A. 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.

PETERSEN, P. M. AND PADUA, D. A. 1993. Static and dynamic evaluation of data dependence analysis. In *Proceedings of the ACM International Conference on Supercomputing* (Tokyo, Japan, July). ACM Press, New York, 107–116.

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, N.Y., June). *SIGPLAN Not. 25*, 6, 16–27.

POLYCHRONOPOULOS, C. D. 1988. *Parallel Programming and Compilers*. Kluwer Academic Publishers, Boston, Mass.

POLYCHRONOPOULOS, C. D. 1987a Advanced loop optimizations for parallel computers. In *Proceedings of the 1st International Conference on Supercomputing*. Lecture Notes in Computer Science, vol 297. Springer-Verlag, Berlin, 255–277.

POLYCHRONOPOULOS, C. D. 1987b. Loop coalescing: A compiler transformation for parallel machines. In *Proceedings of the International Conference on Parallel Processing* (University Park, Penn., Aug.). Pennsylvania State University Press, University Park, Pa., 235–242.

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.

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 2. Pennsylvania State University Press, University Park, Pa., 39–48.

PRESBERG, D. L. AND JOHNSON, N. W. 1975 The Paralyzer: IVTRAN's parallelism analyzer and synthesizer. In *Programming Languages and Compilers for Parallel and Vector Machines* (New York, N.Y., Mar.). *SIGPLAN Not. 10*, 3, 9–16.

PUGH, W. 1992. A practical algorithm for exact array dependence analysis *Commun. ACM 35*, 8 (Aug.), 102–115.

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. Supercomput. 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.

REES, J., CLINGER, W., ET AL. 1986 Revised$^3$ report on the algorithmic language Scheme. *SIGPLAN Not 21*, 12 (Dec.), 37–79.

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.

ROGERS, A. M. 1991. Compiling for locality of reference. Ph.D. thesis, Tech. Rep. TR 91-1195, Dept. of Computer Science, Cornell University.

RUSSELL, R. M. 1978. The CRAY-1 computer system. *Commun. ACM 21*, 1 (Jan.), 63–72.

SABOT, G. AND WHOLEY, S. 1993 CMAX: A Fortran translator for the Connection Machine system. In *Proceedings of the ACM International Conference on Supercomputing* (Tokyo, Japan, July). ACM Press, New York.

SARKAR, V. 1989. *Partitioning and Scheduling Parallel Programs for Multiprocessors*. Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Mass.

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, Calif., June). *SIGPLAN Not. 27*, 7 (July), 175–187.

SCHEIFLER, R. W. 1977. An analysis of inline substitution for a structured programming language. *Commun. ACM 20*, 9 (Sept.), 647–654.

SHEN, Z., LI, Z., AND YEW, P -C. 1990. An empirical study of Fortran programs for parallelizing compilers. *IEEE Trans. Parall. Distrib. Syst. 1*, 3 (July), 356–364.

SINGH, J. P. AND HENNESSY, J. L. 1991. An empirical investigation of the effectiveness and limitations of automatic parallelization. In *Proceedings of the International Symposium on Shared Memory Multiprocessing*, (Apr.). 213–240.

SINGH, J. P., WEBER, W.-D., AND GUPTA, A. 1992. SPLASH: Stanford parallel applications for shared memory. *SIGARCH Comput. Arch. News 20*, 1 (Mar.), 5–44. Also available as Stanford Univ. Tech. Rep. CSL-TR-92-526.

SITES, R. L., (ED.) 1992. *Alpha Architecture Reference Manual*. Digital Press, Bedford, Mass.

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.

STEELE, G. L., JR. 1977. Arithmetic shifting considered harmful. *SIGPLAN Not. 12*, 11 (Nov.), 61–69.

STEELE, G. L., JR. AND WHITE, J. L. 1990. How to print floating-point numbers accurately. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation* (White Plains, N.Y., June). *SIGPLAN Not. 25*, 6, 112–126.

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. Sun Microsystems, Mountain View, Calif.

SZYMANSKI, T. G. 1978. Assembling code for machines with span-dependent instructions. *Commum. 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.

THINKING MACHINES. 1989. *Connection Machine, Model CM-2 Technical Summary*. Thinking Machines Corp., Cambridge, Mass.

TJADEN, G. S. AND FLYNN, M. J. 1970. Detection and parallel execution of parallel instructions. *IEEE Trans. Comput. C-19*, 10 (Oct.), 889–895.

TORRELLAS, J., LAM, H. S., AND HENNESSY, J. L. 1994. False sharing and spatial locality in multiprocessor caches. *IEEE Trans. Comput. 43*, 6 (June), 651–663.

TOWLE, R. A. 1976. Control and data dependence for program transformations. Ph.D. thesis, Tech. Rep. 76-788, Computer Science Dept., Univ. of Illinois at Urbana-Champaign.

TRIOLET, R., IRIGOIN, F., AND FEAUTRIER, P. 1986. Direct parallelization of call statements. In *Proceedings of the SIGPLAN Symposium on Compiler Construction* (Palo Alto, Calif., June). *SIGPLAN Not. 21*, 7 (July), 176–185.

TSENG, C.-W. 1993. An optimizing Fortran D compiler for MIMD distributed-memory machines. Ph.D. thesis, Tech. Rep. Rice COMP TR93-199, Computer Science Dept., Rice University, Houston, Tex.

TU, P. AND PADUA, D. A. 1993. Automatic array privatization. In *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*. Lecture Notes in Computer Science, vol. 768. Springer-Verlag, Berlin, 500–521.

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, Calif., June). *SIGPLAN Not. 27*, 7 (July), 188–199.

WALL, D. W. 1991. Limits of instruction-level parallelism. In *Proceedings of the 4th International Conference on Architectural Support for Programming Languages and Operating Systems* (Santa Clara, Calif., Apr.). *SIGPLAN Not. 26*, 4, 176–188.

WANG, K. 1991. Intelligent program optimization and parallelization for parallel computers. Ph.D. thesis, Tech. Rep. CSD-TR-91-030, Purdue University, West Lafayette, Ind.

WANG, K. AND GANNON, D. 1989. Applying AI techniques to program optimization for parallel computers. In *Parallel Processing for Supercomputers and Artificial Intelligence*, K. Hwang and D. Degroot, Eds. McGraw Hill, New York, Chap. 12.

WEDEL, D. 1975. Fortran for the Texas Instruments ASC system. In *Programming Languages and Compilers for Parallel and Vector Machines* (New York, N.Y., Mar.). *SIGPLAN Not. 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.

WEISS, M. 1991. Strip mining on SIMD architectures. In *Proceedings of the ACM International Conference on Supercomputing* (Cologne, Germany, June). ACM Press, New York, 234–243.

WHOLEY, S. 1992. Automatic data mapping for distributed-memory parallel computers. In *Proceedings of the ACM International Conference on Supercomputing* (Washington, D.C., July). ACM Press, New York, 25–34.

WOLF, M. E. 1992. Improving locality and parallelism in nested loops. Ph.D. thesis, Computer Science Dept., Stanford University, Stanford, Calif.

WOLFE, M. J. 1989a. More iteration space tiling. In *Proceedings of Supercomputing '89* (Reno, Nev., Nov.). ACM Press, New York, 655–664.

WOLFE, M. J. 1989b. *Optimizing Supercompilers for Supercomputers*. Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Mass.

WOLF, M. E. AND LAM, M. S. 1991. A loop transformation theory and an algorithm to maximize parallelism. *IEEE Trans. Parall. Distrib. Syst. 2*, 4 (Oct.), 452–471.

WOLFE, M J AND TSENG, C. 1992. The Power Test for data dependence. *IEEE Trans. Parall. Distrib. Syst. 3*, 5 (Sept ), 591–601

ZIMA, H. P , BAST, H. J , AND GERNDT, M. 1988. SUPERB. A tool for semi-automatic SIMD/MIMD parallelization *Parall. Comput 6*, 1 (Jan.), 1–18.

Received November 1993, final revision accepted October 1994.