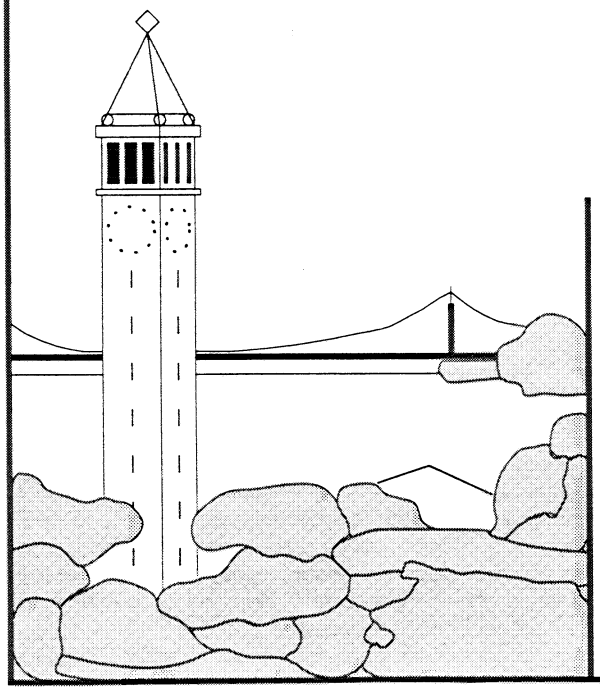
Transforming for Parallelism Using Symbolic Summarization

Oliver Joseph Sharp



Report No. UCB//CSD-95-882

May 1995

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Transforming for Parallelism Using Symbolic Summarization

by

Oliver Joseph Sharp

B.S. (Yale University) 1987
M.S. (University of California at Berkeley) 1991

A dissertation submitted in partial satisfaction of the
requirements for the degree of
Doctor of Philosophy

in

Computer Science

in the

GRADUATE DIVISION
of the
UNIVERSITY of CALIFORNIA at BERKELEY

Committee in charge:

Professor Susan L. Graham, Chair
Professor James Demmel
Professor Phillip Colella

1995

Transforming for Parallelism Using Symbolic Summarization

Abstract

Transforming for Parallelism Using Symbolic Summarization

by

Oliver Joseph Sharp
Doctor of Philosophy in Computer Science

University of California at Berkeley

Professor Susan L. Graham, Chair

Effective use of parallelism is an important goal of computing research. This dissertation describes MAGNIFY, an interactive analysis and transformation tool that is part of the Delirium programming environment. The purpose of the environment is to transform sequential FORTRAN programs to execute efficiently on massively parallel distributed memory architectures.

The main contributions of MAGNIFY are that it provides new and complex parallelization strategies that would be prohibitively difficult to implement by hand, and that it allows the programmer to determine their use. MAGNIFY, selectively guided by the programmer in an interactive dialog, applies a novel set of code transformations to the application. The transformations reveal opportunities for concurrency beyond those available through traditional loop-based optimizations. Normal parallelizing compilers focus on individual parallel computations, usually expressed in loops, and introduce synchronization operations after each one. MAGNIFY is able to manage the interactions among parallel computations to achieve more efficient performance.

MAGNIFY summarizes the data access behavior of sub-computations (such as loop nests) using *symbolic data descriptors.* The descriptors contain extensive symbolic and conditional information, providing more accuracy than previously developed summary structures. Once the code is analyzed, MAGNIFY uses the descriptors to apply transformations that expose concurrency and pipelining opportunities. The key transformation is *split*, which reduces synchronization constraints by sub-dividing computations. MAGNIFY also applies traditional loop transformations like interchange and loop-invariant code motion.

After the programmer has used MAGNIFY to transform an application, the parallelization strategy is encoded in an intermediate form based on two notations: a coordination language called Delirium and an annotation language called Dossier. An adaptive run-time system executes the application, using run-time information to improve the scheduling efficiency. The run-time system incorporates algorithms that allocate processing resources to concurrently executing sub-computations and choose communication granularity.

MAGNIFY has been used to analyze and transform three production scientific applications. Performance measurements show that the resulting parallel implementations are far more efficient than traditional static decomposition strategies on large numbers of processors.

Contents

List of Figures

List of Tables

Acknowledgements

This research has been sponsored in part by each of the following organizations, and we gratefully acknowledge their support:

- The Defense Advanced Research Projects Agency (DARPA), under contract DABT63-92-C-0026. The content of the paper does not necessarily reflect the position or the policy of the Government and no official endorsement should be inferred.
- The Fannie and John Hertz Foundation. Oliver Sharp was supported by a Hertz Fellowship during part of his tenure as a graduate student.
- Lawrence-Livermore National Laboratory. This work was performed in part at the Laboratory under the auspices of the U.S. Department of Energy contract No. W-7405-ENG-48.
- NASA Ames Research Center, San Diego Supercomputer Center. Both of these organizations provided access to their parallel computing facilities.

Chapter 1

Introduction

This dissertation describes MAGNIFY, a tool that is part of the Delirium programming environment. The purpose of the environment is to transform sequential FORTRAN programs to execute efficiently on massively parallel distributed memory architectures. We have successfully parallelized several production scientific applications, including GCM, a global climate model developed at UCLA, PSIRRFAN, an X-ray tomography application from UC/Berkeley, and AMBER, a molecular dynamics simulation from UCSF.

When parallelizing an application, there are four main tasks:

- Discovering time-consuming operations that can be executed in parallel.
- Transforming the code to reveal additional concurrency.
- Decomposing the program into sub-computations that can be executed independently.
- Choosing an efficient schedule for those sub-computations at run-time.

MAGNIFY is responsible for assisting in the first three tasks.

Existing parallelization systems fall into two categories: fully automatic compilation environments that perform all optimization without programmer assistance, and interactive systems that rely on the user to guide transformations. MAGNIFY belongs to the latter category; turn-key compilers have been disappointing in the performance that they can provide. The difference between MAGNIFY and other interactive environments is that it gathers extensive symbolic information and uses it to apply a new class of higher-level transformation. Application programmers have used these kinds of optimizations in the past but have been forced to apply them by hand - a process that is both tedious and error-prone. MAGNIFY carries out the optimizations automatically.

These new optimizations are based on the *split* transformation and rely on summarization analysis. There are two main strategies that are used to identify opportunities for parallelism: dependence analysis [22, 125] and summarization. The focus of dependence analysis is to analyze the array subscript expressions that appear inside of loops, applying various

tests to determine whether any two iterations can access the same data. Summarization, on the other hand, seeks to assemble a descriptor that encapsulates all the data that is referenced by a block of code. Existing systems do use summarization, primarily to examine interprocedural behavior, but they rely more heavily on dependence analysis because it is more accurate in identifying independence between loop iterations.

We chose to use summarization in the Delirium environment because it provides exactly the information needed by MAGNIFY and the run-time system to parallelize an application efficiently. In order to arrange for a computation to execute in parallel, the system must do two things:

- Allocate pieces of the computation to different processors.
- Arrange that the data used by each piece is available so that the piece can be executed.

The first task relies on finding opportunities for independent execution and dependence analysis is able to find more of them than summarization. However, the second task is equally necessary. If the compiler can identify, at compile time, the data that a computation will need, it can optimize communication and can control its cost. Otherwise, at run-time processors must continually request data on demand. That strategy is hopelessly inefficient on distributed memory machines because of the cost of message transfer between processors.

To achieve highly efficient parallel execution, data must be packaged together and its transfer carefully optimized. As a result, the additional accuracy of dependence analysis would not be useful to the Delirium environment — MAGNIFY can take little advantage of concurrency when the data usage cannot be summarized accurately.

The heart of MAGNIFY is the *symbolic data descriptor* (SDD), a summary of the data that is used by a block of code. This summary can be computed for any program fragment. Once two code fragments have been summarized, MAGNIFY can compare their usage of memory efficiently to identify independence. It can also use the descriptors to transform fragments, revealing additional opportunities for parallel execution. The process of transforming the program takes place as a dialog with the programmer; some transformations are applied automatically and others are done by request.

Once a program is fully transformed and decomposed, MAGNIFY outputs it in an intermediate form that relies on two languages: Delirium and Dossier. Delirium is a *coordination* language which expresses the relationship among a set of program fragments. Dossier is an annotation language for describing data transmitted between fragments. Both languages are described in detail in Chapter 7.

The program is then executed by RTS, a run-time system that incorporates a set of adaptive algorithms for efficiently scheduling the application on the available processors [83]. The scheduling decisions depend on data usage summarization; without accurate information, the run-time system is crippled. On existing distributed memory architectures, where communication latency is relatively high, it is impractical for a computation to acquire data on demand — it must be provided before the computation begins executing. The run-time system incorporates data communication overhead in making its scheduling decisions.

The remainder of the dissertation is organized as follows: Chapter 2 gives an overview of the Delirium programming environment and describes each of its components. Chapter 3 introduces the symbolic expressions which are used in the descriptors and Chapter 4 describes the various forms of analysis that MAGNIFY performs to prepare for building descriptors. Chapter 5 presents the algorithms for building descriptors and Chapter 6 demonstrates how they guide the transformation of the code. Chapter 7 examines the output of MAGNIFY, which feeds into the run-time system. Chapter 8 gives experimental results for MAGNIFY both in isolation and in combination with the other components of the environment. Chapter 9 concludes. Where relevant, each chapter ends with a discussion of related work.

Chapter 2

Overview of the Environment

MAGNIFY is part of the Delirium programming environment, a system that aids the programmer in transforming a sequential application into an efficient parallel version. The environment consists of several components, as diagramed in Figure 2.1.

2.1 MAGNIFY

The programmer begins with a FORTRAN program and an optional profile of its execution. MAGNIFY can often make better optimizations if it is given the execution profile of the program as reported by the tool `gprof` [57]. The programmer then runs MAGNIFY, which summarizes the data usage of the program at various granularities (loop-level, function-level, etc).

MAGNIFY consists of a sequence of passes. After the code is converted into static single assignment form (SSA) [41], subsequent passes manage potential aliases, track conditionals, perform extensive interprocedural analysis, and assign symbolic expressions to as many scalar values as possible. After the analysis is complete, MAGNIFY can summarize the data-access behavior for any program fragment of interest. The summary is expressed as a *Symbolic Data Descriptor* (SDD) and contains all the data that the code fragment reads and writes.

Once summarization is complete, MAGNIFY engages the programmer in an interactive dialog. Using a series of transformations, the programmer converts the program from its original sequential form into a parallel implementation. MAGNIFY maintains the current state of the program in an internal representation called WEB, tracking the effect of each transformation. The programmer can view the WEB representation and can examine each block of code that has not yet been parallelized. Where possible, MAGNIFY identifies independence between computations and between iterations of loops.

MAGNIFY performs some transformations automatically and carries out others at the behest of the programmer. In addition to traditional loop-based transformations, MAGNIFY is able

to use its data summary to apply more powerful and systemic modifications based on the *split* transformation (discussed in detail in Section 6.3). Because split is not limited to a single loop nest, MAGNIFY is able to work with the interactions between different loops and other sub-computations.

MAGNIFY outputs the transformed program in three forms: a set of FORTRAN code fragments (called *operators*) that perform individual computations, a description of the interaction of the fragments (in the coordination language Delirium), and a description of the data passed between fragments (in the annotation language Dossier).

Operators represent the finest level of computational granularity; the task of the run-time system (RTS) is to schedule operators on appropriate processors and to route data from one operator to another as needed. Because MAGNIFY will often be unable to determine the best granularity at compile-time, operators usually accept arguments which determine their execution cost. Most commonly, an operator represents the body of a loop and performs some sub-set of the loop's iterations. Here is a simple example:

```
do i = 1, n
  A[i] = A[i] + 1
```

The corresponding operator might look like this:

```
loop_operator(A,low,high)
  real A[low:high]

  do i = low, high
    A[i] = A[i] + 1
```

The operator is given a range of iterations to compute and a fraction of the array A to work on. RTS computes values for `low` and `high` depending on the current state of the system. When the amount of parallelism available is high and all the processors are busy RTS can reduce overhead by performing the original loop with just a few calls to `loop_operator`. When processors are idle, RTS will make many calls to the operator, giving each one a few iterations to perform.

The Delirium code for the program is a framework that defines the dependence relationship between the operators. Delirium is called a *coordination* language [53] because it does not perform computation itself, but merely arranges for the correct execution ordering of primitive computations that are expressed in another language.

While Delirium provides the dependence information needed for *correct* program execution, its variables are untyped and the primitive computations are treated as atomic objects. RTS requires much more information about data sizes and the execution behavior of the operators. Hence, the Delirium description is extended with a set of Dossier annotations that characterize the data being passed between operators and the behavior of the operators themselves.

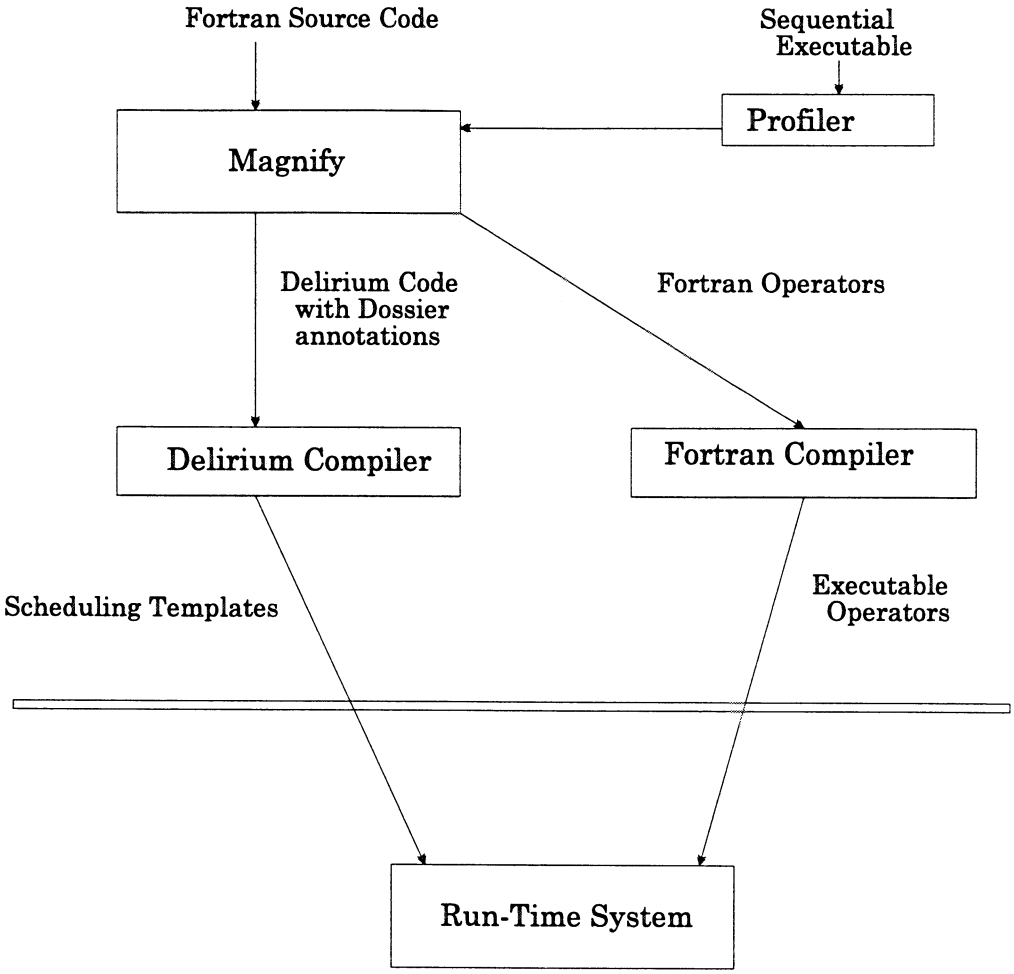


Figure 2.1: Overview of Delirium Programming Environment

2.2 Delirium Compiler and Run-Time System

The Delirium/Dossier description is passed to the Delirium compiler, which converts it into a series of *scheduling templates.* These schedules represent pre-defined scheduling strategies for the program, reducing the number of decisions that must be made at run-time. A particular schedule may handle the scheduling of a single Delirium statement or of a large number of them. RTS monitors the state of the system as the templates are executed, using adaptive algorithms to choose operator granularity and to allocate computation to the available processors efficiently.

2.3 Related Work

There are a variety of interactive systems that aid the programmer in parallelizing an application. They can be divided into two main groups, depending on whether they analyze the sequential code or not.

The systems that do not perform analysis are designed to simplify the task of expressing a parallel application. The programmer does all the work to decompose the program, but the system provides a linguistic or graphical mechanism for connecting the decomposed fragments together.

One of the earliest such systems was Schedule [45]. It provided a procedural way to describe the dependence relationships between chunks of FORTRAN code. The programmer calls explicit primitives to build the dependence graph on the fly, and the scheduler begins executing a code block when its predecessors have reported that they are finished.

Most of the other systems that follow this approach rely on graphical presentation and manipulation. Poker [95], for example, allows the programmer to use a graphical environment to connect primitive computations together. While the focus on visual programming is convenient for simple coordination patterns, it is not suitable for complex interconnections that are better expressed in a textual notation.

There are a number of other environments that provide this kind of functionality, enhancing the basic notion with features like a constraint language [112], language independence [24], or a virtual machine model that performs some automatic mapping to the physical machine [100].

The more ambitious interactive projects are similar to the Delirium programming environment; they perform extensive analysis of a sequential program, and present the results to the programmer. The programmer can modify and sharpen the results of the analysis based on contextual knowledge about the program. The system generally provides a set of restructuring operations that can be applied to the code. However, existing systems have focused on shared memory machines, relying most heavily on dependence analysis rather than the summarization used by the Delirium environment.

The ParaScope system [69] is one of the most ambitious restructuring environments. It grew out of an earlier project that developed a dependence analyzer, PFC [7], and an associated dependence browser called PTOOL [6]. Parascope supports a variety of views of the program, displaying dependences graphically and allowing the user to edit them. The system also has a broad spectrum of loop-based transformations that are applied automatically or at the programmer's behest. Parascope stores the results of many analysis passes in a database that can be updated and searched, using incremental analysis to reflect the changes due to code transformations.

Parafrase-2 [98] also relies on the dependence graph as the medium of communication with the user. It identifies potential dependences, using interprocedural analysis for more accuracy. It does not have as complete a set of interactive transformations as Parascope. The system is a source-to-source translator; it exploits parallelism for loops that are found to

be independent. The scheduling policy is embedded into the loops by inserting calls to coordination primitives within them. Parafrase-2 targets shared memory architectures.

The hierarchical dependence graph environment [113] is a graphical interface to a dependence database. The programmer can manually remove dependences that are spurious. The application can then be decomposed into a hierarchical graph representation (via interactive modification in the dependence browser). The output relies on a run-time system that schedules the nodes of the graph when they are ready to be executed [122].

Chapter 3

Symbolic Data Descriptors

A symbolic data descriptor (SDD) summarizes the memory access behavior of a block of code. Before giving a detailed description of the descriptors themselves, we will define the symbolic expressions that that can appear in them and the underlying memory model.

3.1 Symbolic Expressions

There are a few different kinds of symbolic expressions in MAGNIFY. *Arithmetic* expressions are sums; the terms are either integer constants, real constants, or a symbolic variable with associated integer coefficient. *Range* expressions describe a contiguous range of values and take the form [*start* .. *stop*, *skip*]. The endpoints of the range are arithmetic expressions and *skip* is an optional integer value. A *value* expression is either an arithmetic expression or a range; we chose this representation because the set of value expressions is closed under addition, subtraction, and multiplication by an integer. Also, various application studies have demonstrated that subscript expressions are generally limited to linear expressions of variables.

In addition, MAGNIFY supports symbolic *predicates*. A predicate is the disjunction of conjunctions of *inequalities*. An inequality describes the relationship of two value expressions with a comparison operator.

Here are some examples of each kind of expression:

arithmetic expressions:	5	$2 * a + 2 * b$	$2 * a + 4$
range expressions:	$[1..5]$	$[1..n, 2]$	$[a + 1..a + 2]$
predicate expressions:	$a < 1$	$(a = 3)$ or $((b < 5)$ and $(c = 3))$	

MAGNIFY includes a symbolic engine that provides a set of primitives for manipulating and evaluating symbolic expressions. For example, every kind of expression has a constructor that builds it out of primitive elements like integers, floating point values, and individual symbols. Table 3.1 describes the other operations that are available.

Operations on Predicates	
`or`(A,B)	return the disjunction of two predicates
`and`(A,B)	return the conjunction of two predicates
`not`(A)	return the negation of a predicate
Operations on Value Expressions	
`add`(A,B)	return the sum of two value expressions
`subtract`(A,B)	return the difference of two value expressions
`multiply_by_constant`(A,c)	return the product of a value expression and an integer or floating point constant
Evaluation Functions for Predicates	
`implies`(A,B)	determine whether asserting one predicate implies that a second is true
`are_disjoint`(A,B)	determine whether the predicates conflict (i.e. if one holds, the other can never be true)
`are_identical`(A,B)	determine whether the predicates are redundant because they express the same information

Table 3.1: Operations on Expressions

The evaluation functions return one of three values:

- TRUE: the property is proven to be true
- MAYBE: MAGNIFY cannot be certain one way or the other
- FALSE: the property is proven false

3.2 Memory Model

Descriptors rely on a segmented model of memory; the storage that a program can access consists of a set of disjoint segments. Each segment contains any number of individual memory blocks, each with a starting location and a length, and memory blocks may overlap

descriptor	::=	*guard* memory_block *pattern*
guard	::=	predicate_expr '**:**' | ϵ
pattern	::=	'**<**' *dim_patterns* *coupled_expr* '**>**' | ϵ
dim_patterns	::=	*dim_pattern* '**,**' *dim_patterns* | ϵ
dim_pattern	::=	*start* '**..**' *stop skip mask* | ϵ
start, *stop*	::=	arithmetic_expr
mask	::=	'**/**' predicate_expr | ϵ
skip	::=	'**,**' *integer* | ϵ
coupled_expr	::=	'**:**' predicate_expr | ϵ

Figure 3.1: Descriptor Grammar

each other arbitrarily. MAGNIFY creates a block for each variable and array in the program. It may also create blocks for parts of arrays or for an entire group of variables, depending on the behavior of the code. Note that the model does not currently attempt to track dynamic heap allocation and deallocation.

When a memory block cannot be located precisely, MAGNIFY creates an *unresolved memory block* (UMB). A UMB contains a mask that describes which segments it is known *not* to inhabit. MAGNIFY uses UMBs to summarize the behavior of a procedure when it doesn't know which memory blocks were passed in as arguments.

The reason that MAGNIFY uses the UMB is that even imprecise information is useful. For example, an unknown argument can be aliased to any global variable or any local variable outside the function, but it cannot refer to the called procedure's local variables. The UMB captures this restriction straight-forwardly. Such a UMB can then be passed through subsequent call sites along with local variables, and the disjunction will continue to be recognized.

3.3 Triples

An SDD consists of two sets of *triples*, one for the areas of memory that are read and one for those that are written. The read set contains locations which are (or may be) live on entry to the code being annotated; reads known to be preceded by writes in the write set are not included.

Each triple describes access to a given block of memory and is represented in the form $(G)B < P >$. G is an optional symbolic guard expression; the access represented by the triple is known not to occur if the guard is proven false. B is the memory block accessed. P, also optional, describes the pattern of access; if P is not specified, the triple refers to the entire memory block.

The grammar for a descriptor is given in Figure 3.1. It refers to three terminal symbols:

- memory_block — a structure that describes a memory block, as discussed above in Section 3.2
- predicate_expr — a predicate expression, as defined in Section 3.1
- arithmetic_expr — an arithmetic expression, as defined in Section 3.1

A pattern represents access to a subset of a memory block. When a block corresponds to an array, there is a pattern expression for each dimension of the array. The pattern for a particular dimension can be nil, meaning that the entire dimension is accessed, or it can be a range expression that covers a subset of the values in that dimension. The dimension pattern can be further restricted by a *mask* expression, as is shown in more detail below. The goal in computing the pattern expression and the mask is to express the most restrictive summary of the dimension values that might be accessed during the corresponding computation.

In addition to the pattern expression for each dimension, there is an optional predicate to handle *coupled subscripts.* Two expressions are coupled if the same variable appears in both of them. Figure 3.2 shows a few sample code fragments and their equivalent triples.

The fragment in Figure 3.2 (a) is an update to an array element which is conditional on the value of a boolean masking array called `act`. The associated descriptor has a guard expression on the access.

The next fragment, in (b), shows the same expression inside a loop. The descriptor summarizes all the references by replacing the induction variable with its range of values. Note that the upper end of the range is only available as a symbolic value. Also, the guard expression has been *promoted* into a mask. The predicate expression that makes up a mask is allowed to reference a class of special symbols that are used as placeholders. This example uses the symbol $*$ as a place-holder for a value in the range; the mask expression indicates that the j'th element in the range is not accessed if the expression $(act[j] = 1)$ is false.

Figure 3.2 (c) shows a fragment that has a coupled subscript, meaning that the same variable appears in more than one subscript expression. The weakness of a summary strategy that handles each array dimension separately is that it cannot directly account for coupling. The predicate expression uses the generalized placeholder symbol $*_k$ to refers to the value of the k'th subscript. In this example, the expression captures the fact that the value of the first and second subscripts must always be equal.

The descriptor representation is designed so that any symbolic expression can be safely eliminated. Guards and masks specify access more precisely; removing them yields a more conservative descriptor that implies the access will always take place. Pattern expression can be eliminated and MAGNIFY will make conservative assumptions about the data referenced.

```
if (act[i] = 1)
    A[i] = A[i] * 10
```

write: $(act[i] = 1)$:	$A < i >$
read:	$act < i >$
	i
$(act[i] = 1)$:	$A < i >$

(a) Descriptor for Conditional

```
do i = 1, n
    if (act[i] = 1)
        A[i] = A[i] * 10
enddo
```

write:	$A < 1..n/(act[*] = 1) >$
	i
read:	$A < 1..n/(act[*] = 1) >$
	$act < 1..n >$
	i

(b) Descriptor for Loop

```
do i = 1, n
    A[i,i] = A[i,i] * 10
enddo
```

write:	$A < 1..n : (*_1 = *_2) >$
read:	$A < 1..n : (*_1 = *_2) >$
	i

(c) Descriptor with Coupled Subscripts

Figure 3.2: Various Code Fragments and their Descriptors

3.4 Descriptor Operations

There are a number of operations that can be applied to descriptors. Table 3.2 summarizes them. The evaluation functions determine whether properties hold and return TRUE, MAYBE, or FALSE (see Section 3.1).

3.5 Related Work

This chapter has described Symbolic Data Descriptors, a summarization form. Summarization and dependence analysis are the two primary strategies for parallelizing sequential code. Although MAGNIFY does not use dependence analysis, its analysis strategy has taken ideas from that field and therefore this section gives a brief discussion of both dependence analysis and previously suggested forms of summarization. Further details on dependence analysis are available elsewhere for the interested reader [22, 125].

3.5.1 Dependence Analysis

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.

There are two kinds of dependences: *control dependence* and *data dependence*. There is a control dependence between statement 1 and statement 2 when statement S_1 determines whether S_2 will be executed. For example:

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

Two statements have a *data* dependence if they cannot be executed simultaneously due to conflicting uses of the same variable. There are three types of data dependences: *flow dependence* (also called *true dependence*), *anti-dependence*, and *output dependence*. S_4 has a flow dependence on S_3, for example, 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
```

Dependence analysis on scalar variables is relatively straightforward, but understanding the behavior of arrays is much more complex — particularly if they are being accessed within a loop. In straight-line code, each statement is executed at most once, so the three

Operations on Descriptors	
`make_empty`	Returns a null descriptor.
`expr_to_descriptor`(E)	Given an expression in FORTRAN, analyzes its behavior and constructs a descriptor that summarizes the data referenced
`guard`(D,E)	Given a descriptor and a symbolic expression, applies the expression to the contents of the descriptor (using ranges where possible, guard expressions where not) and returns the result.
`union`(D_1,D_2)	Given two descriptors, combines them. Groups related triples together when simplification is possible. Simplifies guard expressions or, when appropriate, removes them.
`intersect`(D_1,D_2)	Given two descriptors, returns the descriptor of the data touched by both of them. The operation is performed conservatively, so a null intersection guarantees that the two associated code fragments are independent.
`clean`(D, S)	Given a descriptor D and a set of acceptable variables S, finds every variable in D that is not in the set and remove it.
Evaluation Functions for Descriptors	
`interfere`(D_1,D_2)	Given two descriptors, determines whether they include any of the same data. This operation is an efficient alternative to testing for a null intersection.

Table 3.2: Operations on Descriptors

basic types of dependence 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.

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

Dependence analysis reveals whether two array references can refer to the same element in different iterations 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, they are described with a direction or distance vector. If the subscript expressions are too complex to analyze, dependence analysis must assume that 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 [20]. Two general and approximate tests are the GCD [117] and Banerjee's inequalities [18].

In addition, 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 [21, 125]. The Delta Test [56] 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 λ-test [79], multi-dimensional GCD [18], and the power test [126]. The Omega test [101] 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 [87]. They use a general algorithm (Fourier-Motzkin variable elimination [42]) as a backup. One advantage of the multiple dimension exact tests is their ability to handle coupled subscript expressions [56].

3.5.2 Summarization

Summarization is an entirely different strategy; rather than considering the relationship of various statements, it seeks to describe the data that the code fragment reads and writes. It is often used in interprocedural analysis, as is discussed in more detail in Section 4.5. The main disadvantage of summarization is that it is less accurate; dependence analysis can reveal that a given loop has independent iterations where summarization fails to do so. However, dependence analysis may find independence without being able to determine the data that each iteration relies on. While knowing that iterations do not conflict is sufficient to parallelize on a shared memory machine, a computation obviously cannot be allocated on a distributed memory machine unless the system can ensure that the data it needs will be available. The decisions made by the RTS are critically dependent on precise knowledge about the data being operated on.

There are a number of summarization forms which have been discussed in the literature. The earliest emerged from classic dataflow techniques which sought to classify behavior across function calls. The analysis involved computing sets of variables that *may* be read and written by the callee [23, 39]. The major weakness of this work is that it treats arrays as scalar values; any modification of an element is treated as a modification to the entire array. While highly efficient and adequate for many traditional optimizations like common sub-expression elimination, the analysis is not sufficiently accurate for the purposes of a system like MAGNIFY.

The rest of the forms concentrate on describing arrays. Like SDD's, they define the areas within the array that are read and written. The Symbolic Data Descriptor is similar to several of these proposals, but it extends them to include conditional information. It also is more aggressive in its use of symbolic expressions. The *split* transformation, discussed at length in Chapter 6, relies heavily on the extensions in the SDD. Chapter 8 analyzes the cost of computing the additional information and evaluates its usefulness in practice.

Burke and Cytron [27] *linearize* array references into a one-dimensional form. They describe array accesses as either a set of these references or in a polynomial that summarizes a regular pattern. Union operations are efficient, but testing for independence is expensive. This strategy is also not suitable for symbolic or conditional expressions used in the SDD.

The rest of the strategies define regions bounded by various kinds of expressions. Triolet et. al. [118] computes linear inequality expressions that bound the area accessed. These inequalities allow restricted symbolic expressions that include induction variables and other scalars. There may be an arbitrary number of bounding expressions, allowing complex shapes to be described. However, the generality exacts a price. The union operation, for example, involves finding the convex hull of the inequality expressions of the two descriptors being combined. The SDD does not face this difficulty because it contains a fixed number of expressions. It also includes conditional information, allowing it in some cases to describe data access more accurately.

Regular Sections [30] define ranges on each dimension of the array, like an SDD, and allow a stride expression. Symbolic expressions are in terms of global variables and procedure arguments; the SDD is more general because it relies on the SSA analysis to incorporate a wider range of symbolic values. Data Access Descriptors [14] can be thought of as a variant of regular sections; they use inequalities to define a convex polytope that contains the slice of the array being accessed.

Finally, atom images [77] bound the regions with linear expressions of induction variables. Each reference is stored in its own image, forcing independence tests to examine every pair of images.

Havlak and Kennedy [62] examine many of these representations, arguing that regular sections are sufficiently general to handle the data patterns that their environment encountered in practice. Chapter 8 demonstrates why the extensions in the SDD are necessary to support the split transformation.

Chapter 4

Preparing to Build Descriptors

A descriptor encapsulates the data that is accessed by a block of code, focusing in particular on arrays and the subset of their elements that are read and written. Before the descriptors can be assembled, the following sequence of passes transform the code into a more useful form and analyze it:

1. Construct a control flow graph.
2. Convert the code into static single assignment (SSA) representation.
3. Propagate scalar values through aggregate assignment.
4. Break aliases.
5. Perform interprocedural analysis.
6. Analyze conditional statements.

The process begins by constructing a *control flow graph* [2] and converting the code into static single assignment form (SSA) [41]. SSA has become a popular intermediate form for compilation because it simplifies many analysis algorithms. The key advantage to SSA is that it generates a unique name for every scalar value that is computed by the program. This is extremely useful in assembling descriptors because it allows the unique names to appear in symbolic expressions without being dependent on the context. An expression like $A > 5$ may be variously true and false at different points in the control flow graph, but $A_{10} > 5$, where A_{10} is an immutable value, does not suffer from the same ambiguity.

After performing SSA renaming, MAGNIFY traverses the flow graph and attempts to annotate each scalar variable with a symbolic expression that represents its value. MAGNIFY shares the basic idea of propagating symbolic expressions with strategies for performing *partial evaluation* [35], but the ultimate intent is different. Partial evaluation seeks to improve the performance of a compiled program by handling some of the computations at

compile time. Expressions containing constants are evaluated, and branch expressions are computed to identify and prune unreachable code.

While MAGNIFY does apply some of these optimizations, such as pruning code, it is primarily attempting to assign a meaningful expression to every variable that appears in a subscript expression. The intent is to restrict the set of elements that the subscript can refer to. In order to assemble an accurate description of array usage, MAGNIFY must aggressively assign values to scalar variables that are used in the subscript.

In addition to a value for each scalar, MAGNIFY annotates each CFG node with a set of *guard* expressions. These guards indicate that the given node is only executed under certain circumstances – when all the guards are true. When descriptors are built, the guards are added to that node's triples and, when possible, promoted into mask expressions.

The next six sections describe each of the steps in preparing for SDD construction. The end result is an annotated CFG with a symbolic expression for each scalar. When MAGNIFY is unable to track the value in the scalar, the expression will be nil.

4.1 CFG Conversion

MAGNIFY converts the program into a standard control flow graph[2]. To simplify algorithms that operate on the graphs, MAGNIFY performs code replication if necessary to assure that the graph is reducible. Each node is annotated with a set of descriptions, one for each statement in its associated code. Each statement description lists the scalar variables that are read, the variables that are assigned, and a descriptor containing any array references. For each assignment, MAGNIFY examines the expression being assigned and construct a symbolic expression for it (which may be nil). The assigned variable is annotated with that expression.

Figure 4.1 shows a simple procedure and its associated CFG. The flow graph reflects the conditional branching structure; each statement has a summary of its memory access behavior with respect to both scalars and array elements.

4.2 SSA Conversion

The next step is that the program is converted to SSA, yielding a version that has a unique name for each scalar value that is computed. The basic process of performing SSA conversion is somewhat involved and is thoroughly discussed elsewhere [41]. Part of the conversion process involves the construction of a dominator tree for the control flow graph; MAGNIFY uses the algorithm developed by Lengauer and Tarjan [75]. The tree is also used by subsequent passes.

There are two issues that are important in using SSA within MAGNIFY. The first is that, with one exception, assignments to array elements are ignored. The next section identifies one loophole in this general rule and the rationale behind it.

```
a = b[i]
if (x > 5) then
   b[i-1] = a + 5
else
   b[i] = b[i] + 1
endif
```

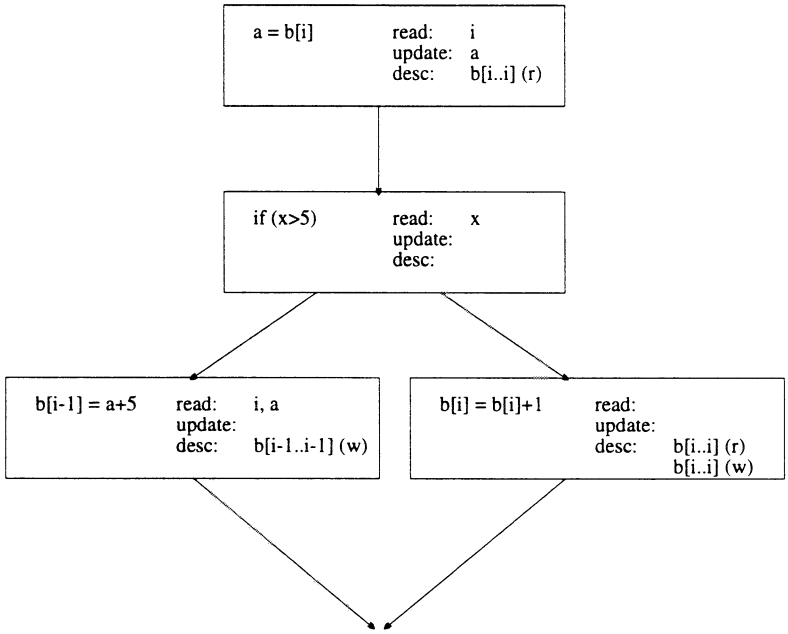


Figure 4.1: A code fragment and its CFG

```
a()
   common seen
   y = 3
   x = 5
   seen = 7
   b(x,y)
   return(seen+x)

b(i,j)
   common seen,unseen
   seen = i
   unseen = j
   i = i + 1
```

original source code

```
y_0 = 3
x_0 = 5
seen_0 = 7
(seen_1,x_1) = b(x_0,y_0)
return(seen_1+x_1)
```

SSA version of a()

Figure 4.2: Code fragment and equivalent SSA version

The other special consideration in MAGNIFY is interprocedural analysis. MAGNIFY must perform some interprocedural analysis during SSA or be forced to make very conservative assumptions that would greatly reduce accuracy. A second and much more elaborate form of interprocedural analysis is performed later; it is discussed in detail in section 4.5. For the purpose of converting the code to SSA, MAGNIFY can use a relatively simple approach. On first processing a procedure, MAGNIFY annotates the exit node of the CFG with a list of every modified *visible scalar*.

A visible scalar is a variable which, when modified, affects program state outside the function that contains the assignment. There are two ways for a FORTRAN variable to be visible: either it was passed into the procedure as an argument, or it is a global. Any alias to a visible variable is also visible.

To perform SSA correctly on a call site, any assignment to a visible scalar by the callee may need to be reflected in the caller. In our case, there is a corresponding assignment for every scalar that is passed in as a function argument. MAGNIFY treats the global variables more selectively, however.

For each assigned variable V that is visible because it is a global, the caller checks to see if V is in its name space. Through the mechanism of common blocks, a "global" variable in one FORTRAN procedure is not necessarily defined in another. Or, a global scalar in one procedure can be aliased to an element of an array in a different procedure. At a call site, MAGNIFY adds assignments to globals when there is an equivalent scalar in the caller's name space.

Figure 4.2 shows a sample piece of code and the associated SSA representation. It demonstrates how the assignment to the global `unseen` is ignored, while the assignment to `seen` and to the function parameter `x` both affect the variable renaming strategy.

Once SSA conversion has been performed, each variable in the program has been renamed to a unique value. These values may appear freely in symbolic expressions without consideration of the context. This is valuable because it allows a given SDD to be used for comparison anywhere else in the program without first considering naming scope.

4.3 Aggregate Propagation

It is infeasible, in general, to track the contents of every array element in a program. However, MAGNIFY does perform *aggregate propagation* to catch certain commonly used idioms:

```
do i = 1,10
  index[i] = a+i
  x = index[i] + 1
  B[x] = 10
enddo
```

In this code fragment, the programmer assigns the value `a+i` to an element in the array `index` and then uses that value two lines later. The programmer is using the array element to transmit a value, rather than creating a new temporary variable. To keep track of the fact that the variable `x` has the value `a+i`, MAGNIFY assigns an SSA name whenever an array reference uses the same subscript expression as a previous assignment.

The process of assigning reference names is similar to value numbering [36]. Whenever an array element is assigned a value that can be converted into a symbolic expression, the array name and its subscript expression are hashed into a table. For each array element reference, MAGNIFY checks the table to see if there is a match. If so, it creates a reference name and uses it in the two statements that are now linked together.

In assigning reference names, MAGNIFY begins by ignoring aliases. If there are no aliases, two array references with identical subscript expressions will have the same value because those expressions have been SSA converted. Aliasing can invalidate the propagation through the temporary names in two ways, however, as demonstrated by the following code fragment:

```
do i = 1,10
  index[i] = a+i            ; S1
  C[i] = C[i] * i
  y = i+1
  x = index[i]              ; S2
  B[x] = 10
enddo
```

As before, MAGNIFY creates an SSA variable for `index[i]` and tries to propagate the value to S2. The first kind of alias that can render the propagation invalid is an array alias — in this example, if the array `C` is an alias for `index`, by the time statement S2 is executed, the value of `index[i]` and hence the resulting value of `x` is no longer equal to `a+i`. The other kind of alias that prevents propagation is one that invalidates the subscript expression; if `y` and `i` are two names for the same variable, the assignment to `y` will change the location referred to by `index[i]`.

4.4 Alias Breaking

SSA creates a unique name for each value computed by a program. However, as was shown in the previous section, the presence of potential aliases can invalidate some of the SSA analysis. One solution is to use dataflow analysis to assemble the set of variables which may be aliased [23, 40]. These techniques have been generalized to consider array subscript expressions as well [27]. However, the data usage information that is assembled in order to build an SDD includes all the memory locations that have been modified and is therefore sufficient for performing alias analysis. Rather than performing redundant and separate alias analysis, MAGNIFY uses that information to detect potential aliases. By examining the descriptor for a code fragment, MAGNIFY can determine whether the fragment may have written over any SSA variables.

The process of handling a procedure begins by assigning a memory block to each argument. If we are analyzing the procedure for a particular call site, that site may provide a block for each argument and, if it is a scalar variable, an expression for its value. If no block is provided for the arguments, MAGNIFY assigns each a UMB.

The algorithm below analyzes the body of the procedure; MAGNIFY traverses the dominator tree of the CFG keeping track of the set of SSA variables whose values are known to be valid. When a symbolic expression is encountered, it is discarded if it refers to any invalid variables. The procedure is called initially with the CFG entry node and an empty set:

```
invalidate_aliases(N,legal_vars)
    for each valid φ
        for each source var V
            if V is invalid, mark φ invalid
        legal_vars = legal_vars ∪ phi's var)
    ∀S, S a statement in N
        if (S is a procedure call)
            handle_proc_call(S)
        invalidate_descriptor_exprs(legal_vars)
        ∀V ∈ legal_vars
            if (V's memory intersects S's descriptor)
                mark V invalid
            if (assignment in S intersects V)
                mark V invalid
            for each scalar assignment A in S
                legal_vars = legal_vars ∪ A
        for each rhs
            if (rhs_vars - legal_vars) ≠ ∅
                rhs = NULL
    ∀C, C ∈ N_cfg_children
        for each φ
            for each source var V
```

```
            if V not in legal_vars
                mark φ invalid
∀D ∈ N_dom_children
    invalidate_aliases(D, legal_vars)
```

The algorithm examines every definition of an SSA variable and each descriptor once. Descriptors appear at call sites, which generally have at least one SSA variable associated with them, and at each statement in the program. In practice, the cost of the analysis is well approximated by the number of SSA variables that have been defined.

4.5 Interprocedural Analysis

Interprocedural analysis is not, strictly speaking, a separate phase; it is intertwined within alias breaking. However, it is such an important part of the analysis that it requires a degree of elaboration and merits its own section. Before discussing the actual mechanism for propagating information through a call site, we give a brief overview that presents a unifying framework for a spectrum of interprocedural analysis strategies. By casting the analysis in terms of a general mechanism, MAGNIFY can easily tune its aggressiveness statically or dynamically.

4.5.1 The Analysis Framework

Interprocedural analysis is a crucial part of memory access summarization. When it encounters a call site, MAGNIFY must decide how aggressively to pursue its analysis. The simplest idea is to ignore interprocedural issues entirely and assume the worst about the callee's behavior. Most existing sequential compilers do exactly that, at least as their default behavior. However, parallelizing systems cannot be so indifferent without missing many or most of their opportunities to exploit concurrency. Large applications rarely provide sufficient parallelism within the bounds of a single procedure.

The next step up is to examine each procedure once in isolation, constructing a summary that is used by each caller. Most existing parallelization environments, as well as those commercial compilers that perform interprocedural analysis at all, use the strategy of a single summarization. While such a summary is a major improvement over no analysis, there are many situations where information is lost if a procedure is analyzed without considering its call sites.

The most aggressive approach is equivalent to inlining the body of every called procedure. Rather than examining each procedure a single time, the system reanalyzes every procedure at every call site. The additional accuracy of this exhaustive approach exacts a price: there is a potentially exponential increase in analysis time. While the cost does rise substantially, in practice, the threatened combinatorial explosion is unlikely.

There are relatively few published experiments that demonstrate how aggressively interprocedural analysis should be performed. Rather than choosing a single strategy and hard-

coding it into the system, MAGNIFY uses a general framework that allows it to tune its interprocedural analysis easily. The strategy allows the user to adjust the analysis, but also provides a simple mechanism for MAGNIFY to regulate itself.

The key idea is that MAGNIFY divides the call sites that invoke a given procedure into a set of equivalence classes. The call sites can either be *textual*, as they appear in the program, or *fully resolved*. The distinction is necessary because a given call as it appears in the source code can be reached through different paths in the call graph.

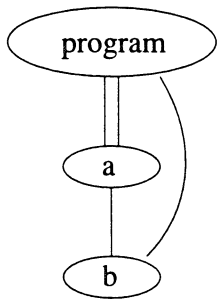
Figure 4.3 shows a short program, its static call graph, and its dynamic call graph. Note that the single textual call to procedure `b` within `a` generates two dynamic invocations with different arguments. When MAGNIFY refers to a call site, it can either refer directly to the textual location or use an extended naming mechanism: the fully resolved call site. The longer version includes information about the path through the call tree that preceded its invocation.

A fully resolved reference corresponds to a node in the dynamic call graph. The code fragment in the figure has five of them. MAGNIFY uses a call graph to represent resolved references, but in this dissertation we depict them textually by enumerating the nodes in the path from the initial invocation of the program. The path includes the procedure that is being called as its final node. If one procedure calls another more than once, the textual description will include a number after the procedure name to disambiguate. If the number n appears, it corresponds to the n'th reference to the called function in textual order. In the figure, for example, there are two calls to `a` in `program`, so the fully resolved call sites use the names `program'a'1` and `program'a'2` to refer to them respectively.

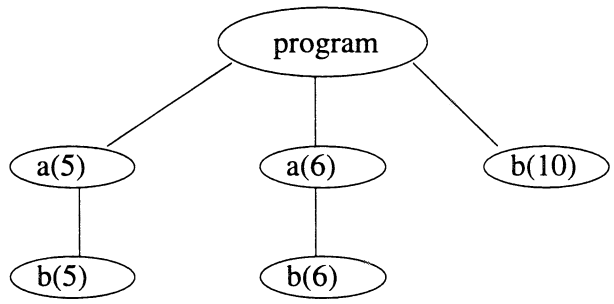
The equivalence class mechanism allows MAGNIFY to control the aggressiveness of the analysis. The simplest approach, summarizing each procedure once, corresponds to the construction of a single equivalence class that contains all the call sites. The most aggressive approach is to place each fully resolved call site into its own class; in other words, every procedure is reanalyzed for each call site. As a procedure generally invokes others which must also be reanalyzed, this strategy can be quite costly. In a recursive language, there must also be additional logic to break cycles.

The call site classification mechanism supports many levels of analysis between these two extremes — MAGNIFY could choose from a variety of partitioning schemes. For example, each textual call site could be its own class. Or, sites could be partitioned based on their characteristics. One natural choice is to partition based on the appearance of constants in the list of arguments, because constants are easily the most useful piece of information in sharpening the accuracy of the analysis.

In the case studies that are discussed in Chapter 8, the most aggressive approach proved to be acceptable in practice. The reported results quantify the added cost of exhaustive analysis vs. using a single summary per procedure. One design decision acting to reduce the cost of analysis is that MAGNIFY divides its effort on a given procedure into two phases. The first phase is call-site invariant and is only performed once regardless of the interprocedural strategy. The second phase is done for each equivalence class.



static call graph



dynamic call graph

```
program program
  a(5)
  a(6)
  b(10)
  end

subroutine a(n)
  b(n)
  return

subroutine b(i)
  x[i] = i
  return
```

Textual call sites:

`a(5)`	The first call in `program`
`a(6)`	The second call in `program`
`b(10)`	The third call in `program`
`b(n)`	The call in function `a`

Fully resolved call sites:

`program‘a‘1`	The first call in `program`
`program‘a‘2`	The second call in `program`
`program‘b`	The third call in `program`
`program‘a‘1‘b`	The call by the first execution of `a`
`program‘a‘2‘b`	The call by the second execution of `a`

Figure 4.3: Call site classification

If the current level of analysis becomes too costly in the future, MAGNIFY can use the classification mechanism to reduce it. Such a tuning mechanism might also be very useful in a stand-alone compiler, where the permissible amount of analysis is much lower than in an interactive tool like MAGNIFY.

4.5.2 Equivalence Classes Defined

In its initial parsing, MAGNIFY examines the call sites to each procedure and divides them into classes. As was just discussed, the current strategy aggressively assigns each fully resolved call site to its own equivalence class.

An equivalence class has two components:

- A — the set of textual call sites and/or fully resolved call site instances that belong to the class
- E — an *argument expression* containing a code for each of the parameters of the call

The latter is a simple pattern that identifies the amount of information to propagate through the call site. For each parameter, the expression is one of the following:

- Σ — pass all information through
- Δ — pass only information that is invariant to the context of the callee's invocation
- c — pass constant (integer, ASCII, or floating point) value c
- ϵ — pass nothing

The choice of argument expression reflects the analysis strategy being used. If each procedure is being analyzed once without considering its callers, Δ is the appropriate summarization strategy to use for each argument. It only considers information that is available locally within the procedure. For example, suppose that the following code fragment appears in an application:

```
subroutine g(a)
    b = 2
    call f(a,b,5)
```

An equivalence class containing the call to function `f` could be described by the argument expression (Δ, Δ, 5). In that case, MAGNIFY would not pass any information for the first argument because its value depends on the way that `g` is called. The value of the second argument does not, so MAGNIFY would use the value 2 for the second argument. The third argument will be the constant value 5.

By contrast, the current analyzer strategy is to place each fully resolved call site into its own class and to use Σ for each parameter in the argument expression. In this example, only the value passed for the first argument is affected. When it looks at the call site, MAGNIFY would pass any available information through; if it has a symbolic expression representing the value of `a`, that expression would be passed through as an annotation on the first argument when the procedure `f` is analyzed based on this particular invocation of it.

Figure 4.4 shows a larger example that uses a more restricted analysis than MAGNIFY would apply. The code contains four calls to a procedure `proc` that have been divided into three equivalence classes. Each call to `proc` in the call graph is labeled with its textual name (one of W, X, Y, or Z) and a number indicating to which of the four equivalence classes it belongs.

The first class contains the first three calls. The information that will be passed through those call sites are the constant value (1) for the first argument, context invariant information for the second, and nothing for the third. The value for the second argument varies over the three calls; for call sites `W` and `X`, MAGNIFY passes in an annotation indicating that the argument is less than 1. However, no information is passed through for the second argument of call `Y` because its value depends entirely on the procedure argument `b`. Simply by looking at the text of `pepper` and without using any information about its callers, MAGNIFY does not know anything about the value of `b`.

All of the previous call sites were allocated textually. The final call site, Z, has two fully resolved instances (`salt'pepper'1'proc'2` and `salt'pepper'2'proc'2`) that have been separated into two classes. The resolved call site naming mechanism elaborates the path through the call tree. The number following `pepper` indicates which of the two calls in `salt` that the path represents; the number following `proc` similarly distinguishes between the two calls in `salt`. For the two classes containing the fully resolved call site, all known information available at the call site is passed through.

Note that although the single textual call has two fully resolved sites based on the fragment as given, if there were more than one call to `salt` or other calls to `pepper` elsewhere in the program, there would be an additional set of fully resolved call sites to manage.

4.5.3 Interprocedural Algorithm

The following algorithm is used to analyze a call site:

handle_proc_call(S, *legal_vars*)
    C = equivalence class of S
    if $C \notin$ *analyzed*
        numargs = number of args in S
        for $n = 1..numargs$
            $argval_n = S_{arg_n} \propto C_{E_n}$
            $argblock_n$ = memory block for S_{arg_n}

```
procedure salt(a)
pepper(a)
pepper(a*2)
   ...
if (a < 1) then
  proc(1,a,x+5)        ; call site W
  proc(1,a,x+7)        ; call site X
endif
end

procedure pepper(b)
proc(1,b*2,x)          ; call site Y
proc(b*2,b,x-3)        ; call site Z
end
```

Equivalence Classes:

1) $A = \{\texttt{W}, \texttt{X}, \texttt{Y}\}$ $\quad E = \{1, \Delta, \epsilon\}$
2) $A = \{\texttt{salt`pepper`1`proc`2}\}$ $\quad E = \{\Sigma, \Sigma, \Sigma\}$
3) $A = \{\texttt{salt`pepper`2`proc`2}\}$ $\quad E = \{\Sigma, \Sigma, \Sigma\}$

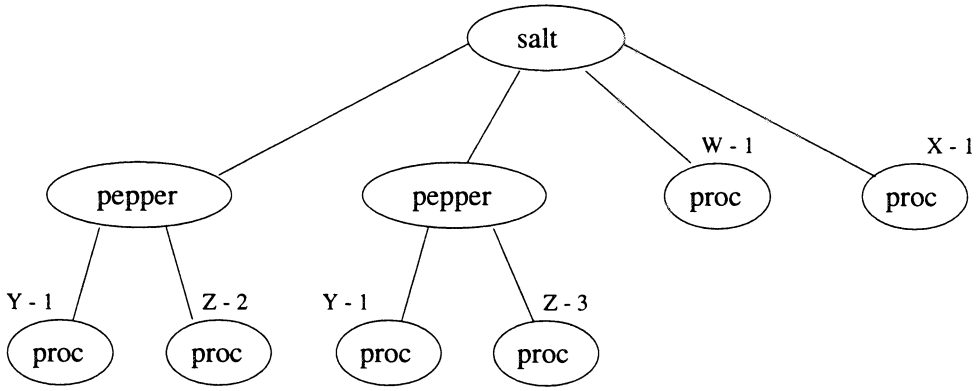


Figure 4.4: An example using equivalence classes

```
    T = procedure that is the target of S
    C_D = build_descriptor(T, argval, argblock)
∀V ∈ legal_vars
    if (V's memory intersects C_D's descriptor)
        mark V invalid
```

The algorithm begins by examining the site to identify its equivalence class. If an analysis has already been performed for that class, a descriptor will be available. Otherwise, MAGNIFY constructs a value and a memory block for each argument. The value is derived by applying the appropriate field of the class's argument expression to the call site. The value S_{arg_n} is the value that MAGNIFY has computed for the n'th argument of the call. C_{E_n} is the n'th element of the argument expression of the class. The application of one to the other is represented in the algorithm by the operator $\propto$. MAGNIFY also assigns a memory block (if none is known or can be passed, it uses a UMB). When a part of an array is passed, MAGNIFY constructs a new memory block for the fraction of the original block that is exposed to the callee.

A common FORTRAN technique causes additional complexity: the language allows the programmer to alias variables other than the ones actually passed as arguments. For example, if a series of scalar variables are declared in a common block, a procedure can pass the first one as an argument that is aliased to an array. The callee can use the array to walk through all the scalars in the common block and modify them at will. Although this is technically a violation of the FORTRAN standard, it is often performed in practice.

To handle this case, MAGNIFY examines the incoming memory blocks and identifies the ones that are smaller than the callee declares them to be. Extended memory blocks are created, so that the memory behavior of the callee will be properly reflected at the call site. When the array bounds are expressed symbolically with values known only at execution time, MAGNIFY assumes conservatively that the memory block may extend across the rest of the segment.

Once the formal parameters of the procedure are appropriately assigned memory blocks and values, MAGNIFY analyzes the procedure body. The result is an annotated exit node containing a symbolic value for each visible scalar and for the return value (if any). MAGNIFY connects these scalar values with the corresponding variables at the call site. Conceptually, all the scalar assignments in the callee are performed simultaneously, followed by the assignment of the return value. By processing in that order, MAGNIFY ensures that aliasing will be detected and will invalidate symbolic expressions correctly. The callee's descriptor is added to the descriptor for the call site. All symbols in the returned expressions and the returned descriptor are mapped into the caller's name space. The final result is a descriptor that describes the memory behavior of the call statement.

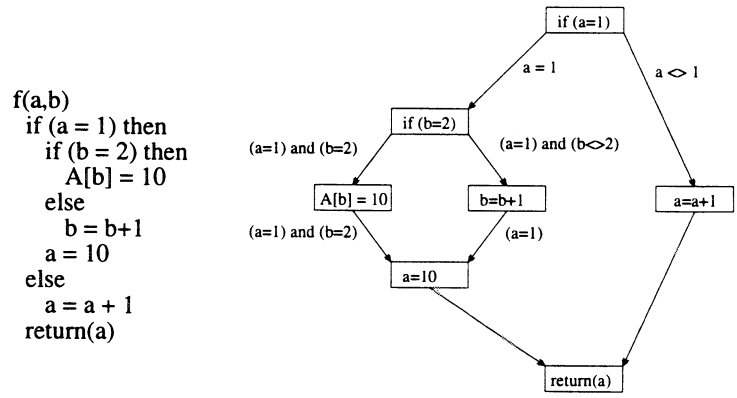


Figure 4.5: Code fragment with equivalent guarded CFG

4.6 Conditional Propagation

The goal of conditional propagation is to annotate each CFG node with a set of *guards* — conditions that must be true for the node to be executed. Guards are computed in two steps; the first takes place during the CFG pass (omitted earlier to simplify the discussion). When each CFG node is annotated with memory usage information, nodes with more than one child are handled specially. MAGNIFY attempts to derive a symbolic expression (*branch_dom*) for each child which, if false, indicates that the child is not executed. Figure 4.5 shows a code fragment, the corresponding CFG, and the guard expressions.

The second step is to propagate the guard expressions through the CFG. If a guard expression determines whether a CFG node N is executed, then it also controls execution of the nodes that N dominates. For each CFG node N, the following algorithm constructs N_{guards}, the set of guard expressions for node N. It is initially called on the entry node, which has an empty guard set:

construct_guard_set(N)
  $\forall D, D \in N_{dom\_children}$
    $D_{guards} = N_{guards}$
    if ($D \in N_{cfg\_children}$ and D has associated *branch_dom*)
      $D_{guards} = D_{guards} \cup$ *branch_dom*
      if (D_{guards} is inconsistent)
        $D_{pruned} = TRUE$, prune_CFG(D)
    if ($\neg D_{pruned}$)
      construct_guard_set(D)

The algorithm uses the control flow graph and its dominator tree to traverse the nodes of the function. $N_{dom\_children}$ is the set of N's children in the dominator tree. $N_{cfg\_children}$ is the set of its children in the control flow graph.

MAGNIFY uses its symbolic expression evaluator to merge multiple guards. When the set

of guards are mutually inconsistent, the corresponding node is guaranteed not to execute. MAGNIFY prunes that node out of the CFG along with any nodes that it dominates. Conditional propagation does not currently represent a significant cost as it involves a single pass over the CFG's dominator tree.

4.7 Related Work

The preparatory phases of MAGNIFY use a number of existing techniques that have been cited in the discussion above. The novel aspects of the analysis are the strategy for handling interprocedural issues and the propagation of symbols; both problems have been studied by other researchers in a variety of contexts.

4.7.1 Interprocedural Analysis

Every parallelization system of any import has been forced to address the problem of analysis across procedure boundaries. In most cases, the safe assumption that any procedure call invalidates most of the current state renders parallelization unworkable.

The simplest strategy for non-recursive programs is to use inlining [3, 110], either selectively [17] or wholesale. Unfortunately, the potentially exponential increase in the size of the code limits the usefulness of the strategy. Some studies have shown that the increase of size in the compiler's internal representation is not as large as the source code increase [38]. Nevertheless, the great majority of compilers that perform interprocedural analysis do so through summarization.

The early approaches were designed to allow optimization of a sequence of code containing procedure calls. The intent was not so much to exploit the behavior of the called procedure as it was to ensure that a given analysis was not invalidated by side effects. Register allocation, for example, improves dramatically when the compiler can determine which values will be overwritten by the called procedure. Otherwise, the only resort is to use a safe and inefficient strategy of saving any value that may be needed after the call.

The general strategy is to compute a summary of the behavior of a procedure and use that summary at each call site. The early approaches that were based on dataflow techniques [23, 29, 93] focused on scalar variables. Arrays were treated as a single atomic unit; modifying any element in the array was equivalent to modifying every element. Such a strategy is efficient to compute but prevents most parallelization of scientific codes, because decomposing the arrays is the single most important issue.

As was argued previously, summarization is an effective strategy to support the array decomposition on a distributed memory parallel machine. Even in compilation environments that rely most heavily on dependence analysis, summarization has generally been used to perform interprocedural analysis. A variety of summarization forms have been proposed, including the Regular Section [30], the Data Access Descriptor [14], the region [118], and the atom image [78].

The regular section represents the data access behavior with a set of patterns drawn from a lattice of expression types. The patterns consist of the upper and lower bounds of a range and a value for stride. The expressibility is essentially the same as array notation in the ANSI Fortran-90 standard [90]. Symbolic expressions can appear if they have been assigned a code through global value numbering. The symbolic code can be used for the bounds of an array range or for the stride. Once an expression has been converted into a code through numbering, no further manipulations are permitted. No conditional information is represented.

The data access descriptor is similar to regular sections but it uses diagonal constraints to enclose a convex polytope representing the area of an array that is accessed. The discussion briefly mentions the possibility of including symbolic values, but does not present algorithms for incorporating and manipulating them.

Triolet's regions use a set of linear inequalities to enclose an array region, allowing more accuracy in capturing triangular array sections. Induction variables are allowed to appear in the expressions. Other symbols are not included, however. Atom images also use bounds on the induction variables to capture triangular spaces. Havlak et. al [62] argues that the increase in accuracy using these approaches instead of regular sections is not necessary for the transformations in the PFC parallelization environment.

Symbolic Data Descriptors are similar to several of these forms, regular sections in particular. The SDD is distinguished chiefly by the amount of symbolic information that it contains. By incorporating conditional information and symbolic expressions that can be manipulated, MAGNIFY is able to identify additional opportunities to expose concurrency. While traditional loop-based optimizations are well served by the less detailed representations, MAGNIFY embodies a new set of transformations based on *split* that use the additional symbolic information heavily. The case studies in chapter 8 describe the series of transformations needed to parallelize three scientific applications. The discussion identifies those transformation steps that could not have been accomplished using the other summarization forms.

An alternative to inlining is *cloning* [37], where a given procedure is specialized into multiple versions. The call sites that invoked the original procedure are modified to call the appropriate new version. The reason for the specialization is to take advantage of features of the call sites — most commonly several call sites pass the same constant value for a given argument. When the compiler knows that a given argument will always have a particular value, it can optimize the body of the procedure more effectively. Additionally, there may be opportunities for parallelization that were not legal in the original form.

The effect of cloning on analysis can be achieved in MAGNIFY by dividing the textual call sites into one equivalence class per specialized version of the procedure. This strategy yields the same information at each call site as cloning does, but it does not provide the benefits of code replication.

4.7.2 Symbolic Analysis

Symbolic analysis has been investigated for a variety of purposes. Compiler researchers have used it on scalar variables to perform *partial interpretation*, evaluating expressions at compile-time. The evaluation could then be used, for example, for verification that a computation is correct [60, 70], test data selection [26], debugging [47], and optimization of execution time [32]. Clarke et. al present a survey of these applications [34].

One early strategy for assembling symbolic information about a program was to identify paths of execution through the code and symbolically execute it [35]. The result was a set of constraints on the program's variables to verify correctness assertions.

A more abstract group of research projects have investigated the bounds of symbolic arithmetic and using symbolic values to capture ranges [11, 61].

Symbolic information has also been used by vectorizing and parallelizing compilers, in addition to the work discussed above for summarization. One of the earliest compilers to manage symbolic expressions was the Bulldog compiler [46]. It attempts to compute a range of possible values for scalar variables appearing in a subscript expression. In addition, the user can refine the statically derived range by asserting simple inequality expressions.

The VATIL vectorizer [80] uses symbolic expression manipulation in attempting to vectorize loops. If the classic dependence tests fail to reveal independence, the system applies additional and more costly ones that incorporate symbolic inequalities.

Haghighat et. al [58] describe the use of symbolic expressions in Parafrase-2, a compilation environment developed at the University of Illinois. It combines symbolic expressions using a join function, as opposed to the path-based strategy that was discussed above. MAGNIFY avoids the need for a separate join operation because the SSA representation handles the merging of expressions from different incoming control flows. The paper also summarizes the use of symbolic analysis in other parallelizing compilers.

Chapter 5

Building the Descriptors

After the analysis described in the previous chapter, each statement in each CFG node is annotated with a descriptor that summarizes the data it touches. The descriptor contains only symbolic expressions that are known to be valid. Also, each CFG node has a guard set containing symbolic expressions that are known to be true if the node is executed. MAGNIFY can now construct a descriptor for any code fragment.

5.1 The Algorithm

The function that constructs a descriptor, `build_descriptor`, takes two arguments: the set of CFG nodes to describe (S) and the set of SSA variables that can appear as symbols in the resulting descriptor (V). The return value is a descriptor D that summarizes the data access behavior of the specified nodes; it is computed by performing a top-down traversal of the dominator tree. The purpose of the second argument, V, is that it provides a simple but powerful mechanism to customize the returned descriptor for various uses. We will demonstrate its use with several examples, after first explaining the basic algorithm.

As `build_descriptor` proceeds through the dominator tree, it checks each node to see whether it is a member of S. If so, that node's statements are traversed and their descriptors added to D. The statement descriptors are guarded by the accumulated set of guard expressions that appear on the path through the dominator tree.

Because the descriptors for dominating nodes are encountered first, the descriptor union algorithm is able to determine whether new entries to the descriptor's read set are live on entry. It is important for MAGNIFY to track liveness because the analysis can reduce data communication requirements at execution time. Before a given code fragment is executed, every memory location that it reads and that is live on entry must be properly filled in. Without analysis, MAGNIFY would be forced to make the conservative assumption that all data is live on entry. By pursuing an aggressive strategy, MAGNIFY is often able to prove otherwise and hence reduce scheduling overhead. The worst-case is always available as a fallback when the analysis fails or is inconclusive.

The following shows the algorithm for building a descriptor, stated more formally. There are three arguments:

- S, the set of CFG nodes to summarize
- V, the set of acceptable variables that may appear in the returned descriptor
- N, the root of the CFG dominator tree

The real work is done by a recursive routine called `build_dirty_descriptor`; it takes the same three arguments plus these two extra ones:

- D, the current descriptor (on first invocation, empty)
- G, the guard expression set (on first invocation, empty)

```
build_descriptor(S, V, N)
    D = build_dirty_descriptor(S, V, N, ε, ε)
    return(clean_descriptor(D,V))

build_dirty_descriptor(S, V, N, D, G)
    if (N has a guard set)
        add guard set to G
    if (N ∈ S)
        ∀L, L a statement in N
            Temp = L's descriptor, guarded by G
            D = D ∪ Temp
    ∀C, C a child of N
        D = D ∪ build_descriptor(S, V, C)
    return(D)

clean_descriptor(D,V)
    ∀ triples T, T ∈ D
        ∀ expressions E, E ∈ T
            clean_expression(E,V)
```

As explained above, `build_dirty_descriptor` traverses the dominator flow graph recursively. Each time it finds a node that it has been asked to describe, it accumulates that node's descriptors into a summary descriptor. If the node is guarded, the guard is applied to each new entry as it is added. After the dominator tree traversal is complete, the descriptor contains a summary of the data access behavior of the set of nodes. However, we call that descriptor *dirty* because it may contain symbolic variables that are not in V, the set that specifies which variables may legally appear in the final returned descriptor.

The last step, then, is to *clean* the descriptor — i.e, remove the unacceptable variable references. For each variable v referenced in the descriptor, `build_descriptor` checks if

v is acceptable (i.e. whether $v \in V$). If not, then the symbolic expression containing v must be eliminated or replaced by an equivalent expression that references only acceptable variables. `clean_expression` finds each unacceptable variable and systematically replaces it with equivalent symbolic expressions until one of two conditions holds: every variable in the final expression is acceptable, or an unacceptable variable remains that cannot be replaced. In the first case, the expression is considered clean and remains. In the second, the expression is illegal and must be removed.

The descriptor is designed so that MAGNIFY can always opt conservatively to remove a symbolic expression. If a guard or mask is removed, the corresponding memory reference is assumed to occur unconditionally. If a pattern is removed, the entire range of data in the given dimension is assumed to be referenced.

5.2 Applying the Algorithm

The algorithm in the previous section has a second argument which specifies the variables that could appear in the final descriptor. The ability to specify which variables may appear unresolved allows MAGNIFY to customize the descriptor for its intended use.

The following list contains four typical kinds of descriptors. In each case, it gives the arguments for `build_descriptor` to assemble that descriptor properly.

1. **Loop Iteration** To summarize the data referenced in a given loop iteration, MAGNIFY invokes `build_descriptor` on the set of CFG nodes that make up the loop body and includes the induction variable in V, the set of legal variables. By permitting the induction variables to remain as primitive values, description of the data referenced is restricted to the behavior of a single loop iteration. Figure 5.1 (a) shows a loop and (b) shows the descriptor for one of its iterations. Note that the variable `i` remains unresolved.

2. **Loop Body** To summarize the entire loop body, rather than just a single iteration, MAGNIFY again invokes `build_descriptor`. However, it changes the arguments slightly by removing the induction variables from V. When `clean_descriptor` encounters any induction variables in the final descriptor, it identifies them as illegal and attempts to resolve them. Within a loop, the value taken on by an induction variable is the entire range of values in the iteration space, so the result of the cleaning operation is that the descriptor now summarizes the behavior of the entire loop.

   Figure 5.1 (c) shows the result for the previous example; since the variable `i` may no longer appear, it is resolved into the induction range $2..n$.

3. **Function Body** A function's descriptor is based on all the CFG nodes that make it up. Any variable may appear unresolved. Figure 5.2 (a) shows a function and (b) its descriptor.

4. **Function Side-Effects** Often MAGNIFY wishes to know not the data that a function uses but the effect that a call to that function will have on the global state. The data

```
do i = 2, n
  A[i] = A[i] + B[i] * c
  B[i+1] = B[i+1] * C[i-1]
```

(a) loop code

write:	$A < i >$
	$B < i+1 >$
read:	$A < i >$
	$B < i..i+1 >$
	$C < i-1 >$
	c

(b) descriptor for an individual iteration

write:	$A < 2..n >$
	$B < 3..n+1 >$
read:	$A < 2..n >$
	$B < 2..n+1 >$
	$C < 1..n-1 >$
	c

(c) descriptor for entire loop body

Figure 5.1: Loop Code and its Descriptors

of interest are global variables, static variables declared within the function, and the arguments that are passed in. This more specific version of the function's descriptor is computed by declaring that local variables (except static ones) may not appear.

Note that the elimination of local variables from global state does depend on the memory mode. If local variables are stack allocated, as is traditional in Algol-like languages, eliminating them from the side-effect analysis is correct. The traditional FORTRAN model, however, states that a given local variable may be allocated statically to a given memory location. If the compiler follows that convention, more complex handling of local variables may be necessary to support correct execution.

Figure 5.2 (c) shows the side-effect descriptor of the function given in (a). The local variables have been removed but common variables and procedure arguments remain.

```
function sample(A)
common B, n

real sample, A[1:n], B[1:n], c, d
integer n, i

c = 0.
d = 0.

do i = 1, n
  c = c + A[i] * B[i]
  d = d * A[i] * B[i]

sample = c+d
return
```

(a) function code

write:	c
	d
	i
read:	$A < 1..n >$
	$B < 1..n >$
	n

(b) descriptor for entire function

read:	$A < 1..n >$
	$B < 1..n >$
	n

(c) descriptor for function's side effects

Figure 5.2: Function and its Descriptors

$reanalyze_{avg}$	the average number of procedure calls analyzed for each statement
$proc_{avg}$	the average time required to analyze a procedure
$triples_{avg}$	the average number of triples in a descriptor
$expr\_union_{avg}$	the time required to compute the union of a descriptor and an expression
num_exprs_{avg}	the number of expressions appearing in a triple
$dirty\_var_{avg}$	the number of dirty variables per descriptor
ops_clean_{avg}	the number of symbolic operations required to clean a given symbol
op_cost_{avg}	the cost of a symbolic operation

Table 5.1: Average quantities used in the cost formulas

5.3 Cost of Descriptor Assembly

The algorithm for assembling descriptors takes a CFG node set S and traverses the dominator tree of the enclosing function once. For each CFG node in S, the algorithm applies a descriptor union operator to each statement in that node. After the descriptor is fully assembled, each symbolic expression in it is cleaned.

It is not particularly meaningful to analyze the algorithm on the basis of its worst-case complexity, because in theory the cost could be astronomical — each operation could be required to manipulate every variable in the program. In practice, the algorithm cost remains within reasonable bounds even when aggressive analysis is performed. Furthermore, the cost of each operation depends heavily on the complexity of the symbolic expressions that appear and on the frequency of interprocedural analysis. Both are easily controlled within MAGNIFY because the sub-systems responsible for them include a tuning mechanism that allows for self-regulation.

More revealing than a complexity analysis, then, is an estimate as to the expected cost of each operation. The formulas below represent the cost of descriptor assembly based on a number of averages. Table 5.1 summarizes the averages that are used. The actual value for each average is likely to vary somewhat between applications, depending in large measure on programming style, but is expected to remain within a normal range. Chapter 8 measures the value of those averages for several real-world case studies and demonstrates that the cost of assembly is not prohibitive, even using exhaustive analysis. However, if the cost does become unacceptably high, that chapter demonstrates that execution time can be dramatically reduced in exchange for less accurate interprocedural and symbolic analysis.

This formula represents the total cost of one call to `build_descriptor`:

$$\left(\sum_{s \in S} (s_L * union_{avg}) * (reanalyze_{avg} * proc_{avg})\right) + clean$$

where

$$s_L \equiv \text{the number of statements in CFG node } s$$

The expression contains a sum, representing traversal cost, and a second quantity, *clean*, that reflects the cost of cleaning the descriptor. The traversal cost has two components: the cost of performing union operations on descriptors, and the cost of doing interprocedural analysis. The following sections analyze each of these components.

5.3.1 Union Operations

The basic operation performed by the algorithm is, for each statement in each CFG node in S, to compute the union of that statement's descriptor with a running summary descriptor. The formula for computing the cost of a union is

$$union_{avg} = triples_{avg} * expr\_union_{avg} * num\_exprs_{avg}$$

In other words, $union_{avg}$ is the product of the average number of triples, the average number of expressions in a triple, and the average cost of performing a union operation on each expression. These three values are averages and typical values for them are reported in the results section.

5.3.2 Interprocedural Analysis

The second half of the sum in the overall cost expression describes the cost of interprocedural analysis. It is the product of the average number of procedures analyzed per statement and the cost of an analysis. The number of analyses is

$$reanalyze_{avg} \equiv calls_{avg} * reanalyze_{prob}$$

The first quantity, $calls_{avg}$, is the average number of procedure calls per statement. It is dependent on the programmer's style. The second value is the probability that a given call will need to be reanalyzed and is heavily dependent on the interprocedural analysis strategy being used. The most aggressive approach, which is the one that MAGNIFY currently uses, always reanalyzes a procedure whenever it is invoked. In other words, $reanalyze_{prob} = 1$. Using the more common strategy of summarizing each procedure's behavior once yields a much lower probability of $1/invoke_{avg}$, where $invoke_{avg}$ is the average number of textual call sites to each procedure.

The other value controlling the cost of interprocedural analysis is $proc_{avg}$, the average cost of reanalyzing a procedure. Such an analysis involves, among other things, a recursive call to `build_descriptor`. In other words, any change in strategy that reduces the cost of the overall algorithm yields multiple benefits because it also lowers the cost of the recursive calls. To quantify the effect, chapter 8 presents the average measured value to handle a procedure for a variety of applications using more and less aggressive analysis strategies.

5.3.3 Descriptor Cleaning

The last operation is to clean the descriptor. The cost is expressed by the following formula:

$$clean = dirty\_var_{avg} * ops\_clean_{avg} * op\_cost_{avg}$$

The three quantities are the number of dirty variables in a descriptor, the number of symbolic operations needed to clean a dirty variable, and the cost for each of those operations. They are all averages for which the results section gives a range of values measured for several applications.

Chapter 6

Program Transformation

The purpose of code transformation is to take advantage of existing concurrency to subdivide the program or to expose additional concurrency. MAGNIFY can use the results of its analysis to perform a series of traditional loop-based transformations as well as more ambitious restructuring operations based on the *split* transformation.

Traditional parallelization operations focus on finding loops whose iterations are independent and on finding procedure calls that can be executed in parallel. MAGNIFY takes a higher level approach, treating a program as a set of interacting sub-computations. The program is transformed from its original form as a single monolithic computation and is decomposed into pieces as parallelization proceeds. The programmer directs this transformation by identifying computations that should be divided. MAGNIFY uses descriptors to carry out the transformations. When MAGNIFY can find sub-computations that are independent, it performs the division automatically. More aggressive transformations are performed by the *split* transform.

6.1 Internal Representation

MAGNIFY maintains the current state of the program in a variety of forms. WEB, a hierarchical graph-based internal representation, is used to capture the program's evolving control structure. The WEB consists of a graph and a set of additional variables which are used to allow nodes to communicate with each other. The scope of the variables is lexical; a variable that is defined at a given node can be referenced by the node itself and by nodes that it dominates. The WEB graph is made up of nodes and arcs. The nodes represent either individual pieces of sequential code or sub-graphs, while arcs represent control flow dependences.

The original sequential version of the program is represented by a single WEB node; as the control structure of the program is exposed and transformed, the result is recorded as a WEB graph. When the program is finally output in executable form, the WEB control structure is the source of parallelism that RTS manages. Any control structure that is left

inside the sequential FORTRAN code is invisible and inaccessible to RTS. However, when the run-time system manages control, it is also making scheduling decisions and allocating the computation across processors, so there is a significant level of overhead. It is important to reveal only the control that is useful for exposing concurrency or simplifying the behavior of the FORTRAN code. The case studies demonstrate the process in more detail.

6.1.1 WEB Nodes

Figure 6.1 gives a pictorial key to the different kinds of WEB nodes, which will be used in the case studies in Section 8.1.

The node types are:

- Sequential Operator — The simplest kind of node corresponds to a chunk of sequential FORTRAN code that is executed when all of the data it needs is available. The node is annotated with information about its inputs and outputs and with the analysis that has been performed by previous passes of MAGNIFY.

- Conditional — These nodes execute their associated sub-graph if a given conditional is true. The advantage to exposing the conditional is that it can be used in further transformations and RTS can avoid scheduling some code that does not need to be executed.

- Iteration — Although a piece of sequential code can perform its own iteration, when the loop is exposed it can be pipelined or individual iterations parallelized. The iteration node defines one or more induction variables, each with a corresponding range of values to iterate over, and a sub-graph. The sub-graph either carries out a single iteration of the loop or any contiguous range of iterations. The sub-graph initially is a single node, but can become arbitrarily complex if it is to be executed in parallel. The iteration ranges are executed as a nested loop with the first induction variable serving as the outer iteration. The remaining variables appear in the order they are listed, with the rightmost acting as the innermost loop in the nesting.

- Conditional Iteration — Many programs put conditional statements in loops so that the iteration skips some of the elements in the range. As is discussed in detail in Section 6.3 on the *split* transformation, the iterations that are skipped often reveal additional concurrency. This specialized version of the iteration node contains a masking expression in it that RTS uses to determine which iterations are to be executed.

- Map — A map node captures the most important source of concurrency in a program: loop iterations that can be executed in parallel. The node defines one or more induction variables and otherwise contains the same elements as the iteration node — a sub-graph to carry out iterations and a range over which to iterate.

- Conditional Map — The map node has a specialized version which is analogous to the conditional iteration node. It is useful because it exposes the conditional expression

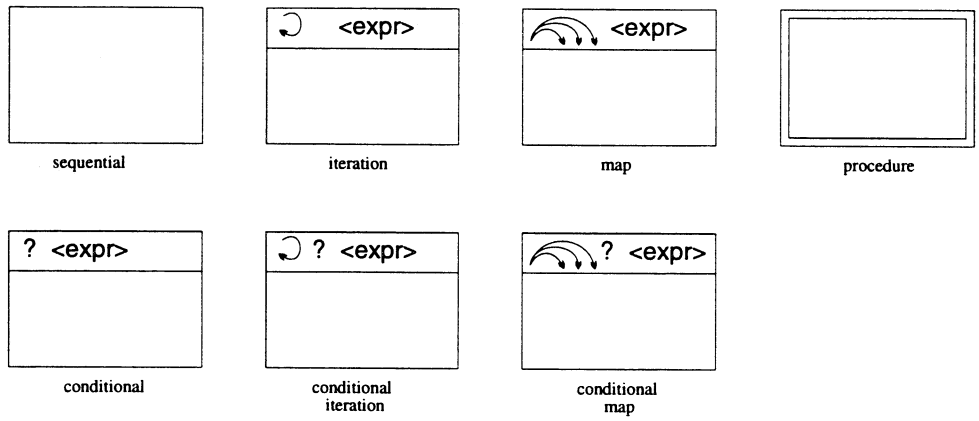


Figure 6.1: Node Types in WEB Representation

```
do i = 1, n                          ; initialization
  do j = 1, n
    A[i,j] = 0

do i = 1, n                          ; increment
  incr = f(total_incr)                   ; compute increment
  do j = 1, n                            ; do increment
    A[i,j] = A[i,j] + incr
  total_incr = total_incr + incr         ; update total increment

output_results(A, total_incr)        ; output
```

Figure 6.2: Source Code for Example

to the run-time system and thus allows it to improve the amount of concurrency and pipelining.

- Procedure — The procedure node is used whenever a sub-graph is useful even though there is no enclosing control structure like iteration. This introduces a level of abstraction to the graph which can reduce control complexity, simplify the name space of graph variables, and so forth.

To demonstrate how WEB expresses the control structure of parallel operations, we present a simple example based on the original source code shown in Figure 6.2.

Figure 6.3 shows a parallel version of the code fragment using WEB to represent the control structure. The parallel version begins by initializing the elements of A to zero. Because the computation is inside a map node, it can be carried out in parallel. The range of the node is the two-dimensional space $1..n, 1..n$.

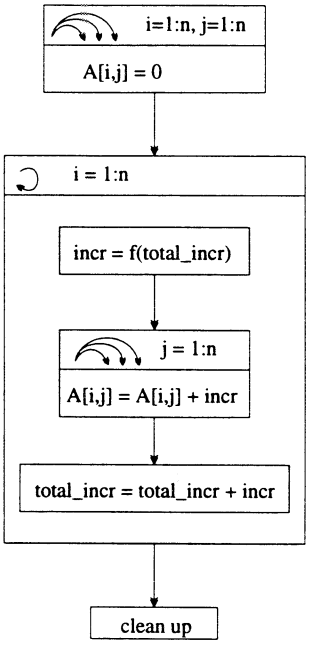


Figure 6.3: WEB Representation for Example

After initialization is complete, the next node is an iteration. It contains three WEB nodes in a sub-graph that it execute for n iterations, setting the induction variable `i` accordingly. During an iteration, each of the three sub-computations are performed in order. The first computes the increment value based on the value of `total_incr`. The second increments, in parallel, the value of each element in the i'th row of the array. The last operation adds the increment value to `total_incr`.

Once all the iterations are complete, the final WEB node, a sequential node, calls an output routine to report the results of the computation.

6.1.2 WEB Operations

To convert the original single WEB node into a graph that contains the needed control information, MAGNIFY applies a variety of operations to the evolving WEB representation. Some of the operations are in response to interaction with the programmer and others are done automatically. The major ones are:

- Peel Conditional — Given a block of code with a conditional expression that determines whether it executes, remove the conditional from the code and use a corresponding conditional WEB node instead.
- Peel Iterations — Remove the outer loop of a sequential block of code, exposing it as a WEB iteration node. When the node is converted into Delirium, the dependences

between iterations will be expressed directly. The iterations will be executed sequentially if they depend on one another but otherwise RTS is free to pipeline them for additional concurrency.

- Parallelize Outer Loop — Remove the outer loop of a sequential block of code, exposing it as a WEB map node. If there is an attached conditional that has been identified by MAGNIFY, the Conditional Map node provides a more accurate representation.
- Combine Map Nodes — Convert two successive map nodes into a single one that contains the code from both. The mapping operation in both nodes must be identical.
- Chop — Divide a given sequential node into two pieces, the second of which has a control dependence on the first. The transformation may require additional variables to be introduced in order to handle complex control interactions caused by conditionals and looping constructs.
- Split — Apply the *split* transformation (discussed in Section 6.3) to a pair of nodes which currently have a control dependence between them. The result is three nodes, one of which can be executed independently of the first two. The operation is used to expose concurrency between operations and to pipeline iterations of a loop.
- Wrap — Replace a sequential node with a procedure node. The procedure node's sub-graph contains only the original node.
- Compute Dependence — Modify the dependence relationships among a set WEB nodes based on the dependence information embodied in their data descriptors. This is used when a sequential operator has been chopped into pieces that are not sequentially dependent on one another.

Referring back to the example shown in Figure 6.3, the following set of transformations convert the original sequential code into its corresponding parallel WEB representation:

Chop code into `initialization`, `increment`, and `output`
Peel iteration from `increment`
Chop contents of `increment` into `compute`, `do_increment`, and `update`
Parallelize `initialization`
Parallelize `do_increment`

6.1.3 Transformations and Descriptors

Each WEB node is annotated with a summary descriptor. Any FORTRAN source code that is associated with the node is summarized by descriptors, as discussed in Chapter 5. Descriptors have been assembled for statements, loops, conditionals, basic blocks, and procedures.

When the code is transformed, either by modifying the WEB representation or by transforming code within a single node, existing descriptors are often rendered invalid. MAGNIFY

identifies the code and the WEB nodes that may have been affected and recomputes their descriptors. Typically the recomputation is localized, because most transformations are designed to leave surrounding code unaffected. None of the WEB transformations listed in Section 6.1.2, for example, affect the descriptors of WEB nodes that are not being directly modified.

6.2 Traditional Transformations

There are a large number of code transformations that have been developed by the compiler community; a detailed discussion of them can be found elsewhere [12]. MAGNIFY applies a variety of standard optimizations automatically and also permits the programmer to invoke them. The following are among those supported: loop reversal, loop invariant code motion [36], loop interchange [5], strip-mining [81], unrolling [2], loop fission [92], loop fusion, and tiling [1].

MAGNIFY summarizes the behavior of the original code with descriptors to determine whether a given transformation is legal. To perform an exhaustive analysis for loop-invariant code, for example, MAGNIFY begins by computing the descriptor for each statement in the loop. The following algorithm then identifies those statements which are invariant:

$changed$ = FALSE
$tocheck$ = set of statements in the loop
$LD = \bigcup V$, $V \in$ set of loop induction variables
repeat
  $\forall S$, $S \in tocheck$
    SD = descriptor for S
    if $(SD_{read} \cap SD_{write}$ or $SD_{read} \cap LD_{write})$ or $SD_{write} \cap LD_{write} \cap LD_{read}))$
      $LD = LD \cup SD$
    $tocheck = tocheck - S$
    $changed$ = TRUE
while ($changed$ = TRUE)

The algorithm iterates until convergence. As it examines the loop, it maintains *tocheck*, a set of statements that are currently believed to be invariant, and LD, a descriptor containing those memory locations that are modified during loop execution. The set initially contains all the statements that are in the loop and the descriptor contains only the loop induction variables.

On each iteration, the algorithm examines every member S of *tocheck* to determine whether S varies between iterations. That can happen because S uses values that it computed on a previous iteration — i.e., its read set SD_{read} has a non-nil intersection with its write set SD_{write}. Or, S might interact with the loop variant statements that have already been identified. If S reads a value computed by a loop variant statement, it is also loop variant.

Statements that are identified to be loop variant are removed from the set of invariants and their descriptors are added to the loop descriptor. When the algorithm converges, the statements remaining in *tocheck* are known to be loop invariant.

There are two features of this algorithm to be aware of. The first is that even though a statement is invariant, moving it may require renaming. The other consideration is that the algorithm operates at the statement level, so it will not identify arithmetic sub-expressions that are loop-invariant. This could be addressed by breaking complex statements into their simpler sub-components or by operating at expression rather than statement level.

A variant of the algorithm applies to user-invoked code motion. MAGNIFY will perform any requested transformation, but it first attempts to verify legality. It does so by calculating the descriptor of the code to be moved and the descriptor of the rest of the loop body. If the descriptor passes the test that the algorithm uses for invariance, the transformation is known to be legal.

Loop fusion also takes advantage of descriptors to determine legality. MAGNIFY computes the descriptor of each loop. If they do not interfere and the induction ranges are the same, the bodies of the loops can be merged.

Loop fission is similar; suppose that the goal is to divide a given loop into two pieces, dividing the statements of the loop body into two sets A and B. MAGNIFY computes the descriptors for both A and B over the entire induction range of the loop. If the descriptors do not interfere, the split is legal.

6.2.1 Associativity and Reduction

One of the complications in transforming scientific computations is that many optimizations change the order of execution of the floating point operations. Floating point computation is not associative, however, and in some cases an altered order may be numerically invalid. On the other hand, if the order may not be altered under any circumstances, many opportunities for parallel execution will be sacrificed.

MAGNIFY solves the problem by allowing the user to determine whether order is mutable or inviolate. The default assumption is that order can never be altered, but the user can specify via declarations in the code that (1) all operations in the program can be assumed to be associative or (2) operations on a particular variable can be assumed to be associative.

A particularly important special case is a *reduction* that is computing a running sum or product. A reduction is an arithmetic operation that updates the value of a variable using a computation that depends on its previous value. The following code fragment is an example:

```
total = 0

do i = 1, n
  total = total + A[i]
```

```
do j = 1, m
  total = total + B[j]
```

The variable `total` is used to assemble a running sum of the elements of two arrays. Without modifying the code, MAGNIFY is forced to delay execution of the second loop until the first is finished. If MAGNIFY is permitted to rename the variables, the two loops could be executed independently:

```
total1 = 0
total2 = 0

do i = 1, n
  total1 = total1 + A[i]

do j = 1, m
  total2 = total2 + B[j]

total = total1 + total12
```

Note, however, that the second code fragment performs the addition operations in a different order and hence may not yield the same result. Therefore, MAGNIFY can only rename reductions by using separate accumulation variables if the user has declared that reordering is permitted. In the example just given, if the variable `total` has a reordering declaration, then the renaming is legal. MAGNIFY can similarly rename arrays that are used to accumulate running sums or products.

6.3 Split-based Transformations

The most powerful transformations that MAGNIFY can apply are based on *split*. The goal of the operation is to expose concurrency that is not straight-forwardly accessible from independent loop iterations.

6.3.1 *Split*

The *split* transformation is used when two computations interfere. Figure 6.4 demonstrates a simple example, the division of H into three parts (assuming that we can treat addition as an associative operation). Let DG be the descriptor for G and DH be the descriptor for H.

Then (omitting induction variables):

- DG_{write} = { X[a,1..n]}

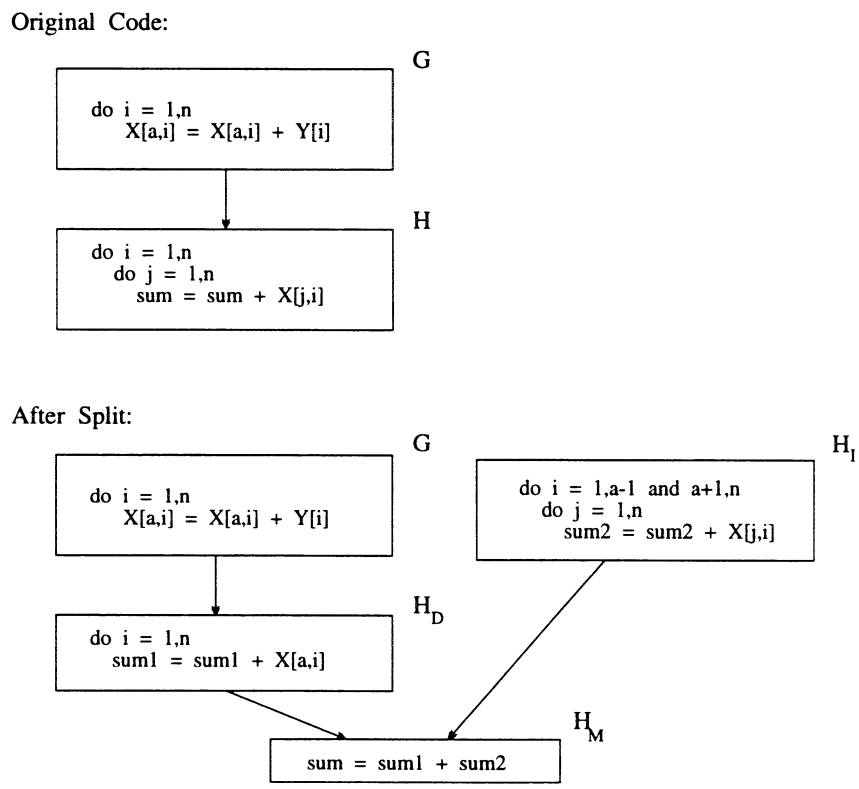


Figure 6.4: Simple Example of Split

- DG_{read} = { X[a,1..n],Y[1..n]}
- DH_{write} = { sum}
- DH_{read} = {X[1..n,1..n], sum}

Since $DG_{write} \cap DH_{read} \neq \emptyset$, the two code blocks interfere.

H is flow dependent on G because part of H reads the column of X modified by G. However, the dependence hides potential concurrency because most of H can be computed simultaneously with G; only the computation on column a must be deferred until G is done. Note that we have assumed, for simplicity, that a is known to be between 1 and n. H can be divided into the piece that must wait for G and the piece that need not; after both parts have finished, the results are merged together. This is exactly what *split* does on this example, dividing one computation into three parts based on its interactions with a second computation.

More generally, *split* takes as input a computation C and a descriptor D of a computation (usually a predecessor or successor), and attempts to find sub-computations in C that do not interfere with D. The effect of the transformation is to convert C into three computations: the dependent sub-computation C_D, the independent sub-computation C_I, and the merging sub-computation C_M. C_I contains the sub-computations that do not interfere with the computation whose descriptor is D and could be separated. C_D holds the rest of C, except for those sub-computations that rely on values now computed in C_I. The remaining sub-computations, along with any needed post-processing code, are put into C_M.

The split algorithm begins by subdividing C into *primitive computations*. Primitive computations are the blocks of code that are managed by the transformation; the choice of primitive computation determines the granularity of the split. We have chosen to consider basic blocks, function calls, and loops as primitive computations.[1] Thus in Figure 6.4, code block G has two primitive computations, a loop and a basic block, whereas block H has three. Subdividing basic blocks would yield a more aggressive split, at the cost of more communication overhead and more cleanup code in C_M.

The next step is to categorize each primitive computation into one of three sets based on its memory usage behavior. In defining the sets, we will say that a computation C *(directly) interferes* with a set S if and only if there exists $s \in S$ such that C interferes with s. We will also use the property of *transitive interference*; C transitively interferes with S if there exists a sequence of computations I_1, I_2, ... I_n, $n \geq 0$, such that C interferes with I_1, I_m interferes with I_{m+1}, and I_n interferes with S. We say C transitively interferes with S_1 *using* S_2 when I_1 through I_n are members of S_2. Notice that the property holds when C directly interferes with S.

[1] When profiling information indicates that a block of code is executed infrequently, the system may choose not to decompose a loop nest into its constituent computations.

The three categories of memory usage are:

- *Bound* – computations that interfere with D.
- *Linked* – computations that do not interfere with D directly but do transitively interfere with it.
- *Free* – computations that do not interfere with D directly or transitively.

The following algorithm takes C, a collection of primitive computations, and assigns each member to a usage category with respect to a descriptor D:

```
Bound = MaybeFree = ∅
for each c ∈ C
   if interfere(c, D)
      Bound = Bound ∪ {c}
   else
      MaybeFree = MaybeFree ∪ {c}
Linked = transitive_interfere(MaybeFree, Bound)
Free = MaybeFree

transitive_interfere(Initial, Target)
     Result = ∅, Testset = Target
     while (Testset)
          Newbound = ∅
          for each c ∈ Initial
              if interfere(c, Testset)
                 Initial = Initial − {c}
                 Result = Result ∪ {c}
                 Newbound = Newbound ∪ {c}
          Testset = Newbound
     return Result
```

The *transitive_interfere* procedure is given an initial set *Initial* and a reference set; it returns a set *Result* containing the members of *Initial* that transitively interfere with *Target* using *Initial*. The members of *Result* are removed from *Initial*. The procedure iterates to fixpoint; the number of iterations depends on the interference pattern between sub-computations. Each iteration looks at every sub-computation that remains in the *Initial* set and moves at least one into *Result* or the algorithm terminates. An upper bound on execution time is $O(n^2)$ *interfere* or *union* computations, where n is the number of sub-computations in C. In general, n is small because a computation that is worth moving will be fairly coarse grained.

Applying the algorithm to the computation H and the descriptor DG, all three primitive computations are in *Bound*, reflecting the fact that $DG_{write} \cap DH_{read} \neq \emptyset$.

As in this case, it is often possible to split the iterations of a loop in *Bound* into two sets, one of which interferes with D and one of which does not. It is legal to split iterations when we have nests of loops that are either independent or computing a reduction; they can be split by placing a conditional on the induction variable. By looking at the intersection of the loop descriptor with D, we determine whether a restriction on the induction variable yields a set of iterations that do not interfere with D.

If a loop is successfully split, we have two sets of loop iterations; one is still in *Bound*, but the other is either in *Free* or in *Linked*, depending on whether it interferes transitively with D. In our example, by restricting `i` so that it never equals `a`, we are left with a *Free* computation. Note that we are computing a reduction, so to split the iterations we must replicate the reduction variable and do the final reduction step in H_M.

A simple assignment of sub-computations to output sets could be done based on the three categories, by putting *Free* computations into C_I and all the rest into C_D. However, it is often important to handle *Linked* computations more carefully, so the set is sub-divided further. This involves one final interference property called *flow interference*, which unlike interference is not symmetric. A successor computation B has a flow interference from a predecessor computation A if $A_{write} \cap B_{read} \neq \emptyset$. This is equivalent to the notion of a *flow* dependence. Transitive flow interference is analogous to transitive interference. The new sub-divisions are:

- *NeedsBound* – *Linked* computations with a transitive flow interference from *Bound*.
- *GenerateLinked* – *Linked* computations from which *Bound* or *NeedsBound* has a transitive flow interference.
- *ReadLinked* – *Linked* computations that do not fall in either the *NeedsBound* or *GenerateLinked* sets.

In Figure 6.5 we show a more complicated example to demonstrate the types of *Linked* computations (but do not show transformed code). Suppose we wish to split T with respect to W's descriptor. For this example, we will consider the named computations to be indivisible.

E is *Free* because it does not have any relationship to W; B is *Bound* because it reads the values of array `X` that are written by W. The rest of the sub-computations are *Linked*. A generates values of `Y` that are used by the *Bound* computation B and hence is *GenerateLinked*. C is *ReadLinked*, because it also needs values from A. D is *NeedsBound* because it uses the value of `sum` computed by the *Bound* computation B.

Here is the algorithm that assigns each *Linked* computation to a set:

$Unrestricted = Linked$
$NeedsBound =$ transitive_flow_up($Unrestricted$,
$Bound$)
$GenerateLinked =$ transitive_flow_down($Unrestricted$,

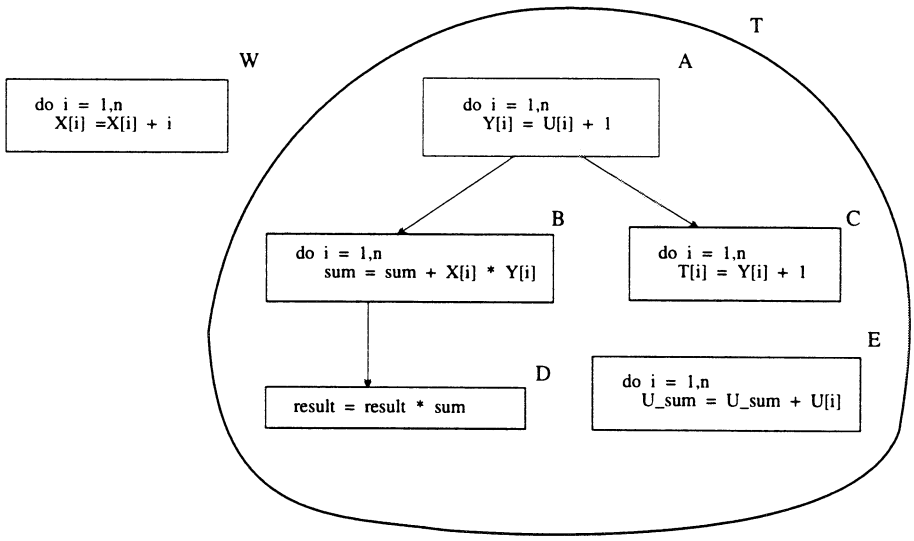


Figure 6.5: Enhanced Example of Split

$$Bound \cup NeedsBound)$$
$$ReadLinked = Unrestricted$$

The computation of $transitive\_flow\_\{up, down\}$ is analogous to $transitive\_interfere$, except that the calls to $interfere$ are replaced by calls to $flow\_interfere$. $flow\_interfere$ is not symmetric, so there are two versions of the transitive routine to check for interference from the candidate set ($down$) and from the target set (up).

$ReadLinked$ computations can be moved into the independent set at the cost of additional replication or movement. From the example, we could move the $ReadLinked$ computation C into T_I if we are willing to replicate A or to move both B and D into T_M. In general, to move a $ReadLinked$ computation r into the independent set, every computation s from which r has a transitive flow interference must also be put in that set.

In our current implementation, we use a heuristic to decide whether moving a member of $ReadLinked$ is worthwhile. The heuristic goes ahead with the move if both of the following are true:

- the number of floating point and integer computations in the code that is to be replicated can be calculated and it is below a threshold
- profiling data shows that the computation is expensive enough to justify moving it

Note that when a sub-computation created by splitting a loop nest is moved, some code must be added to the merge. If a nesting of independent loops is split, the merge glues the results together. If a reduction is involved, a temporary scalar or array must be created and the results stored in it (as was done in Figure 6.4 for the reduction variable **sum**). As a final step in merging, the last reduction is performed.

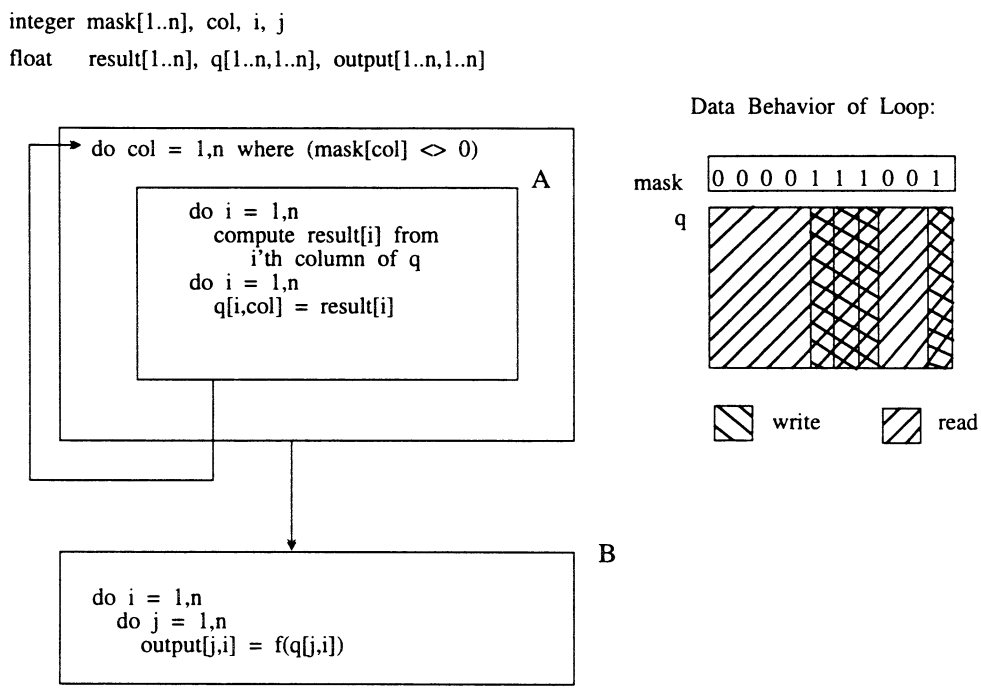


Figure 6.6: Sample Interacting Computations

6.3.2 Pipelining

In the discussion of the example in Section 7.1.3, three sources of concurrency are described, the first of which is a simple split. The other two, however, are pipelining transformations that go beyond *split*.

To pipeline a loop with *split*, first the descriptor for one iteration of the loop is computed. If the induction variable is i, D_{i-1}, the descriptor for iteration $i-1$, is computed. Then the loop body is *split* using D_{i-1}; the resulting independent computation does not interfere with iteration $i-1$. As iteration i is computed, the next iteration's independent computation can be executed concurrently. By transforming the code, that opportunity is exposed to the run-time scheduler. If deeper pipelining is desired, the descriptor for iteration $i-2$ can be computed, etc.

To give a concrete example, Figure 6.6 shows interaction among sub-computations. It is a loop whose induction variable `col` iterates over the columns of a data array `q`. If the associated element of the array `mask` is non-zero, the iteration performs the computation A. A reads all of `q` and modifies column `col`. After the loop is completed, B computes the array `output` from `q`.

Because of the conditional in A, the compiler cannot statically determine an efficient schedule for A. Depending on the values in that array, adaptive techniques may find an efficient schedule. However, on large numbers of processors efficiency may still be poor if there is not enough parallelism (i.e. too few mask elements are non-zero) or if the time to process

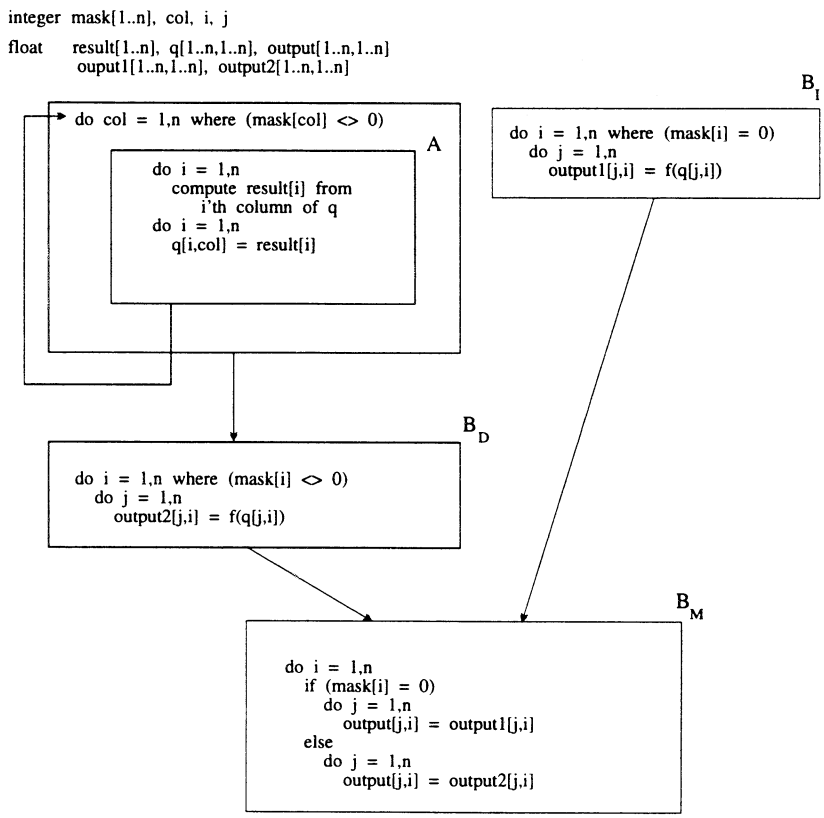


Figure 6.7: Code After Split

each column varies too widely.

Our approach is to transform the code to expose further concurrency; there are three sources that our strategy can reveal. The first of these is to divide B into three pieces, B_I, B_D, and B_M, where B_I processes the columns of q which are not touched by any of the instances of A and B_D processes the rest. B_M merges the results into a single output array. In Figure 6.7, we show the effect of applying this transformation.

The notation	is equivalent to
`do ... where <expr>`	`do ...`
`loop body`	`if (<expr>)`
	`loop body`

For clarity, we show an explicit merge of the two output arrays in B_M. In practice, merging can often be handled implicitly by the run-time system during data communication.

The second transformation we can apply is to pipeline one iteration of the `col` loop with subsequent iterations. We show the code with this further optimization in Figure 6.8. The body of the loop has been converted into three computations A_D, A_I, and A_M. A_D represents the code that is dependent on the previous iteration of the loop; the run-time

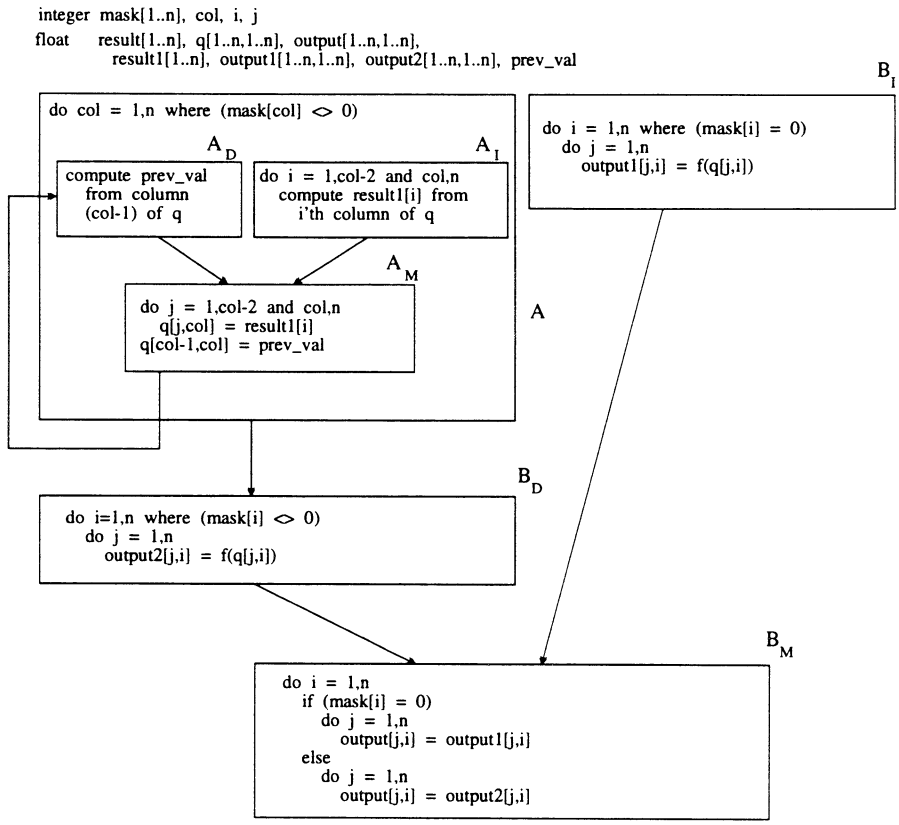


Figure 6.8: Code After Split and Pipeline

system waits for the previous iteration to complete before scheduling A_D. A_I, on the other hand, is independent of the previous iteration and can be scheduled concurrently. By weakening the synchronization constraint between iterations, we are able to pipeline them.

A_I computes the result vector for all but the missing column of `q` that comes from the previous iteration; A_D computes the one missing element into the variable `prev_val`. A_M takes the almost complete vector and the missing value and combines them into a single vector. Again, this kind of merge can often be done implicitly. Note that we use the form **`do var=<range> and <range>`** to denote a discontinuous sequence of values, rather than duplicating the entire loop for both ranges.

The third transformation which we could perform is to pipeline iterations of A with corresponding iterations of B_D, exposing a further source of concurrency.

The form of pipelining described in these last two transformations is different from loop pipelining optimizations defined elsewhere. Previous approaches fall into two classes. The first is *software pipelining* [73], which seeks to reduce overhead by reorganizing the code. The transformed loop performs fewer iterations but has a larger body that handles more than one of the original loop's iterations at once. This strategy works well in improving performance of a regular loop nest. The second approach, suggested by Balasundaran

and Kennedy [14], uses post and wait primitives to allow more than one loop iteration to execute concurrently. Since this strategy imposes a fixed synchronization discipline, it does not admit adaptive scheduling techniques.

6.4 Related Work

There is a wide variety of code transformations that have been investigated by the compiler community. MAGNIFY incorporates several of the loop-based transformations like interchange [5], strip-mining [81], and tiling [1, 52, 74, 124]. There are dozens of other transformations discussed in the literature [12].

There are also a number of transformations that change the way the loop is scheduled, such as software pipelining [73], loop unrolling [44], and unroll and jam [28]. These techniques change the loop structure, perform more than one iteration of a loop at once, or otherwise alter the code to expose additional instruction level parallelism or reduce the number of branching instructions executed.

However, the disadvantage to these transforming and scheduling algorithms is that they are limited to a single loop nest. In order to look beyond one nest, compilers can fuse adjacent loops [4], coalesce them [96, 97], inline function calls within the loop [4, 16, 110], or push loop structure through a call site [59, 106]. However, these techniques simply yield a larger individual loop nest to work on. They cannot address the interactions between logically separate computations that should remain independent. Split is designed to address this weakness. It permits more aggressive pipelining and reveals concurrency between subcomputations.

Because most parallel compilers produce an SPMD program rather than a structure form that is interpreted by a run-time system, they are tightly restricted in the interactions that they can usefully exploit. Since MAGNIFY can rely on RTS to handle scheduling issues, its job is to expose as much concurrency as is practical to exploit.

Although split has the advantage of supporting higher-level transformations than the loop-based approaches, the added flexibility has a disadvantage. Simply determining a good set of loop transformations to apply to single loop nest is very difficult, as there are typically several that are legal and the order in which they are applied can be very important.

The effort to organize the diverse group of transformations into a framework is an ongoing research problem. One 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.

A framework that is being actively investigated is based on unimodular matrix theory [19, 123]. 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.

Sarkar and Thekkath [109] 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 [102] 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.

Unfortunately, split offers too much generality to incorporate neatly within such a framework. However, while the frameworks offer some hope on limited parallelism like superscalar processors, they are far from being able to manage real applications on large numbers of processors. By relying on some programmer intervention, the Delirium environment is able to support the efficient execution of production applications on large numbers of processors. As is demonstrated in Chapter 8, the interaction between computations offers a great deal of additional concurrency that the more limited approach is not able to address.

Chapter 7

Expressing the Decomposition

Once MAGNIFY has finished transforming and decomposing the program, it converts the WEB representation into a form that RTS, the run-time system, can understand. This consists of a set of FORTRAN code fragments with tunable granularity, a Delirium framework that ties the fragments together, and an annotation in Dossier. This chapter describes these languages and shows how a computation is expressed in them.

MAGNIFY decides which loops to parallelize, but it does not choose a specific granularity. It outputs a code fragment that is given induction variable bounds as arguments, allowing RTS to tune the behavior of the code using information about the current state of the system. The following sections describe Delirium and Dossier in detail, using a running example to show how they express the parallelization of a code fragment.

7.1 Delirium

The heart of the Delirium programming environment is Delirium itself, a *coordination language*. A coordination language does not express the basic computation being performed, but rather ties together a set of fragments into a coherent whole. It describes how the fragments relate to each other and the information that they pass amongst themselves.

There are two basic approaches to using a coordination language; one is to scatter calls to a collaborating library throughout the text of the program, and the other is to express the framework of the application in the coordination language. Delirium uses the latter strategy and is therefore an *embedding* coordination language. This strategy allows Delirium to be used directly by a programmer and to be the intermediate form that is output by MAGNIFY. The benefits of embedding coordination languages for the programmer have been argued in detail elsewhere [85].

MAGNIFY uses Delirium to communicate with the run-time system. Because Delirium encapsulates the entire structure of the program and its interactions, the Delirium compiler can convert Delirium into a set of pre-compiled scheduling templates that allow decisions

to be made rapidly at execution time. This process is discussed in more detail below and at much greater depth in Steven Lucco's PhD dissertation [84].

Delirium is a single-assignment language that directly expresses the coordination strategy for a group of *operators*. An operator is a sub-computation expressed in a source language like C or FORTRAN that performs some operations on its inputs and yields a set of outputs. Here is a simple program that demonstrates fork/join parallelism:

```
a_start=init_fn();

PARALLEL_DO
  a=convolve(a_start,0);
  b=convolve(a_start,1);
  c=convolve(a_start,2);
  d=convolve(a_start,3);
END PARALLEL_DO

return(term_fn(a,b,c,d));
```

The `PARALLEL_DO` construct indicates that all the statements in it can be executed simultaneously. Otherwise, all code is executed sequentially as in a traditional programming language. Hence the call to `init_fn` must complete before any of the calls to `convolve`. Similarly, the call to `term_fn` will not begin until the entire `PARALLEL_DO` construct finishes execution.

In Delirium one could express this computation as:

```
let
   a_start=init_fn()
   a=convolve(a_start,0)
   b=convolve(a_start,1)
   c=convolve(a_start,2)
   d=convolve(a_start,3)
in term_fn(a,b,c,d)
```

The calls to `convolve` use the value `a_start` as an argument and therefore they will not be executed until the function `init_fn` has completed execution. The instances of `convolve` can be executed in parallel because none of them depend on each other's return values. The function `term_fn` uses the values returned by the calls to `convolve` and will not be executed until they are completed.

Because Delirium is a single assignment functional language, a program's communication topology can be derived from the data dependences between its operators. This advantage of single assignment languages is well known [88]; the contribution of Delirium is that it harnesses this property to create clean expression of coordination, making Delirium suitable both as a programming language and as an intermediate form for a parallelizing environment.

7.1.1 The Basic Language

One advantage of functional languages is their fundamental simplicity. Only a few constructs are necessary to provide a rich programming framework. This section describes the basic structures in Delirium (see Figure 7.1). The language constructs are:

1. atomic values – integers, strings, floats, arrays, functions
2. multiple values – a collection of atomic or multiple values that can be grouped together, decomposed, and used as return values
3. let bindings – a binding can be a single value, a decomposition of a multiple value package, or a function definition
4. conditionals
5. iteration
6. mapping
7. operators
8. functions
9. function and operator application

A Delirium program consists of a group of functions, one of them called `main`. Here is a very simple example:

```
main()
  start_execution(42)
```

This Delirium program relies on a single FORTRAN operator called `start_execution`. When the program is executed, it invokes the operator and gives it one argument, the integer 42.

More interesting Delirium programs typically define a set of functions. A function definition consists of a name, a formal argument list in parentheses, and a body. The body is an expression. When the function is applied to a set of argument expressions, each formal parameter is bound to the value of the corresponding argument and the body of the function is evaluated. The value returned by the function is the value of the body expression.

For example, the following Delirium program defines a function and applies that function to a pair of argument expressions:

```
main()
  the_function(1,-2.3)

the_function(a,b)
  compute_something(a,b)
```

program ::= *function-list*
function-list ::= *function* | *macro* | *function-list function*
| *function-list macro macro* ::= '`macro`' *id args expr*
function ::= *id args expr*
args ::= '`(`' `)` | '`(`' *arg-list* '`)`'
arg-list ::= *arg-list* '`,`' *id* | *id*
expr ::= *conditional* | *let-stmt* | *iteration* | *mult-value* | *func-app*
| *macro-call* | *prim-expr* | *mapping*
let-stmt ::= '`let`' *bindings* '`in`' *expr*
bindings ::= *bindings binding* | *binding*
binding ::= *var-binding* | *mult-var-binding* | *func-def*
var-binding ::= *id* '`=`' *expr*
mult-var-binding ::= '`<`' *var-list* '`>`' '`=`' *expr*
func-def ::= *id args* '`=`' *expr*
var-list ::= *var-list* '`,`' *id* | *id*
iteration ::= *iterate-while* | *iterate-range*
iterate-while ::= '`iterate`' '`{`' *iter-bindings* '`}`' '`while`' *expr* '`,`' '`result`' *expr*
iterate-range ::= '`iterate`' *expr-range iter-cond* '`{`' *iter-bindings* '`}`' '`result`' *expr*
iter-cond ::= '`<`' *expr* '`>`' | ϵ
iter-bindings ::= *iter-bindings iter-binding* | *iter-binding*
iter-binding ::= *id* '`=`' *expr* '`,`' *expr*
mapping ::= *map-across* | *map-across-where*
map-across ::= '`map_across`' '`(`' *expr-list* '`)`'
map-across-where ::= '`map_across_where`' '`(`' *expr-list* '`)`'
conditional ::= '`if`' *expr* '`then`' *expr* '`else`' *expr*
prim-expr ::= *integer* | *string* | *id* | *float* | '`*`' | *array-ref* | *range*
mult-value ::= '`<`' *expr-list* '`>`'
expr-list ::= *expr-list* '`,`' *expr* | *expr*
prim-expr-list ::= *prim-expr-list* '`,`' *prim-expr* | *prim-expr*
expr-range ::= '`(`' *expr* '`:`' *expr* '`)`' *func-app* ::= *id app-args*
macro-call ::= *id app-args*
app-args ::= '`(`' '`)`' | '`(`' *app-arg-list* '`)`'
app-arg-list ::= *app-arg-list* '`,`' *expr* | *expr*
array-ref ::= *id* '`[`' *prim-expr-list* '`]`'
range ::= *prim-expr* '`:`' *prim-expr*

Figure 7.1: Basic Delirium Grammar

Functions are *first class* objects, meaning that they may be passed as arguments, bound to variables, or returned as values. All of the language constructs listed above are value-producing expressions and any of them can appear in the body of a function.

If a function application supplies fewer arguments than are expected, according to the function definition, the application is *curried*. For example, a function of two arguments `a` and `b` can be applied to a single argument `x`. The result is a function of one argument that has the binding of `a` to `x` bound within it. When this new function is applied to an argument `y`, the result is the same as that of applying the original two-argument function to `x` and `y` at once.

Atoms

The simplest expressions in Delirium are either constants or binding references. Constant types are integers, strings, floating point numbers, functions, and arrays. Some examples:

```
an_integer_operator(34)
a_string_operator(``hello'')
a_float_operator(-345.23)
a_function_operator(f)
an_array_operator(A[1:n])
```

If a variable appears, the name must be bound in the surrounding environment, using standard lexical scope rules.

Conditional Expressions

The syntax for the conditional is:

```
if <expr1> then <expr2> else <expr3>
```

where `<expr1>` must evaluate to an integer. As in C, a zero value is false and a non-zero is true.

Multiple Values

Because functional languages eliminate global variables, it is especially useful for them to allow multiple value return. This is even more true in Delirium, which depends on destructuring operators to divide large data structures into many pieces that can be processed in parallel. The syntax is simple:

```
<exp1,exp2,exp3 ...>
```

where the comma separated values may be any valid expression. This package can be passed around as a single value; its components can be separated only within a `let` binding as described below.

Let Bindings

The let binding associates identifiers with expressions. Its syntax is:

```
let
  <var> = <value>
   ...
in <expression>
```

A binding can also be used to subdivide a multiple value:

```
let
  <a,b,c> = op_that_yields_multiple_value(12)
  <d,e> = c
in do_something(a,b,c,d,e)
```

In this example, the variable `c` must be a multiple value package or an error will be signaled at run-time. To manipulate packages, we have provided a set of operators like `car`, `cdr`, `cadr`, etc. These are simple to write and a different set could be defined quickly by a Delirium programmer if desired.

The last kind of let binding creates a function, eliminating the need for a "lambda" construct by making it easy to define functions where they are needed. The syntax is:

```
let
  <name>(<arg>, <arg>, ...) = <body>
   ...
in <expression>
```

Within the expression making up the body of the let, a new function has been defined. Nested bindings are scoped in standard lexical fashion.

Each of the expressions on the right hand side of the equals sign can refer to any of the variables or functions bound in that let statement (some dialects of LISP, like SCHEME [103], call this construct a *letrec*). For example, this is a legal code fragment:

```
let a = salt(15)
    salt(x) = if imdone(x) then 10
                           else salt(next_value(x))
in figure_something_out(a)
```

Of course, if **imdone** and **next_value** are not correctly written, this expression may not terminate. A value will be returned if **imdone(15)** is true or if **imdone(exp)** is true, where **exp** consists of some finite number of **next_value** applications on to the original value of 15.

Mutually recursive functions are also legal, as in:

```
let salt(x) = if imdone(x) then x
                           else pepper(next(x))
    pepper(y) = if hesdone(y) then y
                              else salt(next(y))
in
   salt(give_start_value())
```

Example

The following example is a code fragment taken from an early version of the Delirium compiler. Its control structure is expressed in Delirium, using only the language constructs that have been introduced so far. The underlying operators were written in C. A detailed discussion of that compiler appears elsewhere [111].

```
main()
  let <init_tree,macro_set> = compile()
      ops = read_operator_info()
      analyzed_tree = analyze(init_tree,ops,macro_set)
  in output_tree(analyzed_tree)

tree_walk_update(tree,node_type,tree_op,extra)
  let <ut1,ut2,ut3> = tree_up_chop(tree,node_type,tree_op,extra)
  in tree_merge(tree_up_op(ut1,node_type,tree_op,extra),
                tree_up_op(ut2,node_type,tree_op,extra),
                tree_up_op(ut3,node_type,tree_op,extra))

compile()
  let <c1,c2,c3> = split_lexemes()
  in forge_parse_tree(partial_parse(c1),partial_parse(c2),
                      partial_parse(c3))

analyze(tree,ops,macro_set)
  let marked_tree = tree_walk_update(tree,ANY,mark_extra,ops)
      pruned_tree = tree_walk_update(marked_tree,ANY,prune_extra,SAFE)
      final_tree = tree_walk_update(pruned_tree,ANY,mark_extra,macro_set)
  in tree_walk_update(final_tree,ANY,prune_extra,FINISH)
```

This fragment parses an input file into an internal tree-shaped representation and analyzes that tree. The function `main` controls execution, beginning with a call to `compile`. `compile` invokes the operator `split_lexemes` to read the source code and divide it into three sets of lexemes. The operator returns the three values in a multiple return value package that is decomposed by the `let` statement into independent values.

Each of the lexeme sets is parsed independently and the results are combined into a single tree by `forge_parse_tree`. The return value of `forge_parse_tree` is a multiple value package containing two entries — a tree and a set of macros. The package is decomposed into its component values by the let statement, assigning the tree to `init_tree` and the set of macros to `macro_set`.

The next step is to call another input routine, `read_operator_info`, which opens a file containing a set of operator descriptions, parses them, and returns a description of them that is bound to the variable `ops`. Then `analyze` is called to analyze the parse tree by making four updating passes over the parse tree. The result is passed to `output_tree`.

The updating passes use the function `tree_walk_update` to express a fork-join operation on the tree. The function accepts four arguments: a tree, a node type, an operator, and an extra value. It uses an operator called `tree_up_chop` to divide the tree into three sub-trees. The operator `tree_up_op` is invoked on each sub-tree, and its responsibility is to apply the operator `tree_op` to each sub-tree. The tree operator is given a sub-tree, the node type to operate on, and the argument `extra`. When each operator has finished, the three sub-trees are merged back together into a single one that serves as the return value for the function.

The fragment has two sources of parallelism in it. Some of the computations, like `compile` and `read_operator_info`, are known to be independent because they do not rely on each other's return values. The second opportunity for concurrency is in the tree update function; the three sub-trees are all handled independently. Because it is a single-assignment language, the Delirium compiler can quickly identify such opportunities for concurrency by analyzing the program's control structure.

In addition to the language constructs described above, there are two others that are used extensively by MAGNIFY when it decomposes scientific applications.

Iteration

The first is iteration, of which there are two basic types in Delirium. The first iterates until a particular condition is true and its syntax is as follows:

```
iterate
   {
      <iter-variable> = <initial-value-expr>, <update-value-expr>
      ...
   }
while <expression>, result <expression>
```

Evaluation begins by binding each iteration variable to its initial value. If the conditional expression is true, each iteration variable is bound to the value of the update expression. The conditional expression is re-evaluated. The looping ends when the `while` expression is false; the final value of the iterate construct is the value of the expression appearing after `result`.

The second form is used to express iteration across a range of values (typically a subset of the elements of an array). This notation, strictly speaking, is redundant because it could be captured within the more general form. However, the notation allows the Delirium compiler to perform more sophisticated analysis in identifying opportunities for pipelining.

The syntax is:

```
iterate ( <variable> = <low-expr>:<high-expr> )
          Optional: < <cond-expression> >
   {
      <iter-variable> = <initial-value-expr>, <update-value-expr>
      ...
   }
result <expression>
```

Both `<low-expr>` and `<high-expr>` must yield an integer value. The iteration will proceed by assigning each value in the range to `<variable>`. If `<low-expr>` has the larger value, no iterations are performed. The optional conditional expression limits the set of values in the range — the expression is evaluated on each potential value and the function application is only performed if the expression is true. The symbol `*` may appear in the expression to represent the element value being evaluated. The Delirium compiler can often take advantage of the fact that the conditional is exposed in the Delirium code to output better scheduling templates that improve concurrency and pipelining.

Here is a Delirium code fragment that uses the construct:

```
iterate (x = 1:n) <mask[*] = 1>
  {
    computed_val = A, do_iter(computed_val)
  }
result computed_val
```

The fragment iterates from one to `n`, skipping iterations that do not have a value of one in the corresponding element of the `mask` array. The first iteration takes the value `A` and applies the function `do_iter` to it. Subsequent iterations repeatedly apply `do_iter` to the result of the previous one. At the end, the final computed value is returned.

Mapping

One of the most common sources of parallelism found by MAGNIFY is a computation that is applied to a range of elements in an array. Delirium supplies two mapping operators that express this common control structure in a form that can be handled efficiently by the Delirium compiler. The first is called `map_across` and has the following syntax:

```
map_across (<func-expr>, <array>, <low-expr>:<high-expr>, ...  )
```

The first argument to `map_across` is `<func-expr>`, which must evaluate to a function. The second argument is the array that is being used for the computation. The third is the range of values, where range expressions yield integers. The effect of the construct is to apply the function to each value in the array that falls within the range. The function that is invoked must take as its first three parameters: (a) the array being operated on, (b) the lower and bound of the range of elements to compute, and (c) the upper bound of the range. It may also be provided with additional values that were supplied to `map-across`.

The idea behind `map_across` is that it allows the Delirium compiler and run-time system to choose the best strategy for dividing a group of independent computations. If there is abundant parallelism elsewhere, the entire group might be computed sequentially. At the other extreme, each individual computation could be allocated to a different processor. The function that is supplied to `map_across` is prepared to handle any range of iterations, allowing a broad range of allocation strategies to be used.

Figure 7.2 demonstrates how `map_across` is used. The function `do_iters` accepts a range of values and performs the computation in the original loop over that range. `map_across` is given the original range and an operator that can handle sub-ranges; it decides how to break up the range into intervals. This is one of the most common ways that RTS chooses granularity.

The second mapping operator is similar to `map_across`; the only difference is that it includes a conditional expression indicating that some array elements are to be skipped. Exposing the conditional to the Delirium compiler allows it to apply various optimization algorithms that improve efficiency.

The syntax of the construct is:

```
map_across_where ( <func-expr>, <array>, <low-expr>:<high-expr>,
                   <cond-expr>, ...  )
```

The function will only be mapped to values in the range for which the conditional expression is true. The special symbol * may appear in the expression to represent the range value being considered. The following Delirium code fragment is an example:

```
map_across_where(compute_val,A,1:n,mask[*] = 1,c)
```

```
do i = 1, n
  A[i] = A[i] + c
```

(a) The original code

```
do_iters(A,a,b,c)
  do i = a, b
    A[i] = A[i] + c
```

(b) The operator for the Delirium version

```
map_across(do_iters,A,1:n,c)
```

(c) The Delirium code

Figure 7.2: Use of `map_across` to Apply Operator Across Range

The function `compute_val` will be applied to elements one through `n` in the array `A`, as long as the corresponding element in the array called `mask` is one. The value `c` will be passed to `compute_val` as an extra argument.

Mapping and iteration are often used together to express a computation:

```
iterate
  {
    array = A, map_across(handle_elements,array,1:m,c)
    convergence = 0, test_converge(array)
  }
while (convergence = 0), result array
```

This fragment repeatedly performs a mapping operation on the array `A` until the operator `test_converge` returns a non-zero value. During each iteration, the `handle_elements` operator is applied to the range of array elements from 1 to m. When the computation converges, the final computed values in the array are returned.

7.1.2 Coordination Structures

In addition to its use as an intermediate form by RTS, Delirium can also serve as a programming language for building parallel applications. The basic features of the language that have been described above are capable of expressing a wide range of parallelization strategies — including all of the transformations that MAGNIFY performs.

However, certain complex parallel control structures can be handled particularly efficiently if they are expressed in a specialized form. That was the motivation behind *coordination structures*, an enhancement to the basic Delirium language that is well-suited to capturing data parallel operations.

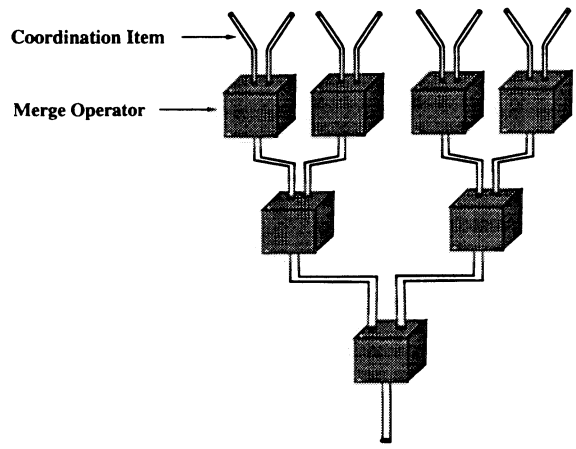


Figure 7.3: Coordination Structure for Mergesort

Coordination structures express a multi-stage pipeline; at each stage in the pipeline, a function is applied to the data flowing through each member in the collection of data pipes. Following a function application, the order of items (data pipes) within the coordination structure may be permuted to create the appropriate data organization for the next stage of the pipeline.

For example, a useful primitive for many parallel algorithms is binary reduction. A binary reduction takes N data items and applies some associative binary operation to successive pairs, yielding a group of $N/2$ results. The same operation is performed repeatedly until there is only one value left. Many algorithms are based on binary reduction, including, for example, merge sort. The pipeline for merge sort is shown in Figure 7.3 and consists of $log(N)$ applications of the merge operator. If the original N values flow into the pipeline as a vector of pipes, the first step is to divide the pipes into $N/2$ pairs. Next, the program applies an instance of the merge operator to each of these pairs in parallel. One pipe flows out of each merge operator, so the cross section of the pipeline has been reduced to $N/2$ coordination items. The grouping and function application are done repeatedly, until at the end only a single pipe flows out of the pipeline and it contains the sorted list.

Coordination structures allow the programmer to create such multi-stage pipelined computations and treat them as first-class objects in the language. They can be applied to data, passed as arguments, and composed. These operations permit the Delirium programmer to achieve very high efficiency on structured data parallel computations. However, coordination structures are not needed to express the parallelization strategies supported by MAGNIFY and are therefore not discussed further in this dissertation. For the interested reader, a detailed description is available elsewhere [86].

7.1.3 MAGNIFY Example

We will use the following code fragment to demonstrate how MAGNIFY uses Delirium to express parallel computations. The variables `n` and `c` have been set elsewhere to a value unknown at compile time and initial values have been assigned to `A`, `B`, and `total`.

```
integer iter, n, i, j, k
real A[1:n,1:n], B[1:n,1:n], total[1:n], c

do iter = 1, n                          ; "iter" loop
  do j = 1, n                           ; nest 1
    do i = 1, n
      A[i,j] = A[i,j] + B[i,j] * c

  do k = 1, n                           ; nest 2
    total[k] = total[k] + B[iter,k]
```

The outer loop (termed above the `iter` loop) invokes two loop nests (nest 1 and nest 2) on each major iteration. Both nest 1 and nest 2 are fully independent within themselves; in both cases, the different iterations can be executed in parallel because there are no loop-carried dependences.

The independence of the first loop is clearly demonstrated by examining the descriptor for the statement in the loop. The descriptor for iteration i, j captures the fact that it only writes memory location $A[i, j]$. The intersection of the write set with the descriptor of every other iteration is null, indicating that the loop is parallel. Similarly the descriptors capture the independence of the second loop. Because the descriptors are similar to ones shown earlier in Chapter 3, we do not present them here.

Both loops are also independent of each other. MAGNIFY would detect the lack of conflict by computing the descriptor for each loop nest and calculating the intersection of each one's write set with the other descriptor. Because the result is empty in both cases, the two loops can be executed in parallel.

However, the `iter` loop iterations are not independent; both nests 1 and 2 compute values in one major iteration that are used during the next one. The intersection of the descriptors for two subsequent iterations reveals that the elements of `A` and `total` are modified in the first and then used in the second.

The most straight-forward way to parallelize the construct is to treat both of the loop nests separately. If the value of `n` is significantly larger than the number of processors available, there is no reason to parallelize the inner loop in nest 1. Instead, the outer loop should be strip-mined and the strips distributed across the processors. Strip-mining [81] is a very common optimization, particularly in vectorizing compilers. It divides an iteration range into fixed sized strips. Here is a transformed version of the outer loop of the first nest after strip-mining has been applied, using a strip size of 64:

```
do j = 1, n, 64
  do j1 = j, j+63
    do i = 1, n
      A[i,j1] = A[i,j1] + B[i,j1] * c
```

The proper strip size for a parallel machine depends on the number of processors and the degree to which they are already occupied with other computations. Nest 2 should also be strip-mined.

Figure 7.4 demonstrates these parallelization strategies using the WEB representation. The fragment starts out as a simple sequential block of code. The next step is to peel the outer loop away from the FORTRAN code and convert the single WEB node into an iteration node. The second step subdivides the code within the node into two pieces, each containing one of the loops. The third step takes advantage of the parallelism in each of the two pieces, peeling the outer loop and converting the node into a map operation. Finally, the fact that both pieces are independent of one another allows them to be placed next to one another without any sequential execution constraint. The WEB representation reveals the opportunity to parallelize each piece and also the two pieces with respect to one another.

Figure 7.5 shows the code that MAGNIFY would produce from the transformed WEB representation. It includes both the FORTRAN operators and the Delirium text that ties them together. The outer iterations of the loop have been captured in a Delirium `iterate` structure. Each of the inner loops is expressed as a mapping across the iteration space.

When a parallel loop represented in WEB by a map node is converted into Delirium, Dossier, and FORTRAN, MAGNIFY outputs them as operators that have tunable granularity. In the example, each loop in the original code is represented by a procedure that accepts a range of values. The run-time system determines how long each invocation of the operator will execute by defining the upper and lower bounds of the range. The values are chosen based on the execution time of the node and on the current state of the system.

If, for example, the operator has demonstrated (or is annotated to reveal) that it will execute very few operations per iteration, the run-time system will try to reduce scheduling overhead by specifying a large range. If the current amount of parallelism available during application execution is too low to keep all the processors busy, RTS will try to compensate by reducing the range. The algorithms for controlling the granularity dynamically are briefly introduced in section 7.3 and described in more detail elsewhere [84].

The Delirium code, because it reflects the dependences between code fragments, exposes three sources of concurrency. The first and most obvious is the parallelism of each loop due to the `map_across` operator. The second is that the two loop nests do not depend on one another. If there are processors available that are not occupied, RTS can start computing the second nest while the first is still executing. The algorithms that RTS uses to accomplish this strategy are outlined in Section 7.3.

The third opportunity to execute in parallel comes from pipelining subsequent `iter` loop iterations. Once nest 1 for iteration `iter` $= X$ is complete, nest 1 for iteration `iter` $= X+1$ can begin immediately even though nest 2 is still being computed. The Delirium compiler

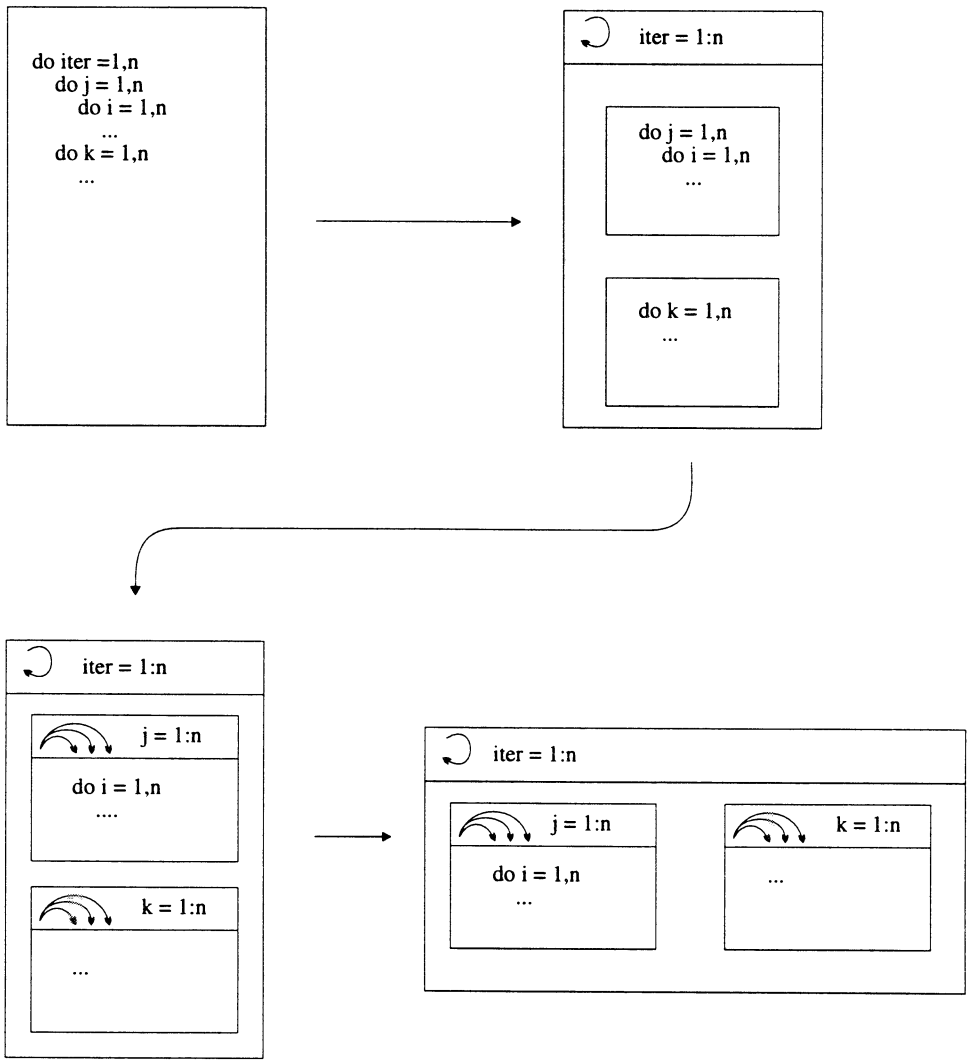


Figure 7.4: Parallelization of code fragment using WEB representation

```
iterate (iter = 1:n)
  vargroup1 = <A>,      map_across(loop1,1:n,vargroup1,B,c,n)
  vargroup2 = <total>, map_across(loop2,1:n,vargroup2,B,iter,n)
result <vargroup1,vargroup2>
```

(a) Delirium Control Program

```
subroutine loop1(A,B,c,n,b,t)
integer n,b,t
real A[n,b:t],B[n,b:t],c

do j = b, t
  do i = 1, n
    A[i,j] = A[i,j] + B[i,j] * c

end

subroutine loop2(total,B,iter,n,b,t)
integer iter,n,b,t
real total[b:t], B[n,b:t]

do k = b, t
  total[k] = total[k] + B[iter,k]

end
```

(b) FORTRAN operators

Figure 7.5: Decomposed Version of Example

can unroll the iteration statement to expose the pipelining.

7.2 Dossier

While Delirium captures the interactions between sub-computations, it does not give enough information to generate an efficient scheduling strategy. In the function `loop1`, `A` is part of a (potentially large) array while `n` is just an integer. RTS should try to avoid moving `A` and should certainly not pay attention to the location of `n`, but the Delirium text does not provide enough information to make that decision.

The Dossier annotations make it clear that the best choice is almost certainly to replicate `n` to every processor at the beginning of the computation and to attempt to perform `loop1` iterations on the processor that currently holds the appropriate part of `A`.

In order to help the run-time system, Dossier makes two kinds of information available:

- For each variable that represents a data structure, the structure's shape and size.
- For each computational operator, a characterization of its execution cost.

As mentioned previously, every WEB node has an associated descriptor that summarizes its memory usage behavior. When the WEB representation is converted into Delirium, MAGNIFY uses those descriptors to accomplish the translation.

Each FORTRAN variable referenced by a WEB node is described by a triple in the associated descriptor. To produce the appropriate Delirium and Dossier, MAGNIFY conceptually generates a Delirium variable for each FORTRAN variable. A simple optimization reduces the number of variables: sets of scalars that are passed from one WEB node to another are grouped together into a package. MAGNIFY uses a straightforward greedy algorithm to identify the groupings.

Once the Delirium variables have been assigned, MAGNIFY annotates each one with a description of its size. The description is condensed from the corresponding triple because triples supply more detail than Dossier can express. Dossier and triple expressions are very similar but the former are much simpler — the additional accuracy provided by the triple is discarded during the conversion.

After the descriptors are used to compute the data size annotations, execution cost estimates are computed by a set of heuristics as discussed in the next section.

7.2.1 Cost Heuristics

In order to help RTS decompose computations, MAGNIFY attempts to annotate operators with their execution cost. The estimate can either come from profile data, from heuristic analysis, or from both. If MAGNIFY is provided with one or more `gprof` [57] profiles for the application, it uses the data in computing estimates. For each operator, MAGNIFY identifies any procedures from the original source code that are contained in it. The average and the variance of the measured execution times for those procedures are used in the cost estimate.

Next, MAGNIFY examines the remaining code in the operator. Using a simplified version of *trace scheduling* [49], it identifies the paths that execution can take through the code. Each node in the CFG is annotated with an estimate of the number of floating point operations (flops) on the longest path to that node from the entry node. If loop bounds are available, they are used; if not, loops are assumed to execute between 50 and 100 times with uniform distribution. The value that annotates the exit node of the computation is used to assign a cost to the entire operator.

If MAGNIFY did not annotate an operator, RTS forms its own estimate using the amount of data that the operator references as a guide to its likely execution time. Regardless of the annotation, when an operator is executed, RTS tracks its execution time and replaces the heuristically derived estimates with measured values. Hence the efficiency of an application

Dossier-description ::= *variables funcs*
variables ::= *variables variable* | ϵ
variable ::= '`var`' *string* ':' *shapes*
shapes ::= *shapes* ',' *shape* | ϵ
shape ::= *int* '`words`' | '`[`' *dims* '`]`'
dims ::= *dims dim* | ϵ
dim ::= *expr* '`..`' *expr*
expr ::= *int* | '`arg`' *int offset* | '`var`' *string offset*
offset ::= *sign int* | ϵ
sign ::= '`+`' | '`-`'
funcs ::= *funcs func* | ϵ
func ::= '`func`' *string* ':' *args retvals exec-info*
args ::= *args* '`arg`' *arg* | ϵ
arg ::= *shapes* ',' *usage*
retvals ::= *retvals* '`ret`' *retval* | ϵ
retval ::= *shape*
exec-info ::= *variance iter-cost*
variance ::= '`variance`' *int* | ϵ
iter-cost ::= '`cost`' *iter-category* | ϵ
iter-category ::= *int* '`flops`' | *int* '`usecs`' | *int* '`secs`'
usage ::= '`R/O`' | '`R/W`' | '`W/O`'

Figure 7.6: Dossier Grammar

will often improve dramatically over the course of its execution, as the initial estimate can be very inaccurate.

7.2.2 Notation Details

When it outputs the control framework of an application in Delirium, MAGNIFY defines a unique name for every variable identifier. Delirium is a single-assignment language, so each name can only refer to a single quantity. The Dossier annotations are appended to the bottom of the Delirium program text, characterizing each of the variables that appeared. Each computational operator is also described. Figure 7.6 gives the BNF grammar for Dossier.

The description of a variable is a *dimension expression* that describes the size and shape of the data that the variable represents. The expression either gives a fixed number of words, or it is an array shape declaration. A few examples:

```
var A : 1 words
var B : [1 .. 10, 1 .. 5]
```

```
var C : [1 .. var A, 1 .. var A + 1]
```

Operators are annotated with two categories of information: argument (and return value) descriptions, and execution behavior. For each argument, the description gives a dimension expression and a *usage* category. Usages are one of $\{R/W, R/O, W/O\}$, depending on whether the variable is read and written, only read, or written before being read respectively. This annotation allows RTS to determine whether replication is permitted; if MAGNIFY cannot be certain of the variable's behavior, it is always safe to mark it R/W.

The execution behavior is described by a pair of values; an estimate of the execution time of the function and an estimate of that time's variance across calls. It is often impossible for MAGNIFY to determine either value at compile-time, but it traces paths through the code and uses information about loop bounds to attempt an estimate. When there are no conditionals or every path includes the same amount of computation, the variance is known to be zero. MAGNIFY counts flops and uses profile information to categorize execution cost; even an inaccurate guess can be a useful starting point for the RTS granularity assignment algorithms discussed below.

7.2.3 Example With Dossier

Figure 7.7 shows the Dossier annotations for the running example. MAGNIFY would append the Dossier to the Delirium text given above.

7.3 Back End

After MAGNIFY has finished analyzing, transforming, and decomposing an application, the back-end of the programming environment is responsible for executing it. The back-end consists of a Delirium compiler that converts the control structure into scheduling templates and RTS. A detailed discussion of these components is outside the scope of this dissertation and can be found elsewhere [84]. The next two sections give a brief overview.

7.3.1 Delirium Compiler

The Delirium compiler takes the coordination structure of a program, expressed in Delirium and Dossier, and compiles it into a form that is directly manipulable by RTS.

The basic task of the compiler is to produce a series of *scheduling templates* for each parallel operation. There are four types of scheduling templates: *sequential*, *static*, *profile*, and *sample*. The first of these executes the operation sequentially. The second uses the owner computes rule on distributed memory machines, and allocates tasks to processors evenly on shared memory machines.

The latter two use the dynamic scheduling algorithms described below. They differ in how they gather information about the distribution of task execution times. Profile templates use

```
func loop1
  arg [1 .. arg 4, arg 5 .. arg 6]  R/W
  arg [1 .. arg 4, arg 5 .. arg 6]  R/O
  arg 1 words                       R/O
  arg 1 words                       R/O
  arg 1 words                       R/O
  arg 1 words                       R/O
  variance 0
  iter_cost 1 usec
func loop2
  arg [arg 5 .. arg 6]              R/W
  arg [1 .. arg 4, arg 5 .. arg 6]  R/O
  arg 1 words                       R/O
  arg 1 words                       R/O
  arg 1 words                       R/O
  arg 1 words                       R/O
  variance 0
  iter_cost 1 usec

var vargroup1 : [1 .. n, 1 .. n]
var vargroup2 : [1 .. n]
var A :       [1 .. n, 1 .. n]
var B :       [1 .. n, 1 .. n]
var n :       1 words
var iter :    1 words
var c :       1 words
```

Figure 7.7: Dossier Annotations for Example

information gathered from previous runs of the program to estimate execution time mean and variance [107]. Sample templates begin with a rough estimate of these two quantities and refine the estimates through run-time sampling of task execution times.

7.3.2 RTS, the Run-Time System

RTS performs two primary tasks:

- Choosing an allocation of computations to processors. The decomposed application consists of a set of primitive sequential operators that are scheduled when the data they need is available. Before the code can begin executing, that data must be routed to the processor (unless the target architecture is a shared memory machine, of course).
- Picking the appropriate granularity for those computations that can be tuned. As has been discussed above, many of the primitive operations are designed to execute a range of loop iterations. RTS is responsible for choosing how large the range should be, using information about the current state of the machine.

The heart of the run-time system is a set of metrics that track the current state of the machine and a set of algorithms that schedule computations and tune granularity based on those metrics. The following description is a brief summarization of the algorithms; they are discussed in detail elsewhere [83, 84].

One consequence of using Delirium as an intermediate form is that MAGNIFY fixes the minimum grain size of the computation. The set of non-re-entrant operators determines the minimum units of scheduling. Henceforth, we'll call these indivisible scheduling units *tasks*.

While the minimum grain size of the computation is fixed by the front end analysis tool, the Delirium compiler and run-time system retain crucial flexibility in determining the grain size at which a data parallel operation is scheduled. A previous paper [83], develops a quantitative relationship between total available parallelism, optimal grain size, and execution time variance of a parallel operation. The relationship is applied to the problem of choosing grain sizes for parallel operations, demonstrating that adaptive scheduling is often both necessary and sufficient for efficient execution of irregular parallel operations.

This section and the next demonstrate that interactions among parallel operations, both regular and irregular, can also benefit from adaptive strategies.

7.3.3 Orchestrating Interactions Among Parallel Operations

Consider a run-time scenario in which the transformed parallel operations A and B_I from Figure 6.8 are executing simultaneously. A begins executing first and has partially completed when B_I begins executing. RTS must decide how many processors to reallocate. If we change the example so that all processors must synchronize upon completion of A and

B_I, an ideal processor allocation would minimize the expected finishing time of these two parallel operations.

The expected finishing time of a parallel operation is a function of the number of tasks that make up the operation (N), the number of processors cooperating to execute the operation (p), the variance in task execution times, and the overhead of scheduling [72, 83]. Thus, approximating the ideal processor allocation requires information available only at run-time.

For this reason, we extended adaptive algorithms developed for single irregular parallel operations to manage interactions among multiple, simultaneously executing parallel operations. There are three main extensions. First, we developed a method for improving the accuracy of estimated finishing times that works for a wide range of scheduling algorithms. Second, we applied this method to the run-time processor allocation problem, using an iterative algorithm to equalize finishing time estimates. Finally, we combined finishing time estimates with run-time communication cost estimates to choose communication granularity for pairs of pipelined parallel operations.

Individual Parallel Operations

To provide a basis for describing these extensions, we review in this section our adaptive run-time algorithms for executing single parallel operations on distributed memory machines.[1]

We use a probabilistic algorithm called TAPER to select the grain-sizes at which tasks are scheduled. The run-time system samples task execution times to compute their statistical mean (μ) and variance (σ^2). It uses this information to reduce overhead by scheduling large chunks (groups of tasks) at the beginning of a parallel operation and successively smaller chunks as the computation proceeds. If we define a scheduling event as the moment when a processor finishes executing a chunk, then for each scheduling event i, TAPER computes K_i, the number of tasks in the ith chunk.

RTS does additional sampling of task costs to build a *cost function*, which estimates task execution times as a function of iteration number within the parallel operation. We use the cost function to scale a chunk size K_i by $s = \mu_g/\mu_c$. In this expression, μ_g is the global mean execution time and μ_c is the mean for the tasks in the current chunk.

Regular parallel operations execute according to the "owner-computes" rule [64]. To execute irregular parallel operations, we begin with some original data decomposition and assign tasks to processors according to the owner-computes rule. During execution, as RTS gains information about the work distribution, it refines the data decomposition.

In the distributed TAPER algorithm the p processors are logically connected as a binary tree with p leaves. Some of the processors act as both leaves and internal nodes of the tree. All processors start in *epoch* 0. When a processor begins executing a chunk it sends its current epoch value (called a *token*) to its parent, which passes the token to its parent (possibly combining messages from both children). When the root receives p tokens from the same epoch, it increments the global epoch value and broadcasts a message through

[1] The shared memory algorithm is described in [83].

the tree to all processors. The message tells the processors to increment their epoch value and may also tell some processors to transfer a chunk of tasks and their associated data to another processor.

Processors compete for the p chunks of each epoch. If processor a can get two tokens of value i to the root before processor b can send one token of value i, then the root will re-assign processor b's chunk of size K_i to processor a. Processor b is then forced to re-interpret the chunk it is currently executing as belonging to some later epoch (and thus containing fewer tasks). If most of the actual task cost is on a few processors, this scheme will degenerate into the centralized TAPER algorithm. If task costs are independent, we expect most tasks to remain on the processor owning them at the beginning of the parallel operation; thus, the algorithm reduces task transfer costs and maintains communication locality.

Interacting Parallel Operations

Now, we return to our run-time scenario in which the parallel operations A and B_I are executing concurrently. When B_I begins executing, the run-time system must reallocate some of the p processors executing A. To accomplish this, RTS uses the following iterative algorithm:

```
epsilon = 5%
p1 = p/2, p2 = p - p1, count = 0
eA = finish_estimate(A, p1), eB = finish_estimate(B, p2)
while ((count < max_count) and (|eA - eB| > epsilon))
      if (eA > eB)
        p1 = p1 + p2/2
        p2 = p - p1
      else
        p2 = p2 + p1/2
        p1 = p - p2
      eA = finish_estimate(A, p1)
      eB = finish_estimate(B, p2)
      count = count + 1
```

We limit the number of iterations to control the amount of overhead imposed. In practice, using a *max_count* of four has been sufficient.

In estimating finishing time, RTS uses the expression:

$$finish = setup + compute + lag + comm + sched \tag{7.1}$$

setup is the maximum of the time to contract the data required by A onto p_1 processors and the time to expand the data required by B_I onto p_2 processors. *compute* is the expected mean time to perform a portion of the computation: $N\mu/p'$ (where $p' = p_1$ for A and $p' = p_2$ for B_I). *lag* is the expected maximum finishing time to perform a portion of the

computation; it is related to the distribution of task execution times for the computation (μ,σ) [83]. *comm* represents the communication overhead of executing the given parallel operation on p' processors.

To estimate *comm* at run-time, we use an algorithm like that suggested by Sarkar and Hennessy [108]. It expresses the computation being performed as a graph and computes a weighted sum of graph edges that cross processor boundaries. The Delirium compiler uses a similar graph-based representation; rather than perform the computation statically, however, the compiler generates code blocks that perform the estimate given run-time parameters such as N and p'.

Finally, we must estimate the scheduling overhead (*sched*). To do so, we need to predict, at run-time, the number of chunks that will be scheduled for the parallel operation (hence the number of epochs in the distributed algorithm given above). The method for predicting this parameter is discussed elsewhere [83]. By balancing the estimated finishing times of A and B_I, the run-time system uses the extra concurrency from B_I to compensate for A's irregular execution behavior.

7.4 Related Work

7.4.1 Functional Languages

The basic structure of Delirium is similar to strict functional languages like VAL [88] and SISAL [89]. The distinguishing features of Delirium are the use of widely available conventional languages to specify primitive computational operations, the use of coordination structures to express data parallelism, special support for parallel mapping computations, and the Dossier annotations that provide additional information about the data values being referenced. The run-time systems that have been developed for strict functional languages do not support the tunable granularity and multiple scheduling policies of the Delirium run-time system.

Other functional languages rely on *lazy evaluation* to express parallel operations. Lazy evaluation means that a computation is not performed until it is needed. Because it is difficult to implement efficiently, languages that support lazy evaluation often restrict it to a particular type of aggregate structure. The programmer describes the procedure for constructing each element in the structure; when an element has been computed, it can be used by a subsequent computation even though the entire structure is not yet completed.

For example, the language ParALFL [65] has strict arrays and lazy lists. Id [10, 94] has a lazy aggregate called an I-structure. The language Haskell [66] supports lazy arrays called *array comprehensions* [8]. Lazy aggregates can be a flexible means of expressing computation, but the more restricted constructs in Delirium can be compiled much more efficiently because they express the control structure explicitly.

7.4.2 Coordination

There are a number of other notations that allow programmers to coordinate the behavior of sub-computations written in conventional languages. The simplest strategy is to call a library of routines from within a sequential program. The two common types of libraries are message passing packages and shared memory packages. The former are provided with every distributed memory machine; the MPI standard [120] is an effort to standardize them. Other examples include PVM [115] and Express [50]. Message passing libraries allow the program to send data between processors and query certain aspects of the system state. Some of them provide more advanced features like asynchronous I/O, barriers, task migration support.

In the shared memory programming model, all the processors use the same common memory locations. Higher-level coordination is done with locking (mutual exclusion) primitives embedded in a host language. On shared memory multi-processors such as the Sequent Symmetry, this model directly reflects the underlying architecture, and is a good low-level environment for programming [48]. Shared memory can also be simulated on distributed memory machines, but the strategy has a substantial and often unacceptable overhead.

Linda [54, 53] is a set of primitives that provides access to a shared associative tuple-space. This space is accessed through read, insert, and remove operations. Tuple space is shared by the processors and serves as the medium of communication between them. The primitive operations request a tuple matching a given pattern, and the associative mechanism performs a lookup. If there are many tuples that match, a random one is selected.

Schedule [45] provides primitives that allow the programmer to build a dataflow graph operationally. The sub-computations are connected by queues; the system will execute a sub-computation when a value is available on each queue that it has specified as an input. The programmer is responsible for decomposing the computation and specifying its coordination requirements. Schedule provides graphical tools for examining the execution behavior of the application.

Ada [119] provides some coordination features as language primitives. The coordination model is based on *task rendezvous*. The relative merits of this model are discussed in detail elsewhere [13]. Like the languages discussed above, one can think of Ada as containing an embedded notation for expressing operations in its coordination model.

Object-oriented systems such as Emerald [25], Amber [31], and Sloop [82] use abstract data types to *encapsulate* shared data. That is, a particular call on an abstract data type can only directly modify local data for the called instance of that type. This encapsulation has the additional benefit that one can improve performance by explicitly moving object instances about the network of processors [68]. Sloop, Emerald and Amber all provide an embedded primitive for specifying object locality.

Delirium is distinguished from all of these approaches by encapsulating the parallel control structure of the application in a textually separate form. The advantage of this strategy is that it allows a run-time system to analyze and manipulate just the coordination of

the application. A notation that is scattered throughout a sequential application either eliminates the need for a run-time system or severely restricts its ability to make scheduling decisions adaptively.

Like Delirium, the PCN [51] programming environment uses a separate coordination text that is executed by a run-time system. PCN relies on a logic-based notation rather than a functional one, and includes primitives in the language to support the notion of a virtual topology. The program can give hints to the run-time system about scheduling by declaring a mapping of computation to the topology.

7.4.3 Data Annotations

The idea behind Dossier is to summarize the shape and behavior of data passing between sub-computations. No other systems use this information to make scheduling decisions adaptively, so there is relatively little related work. There are a few kinds of annotations about data usage, but they focus on side-effects rather than shape.

Jade [105] provides a set of annotations added to sequential programs that divides the text into sub-computations and describes the data the pieces will require while executing. A sub-computation is scheduled when the data that it needs is available; the Jade annotations define tags which are generated and consumed as their associated data is available and in use. The tags stand in for the data they are associated with but do not describe its shape. The goal of Jade is to permit correct execution in a shared memory environment. It does not provide data shape information, which is used in the Delirium environment to improve efficiency on distributed memory architectures.

Another kind of annotation is used by the *fluent* languages [55] to allow imperative code to be mixed in with functional code. The annotations describe the side-effects that an imperative function will cause, so that it can be called from within a functional language without violating the semantic restrictions of referential transparency. Again, the purpose of the annotations is correctness.

7.4.4 Run-Time System

RTS, the Delirium run-time system, contains a number of novel features that allow it to target large numbers of processors efficiently. There are many other run-time systems that support parallel execution, with dramatic variations in the amount of control they have over program execution and the depth of their understanding of the program semantics.

When a run-time system is aware of the semantics of the programming language, it often is serving essentially as a language interpreter. Delirium falls into this category, although for efficiency many scheduling and policy decisions are made by the Delirium compiler rather than by RTS.

The PARTY run-time system [91], designed to support the high level language Crystal [33], also adjusts computational behavior based on the behavior of the application. It

converts the program into a directed acyclic graph, where each node represents a primitive computation. The focus of the system is on aggregation and mapping to balance processor load.

A more restrictive interpretive strategy is to have the run-time system enforce simple constraints that are explicitly declared by the program. The run-time system is still acting as an interpreter, in a limited sense, but it has little flexibility to alter the behavior of the application based on the current execution state. Granularity is fixed and aggregation is not supported. The systems that adopted this approach often supplied a graphical environment for connecting blocks of code and declaring their relationships. Two examples are Poker [95] and Schedule [45] (discussed above).

Another strategy is to use software support to give programs on a distributed-memory system the illusion that they share a common uniform memory space. There are several examples, including Kali [71], Ivy [76], and the Wisconsin Wind Tunnel [104]. All of them suffer from substantial overhead, although they are able to reduce it by relying on programmer or compiler support to reduce the frequency of remote references. Various specialized forms of hardware support can also be exploited to allow local references to occur at native speed.

The Delirium programming environment does not follow this strategy; although it begins with programs that are written for a single memory, all such assumptions are removed during the conversion to parallel form. In the output of MAGNIFY each piece of code expresses all its data requirements explicitly; the run-time system handles the movement of data to allow code to execute, but all the requirements are expressed. The advantage of a system like Kali is that code can be parallelized that is less thoroughly analyzed. Where static analysis was unable to accurately determine data requirements, the missing data can be made available at run-time by the data transport model. However, there is a high price in overhead to be paid for that safety net.

TAPER, the iteration scheduling algorithm used by RTS, has a number of predecessors. Several research projects have investigated how to schedule iterations of a loop on a shared memory machine. Although the developers of these algorithms generally insert code directly into the executable so that each loop schedules itself, the same ideas work within the context of a run-time system that makes dynamic scheduling decisions.

The simplest algorithm is self scheduling [116], where processors request a new iteration when they are free. The process continues until all the iterations have been executed. To reduce scheduling overhead, each request can yield a group of iterations rather than a single one [72].

Tapering algorithms change the number of iterations that are allocated as the computation progresses. Guided self-scheduling [99] and factoring [67] consider the number of tasks remaining when choosing an allocation block size. These algorithms perform very well when the time required to execute each iteration is similar, but TAPER's more sophisticated model of execution yields better performance when variance is an important factor.

RTS also uses some techniques developed for static scheduling systems. Sarkar and Hen-

nessy, for example, use execution time estimates and a critical path algorithm to partition parallel programs statically [108]. The run-time system determines the minimum acceptable grain size using a similar strategy, balancing communication cost against execution time estimates.

Chapter 8

Experimental Results

This chapter analyzes the behavior of MAGNIFY, both in isolation and as a piece of the Delirium programming environment. The first set of results consists of three case studies that demonstrate how MAGNIFY transforms an application for efficient parallel execution. The second set measures the cost of MAGNIFY's analysis using both the case study applications and some sample programs taken from the SPEC benchmark set [43].

The case studies that are presented here were chosen from a set of scientific applications that we have parallelized with the Delirium programming environment. We chose to focus on these programs because they are typical of a wide range of applications, both in the computations that they perform and in the transformations necessary to achieve highly efficient execution. The codes perform standard computations such as integration, Fourier transform, and nearest-neighbor calculations that incrementally update grid point values.

In each case, a straightforward parallelization strategy based on static decomposition of the parallel operations performed fairly well on a few processors but quickly arrived at a performance ceiling where adding additional nodes did not yield further improvement. The great majority of existing parallel applications are implemented by hand and do not go beyond such relatively simple decomposition strategies. Arrays are allocated statically to processors using a regular layout strategy. The application generally synchronizes after each parallel operation.

We were able to achieve significant additional speedup by using the Delirium programming environment to implement more complex parallel control strategies. The two most important optimizations were to expose additional concurrency using split and to combine inexpensive parallel operations for improved efficiency. Such techniques are not widely used in parallel applications because the code becomes extremely difficult to understand. The control structure of the program is obscured by a large number of message passing operations that are scattered throughout the program. To take advantage of prefetching and to interleave communication and computation, transfer statements must be moved to unrelated areas of the program. The resulting code is extremely difficult to debug and unpleasant to maintain.

MAGNIFY allows the programmer to avoid the task of implementing such transformations by hand while still achieving very good efficiency on many processors. The key is that the system can combine parallel operations and can divide them using split, allowing it to go beyond concurrency within an individual loop nest to address the interactions among parallel operations.

8.1 Case Study Format

To demonstrate how MAGNIFY parallelizes applications, the next three sections present case studies. Each study examines an application, beginning with a description of what the program does and its computational structure. Then the study traces the transformation of the application in stages from the original sequential form to a highly efficient parallel implementation. A performance graph is given showing the efficiency of each stage of the program using different numbers of processors.

Each stage in the evolution of the code is the result of applying a series of transformations to the previous stage. These transformations can either be applied to the code itself or to the WEB representation of the application. The code transformations are discussed throughout Chapter 6 and the specific WEB operations are listed in Section 6.1.2.

The case studies describe how the transformations address the performance bottlenecks. For each stage, we have listed the transformations and give a key that characterizes the analysis necessary to apply them. The following is an example:

Transformation	Uses
Chop application into analysis and output	
Parallelize analysis	*ES*, *FI*

Every application begins as a single WEB sequential operator. The first transformation above divides that operator into two, using the "chop" WEB operation. The fact that there are no symbols to the right of the transformation shows that MAGNIFY did not use symbolic, conditional, or interprocedural analysis to carry out the operation. The next transformation, however, converts the second WEB node into a map node. In order to do that, MAGNIFY must first discover whether the operation is concurrent and then modify it to be handled by multiple processors. In doing so, as is shown by the two code symbols, MAGNIFY manipulated descriptors that contained extended symbolic information and required full-scale interprocedural analysis.

Here are the code symbols that may appear next to a transformations and an explanation of their significance:

ϵ	write: $A < 1..10 >$ read: $A < 11..20 >$
S	write: $A < i..i >$ read: $A < 1..j >$
ES	write: $A < i..x + 1 >$ read: $A < 1..n + m + 2 >$
CS	write: $A < 2 * i + j..k >$ read: $A < 1..a + b + c + 1 >$

Figure 8.1: Levels of Symbolic Complexity in Descriptors

S	Used the symbolic engine. The transformation relied on the fact that descriptors can contain references to induction variables in order to carry out the transformation.
ES	Used extended symbolic information. The transformation required the presence of simple expressions.
CS	Used complex symbolic information. The transformation required the presence of complex expressions.
C	Used conditional information. The transformation relied on guards and/or masks in the descriptors.
I	Used interprocedural analysis. The transformation uses summary information for a called procedure.
FI	Used full interprocedural analysis. The transformation relied on the reanalysis of procedures at every call site.

The first three symbols characterize the level of symbolic complexity that was needed. Figure 8.1 shows a set of SDD's that demonstrate the different levels.

If none of the symbol codes appear, MAGNIFY carried out the transformation with SDD's that did not contain any expressions with symbol references in them. If a loop was involved, for example, its iteration bounds must be constant values that are known to MAGNIFY. If the symbol S appears, MAGNIFY did use symbols. However, only loop induction variables appeared in an SDD and they were not part of an arithmetic expression. In the example shown in Figure 8.1, both i and j are induction variables. ES indicates that *simple*

expressions were used. An expression is considered to be simple if every symbol has a constant multiplier of one and there are no more than two symbols plus an integer offset. Any expression that is not simple is *complex*; CS indicates that complex expressions were necessary.

The next symbol, C, appears if conditional information was needed to perform the transformation. Conditionals can appear either as guard expressions or as masks.

Finally, the last two symbols reflect the complexity of interprocedural analysis that was required. If neither is present, MAGNIFY would have been able to apply the transformation without propagating any information through a call site. I appears when MAGNIFY could have computed a single summary descriptor for each procedure and used that summary at every call site. FI indicates that the more aggressive strategy was necessary — reanalyzing the called procedure at each call site.

8.2 Case Study 1: PSIRRFAN

The first example is an x-ray tomography application called PSIRRFAN [63] which was developed at UC/Berkeley. The basic goal of tomography is to take a set of measurements computed by passing a beam through an object at all angles and constructing an image of the interior (in either two or three dimensions). The beam generator is placed on one side of the object and an array of detectors on the other side measures the strength of the signal that penetrates.

The core tomography algorithms generally require that a reading be made through an object at evenly spaced angles. However, some objects cannot be examined from every angle, causing gaps in the data that is needed to assemble the image. For example, if the goal is to find cracks caused by earthquakes in freeway supports, the support may be embedded in a hill and hence the beam generator and detectors cannot be positioned at certain angles. The resulting information has gaps in it and is said to be *limited-angle* data.

In order to continue using algorithms that require full data to be available, one solution is to use PSIRRFAN, a limited-angle reconstruction algorithm. It computes values for the missing data so that an image can be constructed normally.

8.2.1 Computational Structure

The PSIRRFAN application consists of two major computations: an iterative reconstruction algorithm that fills in computed values for the missing data, and a backprojection algorithm that produces a two-dimensional image from the completed data set. The computationally intensive subset of the application is approximately 1500 lines of FORTRAN code.

Here is a structural decomposition of the program:

Reconstruction (*iterate until convergence*)

initialization
outer reconstruction integral (*for missing data columns*)
    initialization
    inner reconstruction integral
    Fourier filter
    data update

Backprojection

Each pass through the reconstruction algorithm updates the data values that were missing in the original data set. The reconstruction iterations continue until those values have converged. Both of the integrals in the reconstruction phase can be executed in parallel, as can the Fourier filtering step. Backprojection is a fully parallelizable operation consisting of three nested loops that traverse the data set to compute an image.

8.2.2 Transformation

This section describes the transformation of PSIRRFAN from its original sequential implementation to an efficient parallel version. There are four parallel versions of the code, each more efficient than the last on large numbers of processors.

Figure 8.2 shows the performance of each stage on a 256 processor Ncube Systems Ncube-2, reconstructing the cross-section of a freeway embankment column that was embedded into a hillside. The column contains steel reinforcement bars and some cracks caused by an earthquake. A 30 degree arc was obstructed by the hill. The detector array contained 256 detectors and the assembly was positioned at 512 evenly spaced angles around the column (except where there was an obstruction). The resolution of the resulting cross-sectional image was 1024 by 1024 pixels. The speedup measurements recorded in the graph compare the parallel version of the program to the original sequential implementation.

Stage 1: Simple Parallelization

The first step takes advantage of the most accessible sources of concurrency in PSIRRFAN. The object is to parallelize both the backprojection computation and the outer reconstruction integral. In order to do so, the computations to be parallelized must be separated into their own WEB nodes.

The application, which begins as a single WEB sequential operator node (see Section 6.1.1), is divided into two sequential nodes representing reconstruction and backprojection respectively. The next operation is to peel the iterations of the reconstruction node, converting it into an iteration node. Next, the contents of that node are subdivided into one node for initialization and another for the outer integral. Then, both backprojection and the outer integral node can be converted into map nodes to reflect the fact that they can be executed concurrently.

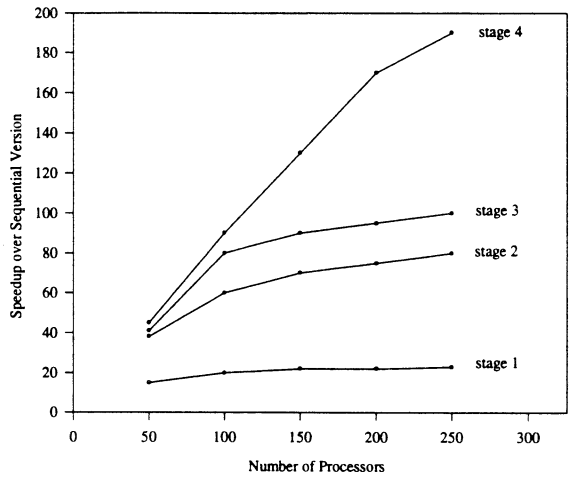


Figure 8.2: Efficiency of PSIRRFAN on a 256 Processor Ncube-2

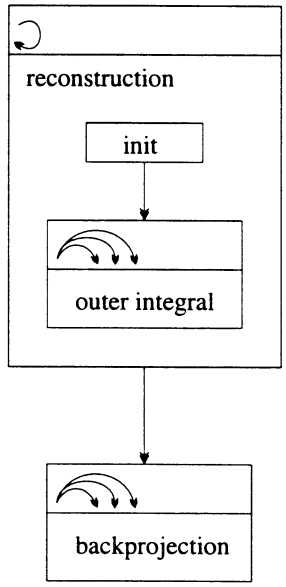


Figure 8.3: PSIRRFAN Stage 1: WEB Representation

The following table lists the transformations for this stage. Figure 8.3 shows the resulting WEB representation.

Transformation	Uses
Chop application into reconstruction and backprojection	
Peel the iterations of the reconstruction node	ES, I
Chop reconstruction to expose initialization and the outer reconstruction integral	I
Parallelize backprojection	S, I
Parallelize outer reconstruction integral	ES, C, I

Stage 2: Improve Inner Parallelization

Stage one reveals a significant amount of parallelism, as is shown in Figure 8.2, but the efficiency drops off substantially with larger numbers of processors. The problem is caused by the limited parallelism available in the reconstruction algorithm. The expensive computations are caused by the columns of data that are missing; they appear in an unpredictably irregular pattern in the data set and there are not enough of them to keep many processors busy.

The second stage improves performance by exposing additional parallelism. Rather than scheduling the body of the outer integral as a single computation, it is divided into its component sub-computations. Then the inner integral and the Fourier filtering step are parallelized. Figure 8.4 shows the modified WEB representation.

Transformation	Uses
Chop outer reconstruction integral to expose initialization code, inner reconstruction integral, Fourier filtering, and update	I
Parallelize inner reconstruction integral	CS, FI
Parallelize Fourier filter	ES, I

Stage 3: Pipeline Backprojection

While performance has been noticeably improved in stage two, the reconstruction step still limits the amount of concurrency. The third stage takes advantage of the relationship of reconstruction and backprojection to alleviate the problem. Most of the data needed by the backprojection algorithm is available before any reconstruction is performed, because only a small fraction is missing when the application starts execution.

The solution is to use the split transformation to expose additional concurrency. By dividing the backprojection step into one part that can begin immediately and one part that must wait for reconstruction, MAGNIFY reveals a great deal of additional concurrency. RTS can use the available backprojection work to make up for limited concurrency during reconstruction, thus using the processors more efficiently. Before backprojection can be split, though, its loops must be interchanged to match the reconstruction operation more exactly.

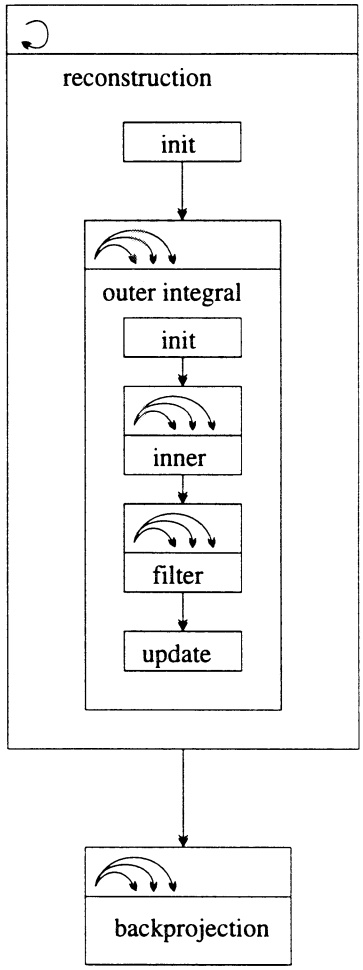


Figure 8.4: PSIRRFAN Stage 2: WEB Representation

Figure 8.5 shows the WEB after stage three.

Transformation	Uses
Interchange backprojection to expose parallel loop that matches the iteration structure of the reconstruction integral	ES
Split backproject to divide iterations of backprojection into those that depend on reconstruction and those that do not	ES, I
Parallelize both sets of backprojection iterations	S, I

Stage 4: Pipeline PSIRRFAN Iterations

The final step exposes some additional concurrency during reconstruction. The reconstruction phase, just like backprojection, performs most of its computation on data that was available from the beginning of execution. While iteration i of the reconstruction algorithm is still executing, there are some parts of the next iteration $(i + 1)$ that RTS can begin executing.

In order to introduce pipelining, MAGNIFY must split the reconstruction iteration to isolate the computation that is independent of previous iterations. The split operation is applied to the iteration body, using the descriptor for the next iteration as the basis of the division. The result is an iteration node that has two separate streams of execution in it, one that depends on the previous iteration and one that does not. The computation in the two streams is fed into a combining node. Figure 8.6 shows the corresponding WEB representation.

Transformation	Uses
Split reconstruction iteration based on subsequent iterations	CS, C, FI
Split initialization code	CS, I
Split outer reconstruction integral	CS, I

8.2.3 Discussion

The strategy in the first stage would be a natural one to use in parallelizing PSIRRFAN by hand. Particularly if the programmer uses a static allocation of computation to processors, the code to manage the parallel control structure is not too complex. The message passing statements are localized to a small fraction of the application and are relatively easy to understand. However, the synchronization that occurs between every parallel operation and the badly balanced load of the outer integral limits performance.

Stage two shows a marked improvement in performance because it exposes a significantly larger amount of parallelism within each iteration of the outer integral. However, creating the equivalent control strategy by hand is a difficult task. The scheduling choices that allocate inner integral iterations are greatly affected by the parallelism available from the outer integral. Unless the programmer chooses to implement a complex global scheduling strategy like the one in RTS, the improvement from the additional parallelism can easily be swamped by scheduling overhead.

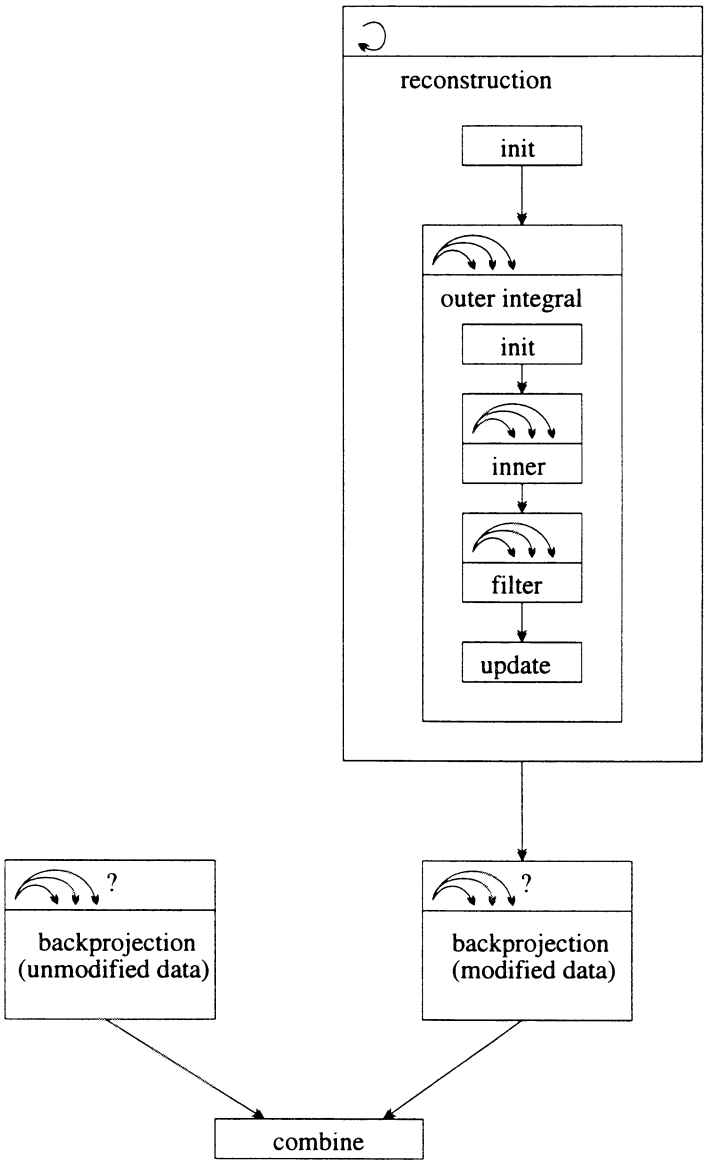


Figure 8.5: PSIRRFAN Stage 3: WEB Representation

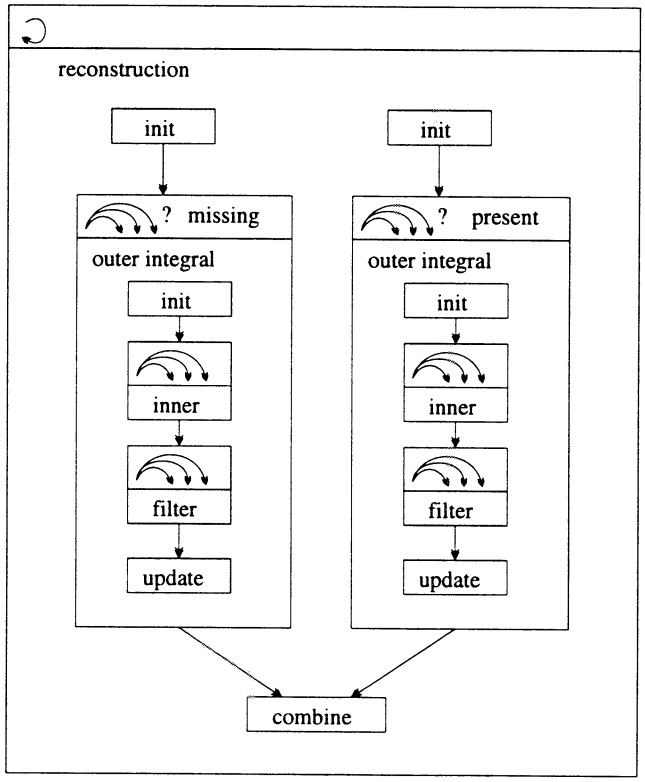


Figure 8.6: PSIRRFAN Stage 4: Partial WEB Representation

In order to achieve the efficiency of stages three and four by hand, the program would be rendered unrecognizable. The code must manage the complexity of multiple heterogeneous computations, allocating computations and choosing granularity based on the current state of the system. As the computation evolves, data must shift from one cluster of processors to another.

In order to allow RTS to perform these operations automatically, the code must be modified to expose all the potential forms of concurrency. The best performance was only achieved once split transformations had been applied repeatedly; as the transformation keys demonstrate, a great deal of symbolic information is needed to apply them. RTS is also dependent on the data sizes provided by Dossier to assess the communication cost implied by a scheduling decision.

8.3 Case Study 2: Camille

The second study examines Camille, a version of the UCLA Global Climate Model [9]. The UCLA simulation has been used as the basis for many climate modeling experiments. One of these projects was undertaken by researchers at the Lawrence-Livermore National Laboratory; they created Camille by starting from the base UCLA code and substantially rewriting it.

8.3.1 Computational Structure

The central data structure used by the UCLA model is a three-dimensional grid representing the atmosphere of the planet. The earth's surface is sub-divided into a two dimensional grid; the atmosphere above each of the grid squares is divided into a series of *levels* that start at the ground and extend upwards. The collection of levels for a given grid square is called a *column*. Using this three-dimensional structure, the atmosphere is divided into parcels, each of which is characterized by a set of quantities such as its temperature and its precipitation level.

This case study examines the part of Camille that models the physical interactions within a given column. Column physics makes up most of the computationally intensive work performed by the program and the code that implements it is approximately 7000 lines of FORTRAN. The reason we chose to work with only a piece of the full application is that Camille has been parallelized by adding calls to process control and message passing routines. The analysis performed by MAGNIFY is designed for sequential application programs.

The column physics is implemented in a sequence of passes. Table 8.1 lists each pass and summarizes its function. The next four sections describe how we incrementally parallelized these computations using MAGNIFY and RTS. Figure 8.7 compares the performance of each version on a 64 processor CM-5.

Name	Purpose
`pressure`	compute pressure in millibars at different altitudes in the model
`eventemp`	calculate the potential temperature at the even numbered levels using the value at the odd levels
`gpotentl`	compute the geopotential function ϕ for each level
`uclainit`	initialize arrays that are used in the subsequent passes
`clouds`	compute quantities characterizing cloudiness such as instability and hypothetical pressure at the bottom of the cloud base
`cumulus1`	calculate the effect on potential temperature, specific humidity, and velocity based on cumulus cloud interaction
`moistadj`	adjust potential temperature and specific humidity due to moist convection
`lsp`	calculate the effect of large scale precipitation on potential temperature and specific humidity
`shortwav`	calculate short wave radiation heating flux
`longwave`	compute effect of long wave (infrared) radiation in the atmosphere
`pbl`	account for the effect of surface turbulence fluxes on the planetary boundary layer (PBL)
`cumulus2`	calculate the effect of cumulus mass flux on velocity
`photchem`	compute ozone production
`groundt`	determine ground values, including temperature (based on surface radiative fluxes) and precipitation
`fluxrate`	compute flux rate based on values that were calculated in all the previous passes

Table 8.1: Computational Passes in Camille Column Physics

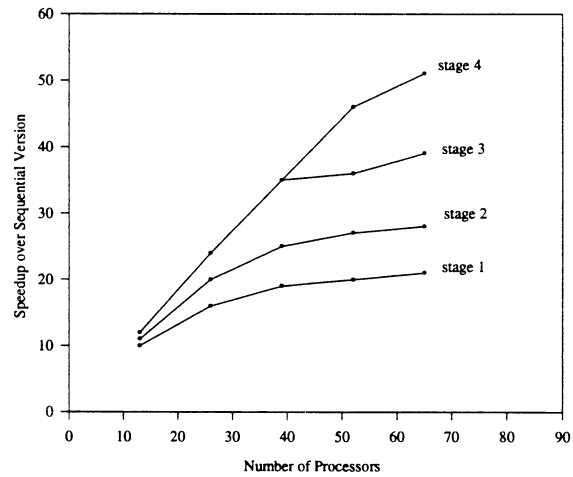


Figure 8.7: Performance of Camille on 64 CPU CM-5

8.3.2 Transformation

Stage 1: Parallelizing the Passes

We begin with the simple strategy of parallelizing each pass independently across the two-dimensional grid structure covering the earth's surface. Because columns interact with their immediate neighbors, it is advantageous, when possible, to allocate adjacent columns to the same processor.

As an initial strategy, therefore, we use a regular decomposition to allocate column iterations to processors. We chose a *tiled* decomposition, meaning that the two dimensional space of columns is divided into contiguous blocks and each processor is given a block to compute. After each pass is completed, the processors synchronize, waiting for the others to finish before starting the next pass. The allocation of columns to processors is not changed between passes.

RTS normally uses the more aggressive scheduling strategies described in Section 7.4.4, but MAGNIFY allows the programmer to specify that a regular decomposition strategy like tiling be used, to support performance comparisons.

The WEB representation of this strategy is simply a linear chain of map nodes, each of which computes one of the passes.

Transformation	Uses
Chop the single WEB node representing the entire column computation into its fifteen component passes	
Parallelize each pass	ES, I

Stage 2: Scheduling for Irregularity

The performance of stage one is shown above in Table 8.1. The speedup quickly reaches its limits as the number of processors increases. After analyzing the parallel behavior, we found that two problems limit efficiency:

- Some passes have irregular behavior, where the amount of time required to handle a given column depends on the value of the data. For example, the amount of precipitation determines the type of cloud computation that is performed. In very dry areas, such as a large desert, many neighboring columns may have similar cloud computation behavior. Using a regular decomposition strategy like tiling, one processor can have dramatically more or less work than another because it receives a cluster of similarly behaving columns. The irregular execution pattern complicates the problem of balancing load across the processors of the machine.
- Some of the passes do not require much computation; either RTS must use relatively few processors to execute them or the scheduling overhead becomes problematic. The `gpotentl` pass is an example.

In stage two, we address the first problem. The solution is to move from a regular tiled decomposition strategy to TAPER and the more advanced scheduling algorithms embodied in RTS(see Section 7.3.3 for a detailed discussion of them). As shown in Figure 8.7, the change in scheduling strategy yields a distinct improvement in performance.

There are no WEB transformations needed for this stage; the only change was to remove the scheduling declaration that restricted RTS to use a regular decomposition strategy. RTS employs TAPER and its normal scheduling algorithms.

Stage 3: Decoupling Independent Passes

Although stage two improves performance, the second limitation on performance still remains: some stages do not perform sufficient amounts of computation to keep many processors efficiently at work. Stages three and four improve the situation by exposing additional concurrency.

The first step is to relax the constraint that the system synchronize globally after each physics pass. The passes do not all depend on the one that precedes them; Figure 8.8 shows how the passes depend on each other. By applying the WEB operation that computes the dependency relationship of a group of nodes, the parallelism is exposed.

As described in Section 6.2.1, if the user allows it, MAGNIFY can rename reductions using accumulation arrays so that they do not prevent parallelization of otherwise independent computations. Many of the dependences shown in the figure are caused by reductions; several of the passes, for example, update the value of the array `deltat` by adding a value to each element of it. If such reductive updates are stored in temporary arrays and the sum computed later, a good deal of additional parallelism is available.

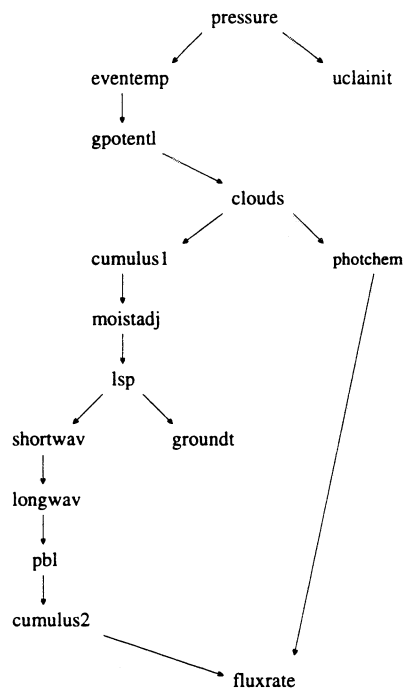


Figure 8.8: Physics Pass Dependences in Camille

Figure 8.9 shows the modified dependence structure that results when the accumulation arrays are handled with temporaries. For Camille, we have specified to MAGNIFY that the reduction conversion is acceptable.

The WEB representation for this stage is essentially identical to the dependency graphs shown in the figures and hence is not shown here; each pass is a map node that executes its computation in parallel. The additional concurrency yields a substantial improvement in performance.

Transformation	Uses
Compute dependences among component nodes	ES, I

Stage 4: Pipelining Successive Passes

The final stage exposes further concurrency by using pipelining between physics passes. When two passes depend on one another, typically the dependence is caused by the fact that one pass needs array values computed during the corresponding iteration of a previous one.

For example, the `cumulus2` pass that takes place after `pbl` uses values of the `entrain` array that are computed in `pbl`. Once a given column has been processed by `pbl`, there is no reason why that same column's `cumulus2` computation cannot start immediately without waiting for `pbl` to finish the rest of the columns. Similar pipelining opportunities exist between other passes, such as `moistadj`, `lsp`, and `groundt`. By applying transformations

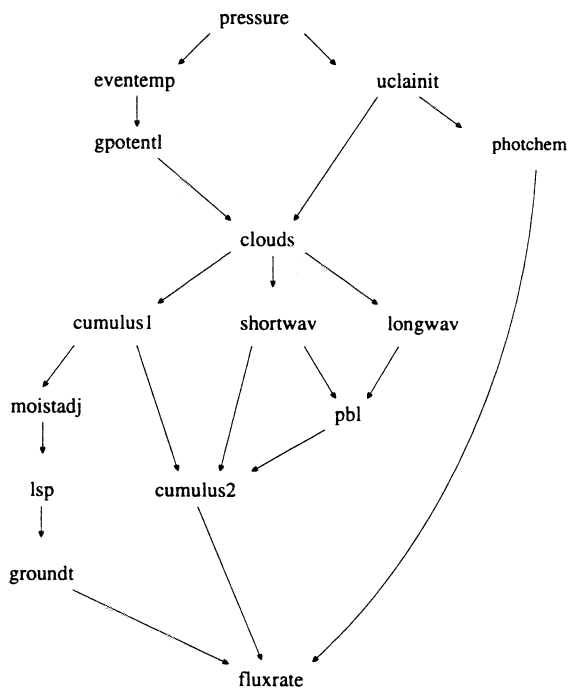


Figure 8.9: Camille Pass Dependences With Accumulation Arrays

to expose these pipelining opportunities, MAGNIFY can reveal enough parallelism for the run-time system to improve execution efficiency.

Rather than giving the full WEB representation for the application, Figure 8.10 shows the WEB representation for the interaction between `moistadj`, `lsp`, and `groundt`. They have been combined into a single mapping that contains a series of sequential operations. RTS can choose how much of each column to compute before scheduling another. The effect is to allow each scheduling decision to yield more computation, reducing overhead and improving efficiency.

Transformation	Uses
Combine map nodes for `cumulus1` and `longwav`	S, I
Combine map nodes for `cumulus2` and `pbl`	ES, C, I
Interchange loop in `gpotentl` to match mapping in `eventemp`	ES
Combine map nodes for `eventemp` and `gpotentl`	S
Combine map nodes for `moistadj` and `lsp`	S, I, C
Combine map node just constructed with `groundt`	ES, I

8.3.3 Discussion

The scientists at Livermore who developed Camille use a parallelization strategy similar to the one in the first stage. They synchronize after each parallel operation is complete and choose a blocking factor that is a compromise between load balance and scheduling

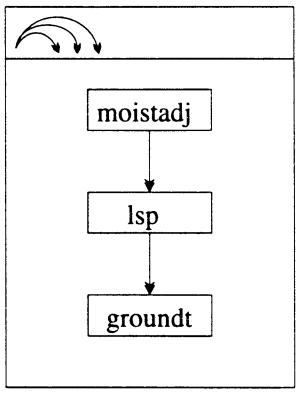


Figure 8.10: Camille Stage 4: Partial WEB Representation

overhead. The decision was a reasonable one to make, given that they wish to maintain portability across different architectures and to ensure that the program is easy to maintain and modify.

However, as the graph in Figure 8.2 demonstrates, a straightforward parallelization limits the performance as the number of processors increases. Stage two moves to a dynamic scheduling algorithm, improving the efficiency noticeably. Implementing such an algorithm manually would require a significant but not prohibitive coding effort.

Stage three, and particularly stage four, would require a major reorganization of the program to eliminate synchronization and introduce complex scheduling algorithms. Although the conceptual difference between stage two and three is not large, the coding complexity is disproportionately high. The programmer is forced to abandon the idea of a single thread of control that contains some parallel operations in it. Unfortunately, sequential languages like FORTRAN rely heavily on the sequential model and major changes to the code are needed to manage parallel threads.

The major difference between stage three and four is that the latter combines parallel operations. Computations like `eventemp` can be difficult to parallelize efficiently even though they offer abundant concurrency. The problem is that they are not doing much computation, so the overhead of scheduling the parallel jobs dominates the overall execution time. By combining inexpensive parallel operations, MAGNIFY can construct an operation that is easy to parallelize out of several that are individually difficult.

Depending on the application, we have found that the two most valuable optimizations are combining low-cost parallel computations and using split to pipeline computations that would otherwise be executed sequentially. Both rely on the concurrency exposed by MAGNIFY, its ability to reorganize the relationship of sub-computations, and the dynamic scheduling algorithms in RTS.

Name	Purpose
`shiag`	prepare for dynamics computation by translating the location of all molecules into the periodic box
`ekcmr`	calculate the total kinetic energy of the center of mass of the sub-molecules
`force`	apply various force models to the atoms to account for angle energy, dihedral energy, and position constraint energy
`scale`	reset box parameters and scale them based on new atom positions
`lsqfit`	perform least squares fit on total energy
`cenmas`	calculate the translational and rotational kinetic energies and velocities

Table 8.2: Computational Passes in Amber Molecular Dynamics

8.4 Case Study 3: Amber

Amber [121] is a molecular dynamics modeling program. It simulates the position of atoms within a molecule by calculating their interactions with one another. It can be used either with small molecules or with polymers. The application is a family of utilities that create data sets, display them, perform various modeling simulations, and present the results in a readable form. This case study describes the parallelization of the molecular dynamics modeling utility program. It consists of approximately 7000 lines of FORTRAN code.

8.4.1 Computational Structure

The modeling program consists of three main parts:

- initialization – data is read from files and various initial values for quantities like velocity and bond parameters are computed.
- molecular dynamics computation — the heart of the application. It proceeds for a specified number of iterations, calculating the forces on the various component atoms of the molecule and updating their position accordingly. The computation, which integrates Newtonian equations of motion, consists of a set of passes that are described in Table 8.2.
- clean up — output the results to a set of files.

8.4.2 Transformation

The following three stages demonstrate increasingly more efficient parallel implementations of the application. Figure 8.11 shows the efficiency of each stage in computing the structure of a 554 atom hexamer complex. The code was running on a 64 processor CM-5 machine.

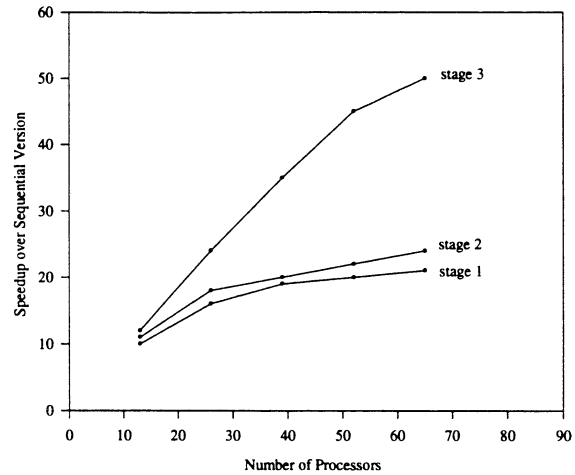


Figure 8.11: Efficiency of Amber on a 64 processor CM-5

Stage 1: Simple Parallelization

As usual, the first step in parallelizing Amber is to divide it into sub-computations, identify the ones that can execute in parallel, and convert their corresponding WEB nodes into map nodes. The initialization and clean up code is computationally insignificant; efficient parallelization depends on managing the molecular dynamics passes.

Unlike Camille, the passes do not all parallelize well. In particular, `lsqfit` uses a sequential computation. `cenmas` performs one set of calculations that parallelize well (calculating energy and velocity values for each atom) and one set that sequentially assembles global averages. In this stage, we continue to execute `cenmas` sequentially. The most expensive pass by a wide margin is `force` and it parallelizes well.

The following table lists the transformations for this stage. Figure 8.12 shows the resulting WEB representation.

Transformation	Uses
Chop application into initialization code, dynamics computation, and clean up	
Peel iteration from dynamics computation, converting it into an iteration node	S, I
Chop dynamics into component passes	I
Parallelize `shiag`	CS, I
Parallelize `ekcmr`	ES, C, I
Parallelize `force`	CS, FI
Parallelize `scale`	ES, I

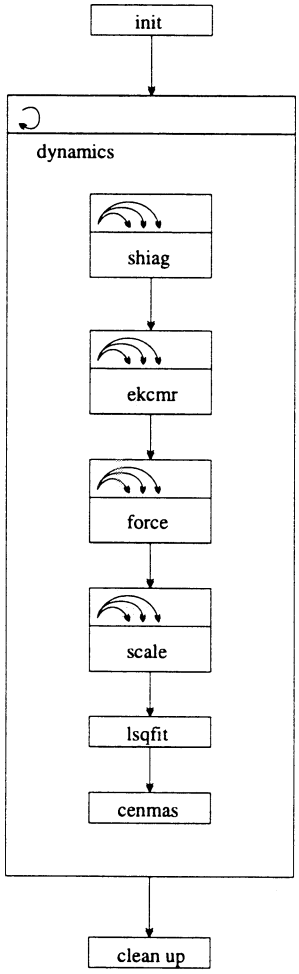


Figure 8.12: Amber Stage 1: WEB Representation

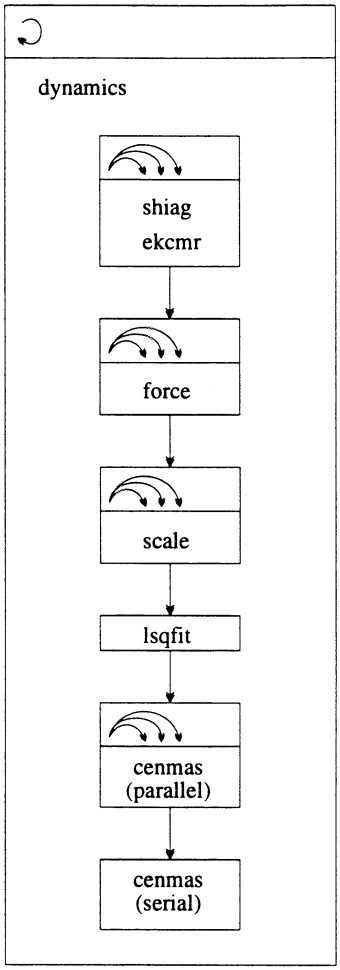


Figure 8.13: Amber Stage 2: WEB Representation

Stage 2: Improve Dynamics Parallelization

As is shown by the performance graph in Figure 8.11, the stage one parallelization runs into a bottleneck after more than 10 processors are used. The problem is the limited parallelization at the end of the dynamics computation, where both `lsqfit` and `cenmas` are being computed sequentially. In addition, there is a limited amount of work in `shiag`, causing high overhead during its computation.

In stage two, `cenmas` is divided into its sequential and parallel components. Also, `shiag` is combined with `ekcmr` to ensure that each scheduling event yields a larger computation and hence scheduling overhead is reduced.

The following table lists the transformations for this stage. Figure 8.13 shows the WEB representation of the modified dynamics computation.

Transformation	Uses
Chop `cenmas` into its component parts	
Parallelize the first half of `cenmas`	ES, I
Combine map nodes for `shiag` and `ekcmr`	CS, I

Stage 3: Pipeline Dynamics Iterations

Although stage two has better performance than stage one, efficiency still is constrained by the parallelism limitations. Stage three achieves a further improvement by taking advantage of an additional source of concurrency: pipelining the dynamics iterations.

Pipelining is useful because the initial computation of an iteration largely depends on the values computed by `force` in the previous one. The results from `lsqfit` and `cenmas` are not used in the next iteration until `force` begins. Therefore, the combined `shiag` and `ekcmr` computation for iteration $i+1$ can begin as soon as iteration i's `force` has completed.

In order to pipeline the loop, the `shiag/ekcmr` operation must be split against the descriptor for the previous loop iteration. When MAGNIFY outputs the Delirium for the pipelined loop, one of the pieces of the split computation will be independent of the data computed during the previous iteration. As a result, RTS is able to use the additional concurrency when a single dynamics iteration does not provide enough to keep the processors efficiently busy.

The following table lists the transformations for this stage. Figure 8.14 shows the WEB representation of the pipelined dynamics computation. The dashed line from `force` back to the conditional map operation is a visual indication of the data dependence between iterations. It is not present in the actual WEB representation used by MAGNIFY because the descriptors encode the dependence information implicitly.

Transformation	Uses
Split the combined `shiag/ekcmd` node	CS, I
Peel conditional from each of the split nodes to convert them into conditional map nodes	CS, I

8.4.3 Discussion

As in the previous two case studies, the first stage uses a variation of the strategy that most programmers would use in parallelizing Amber by hand. It performs fairly well on small numbers of processors but reaches a maximum speedup of approximately 20.

The next stage is also not difficult to implement manually and would not require extensive changes to the code. However, it does not provide much of an improvement in performance.

The real increase in efficiency comes from stage three, which applies the split transformation to pipeline the major iterations of the simulation. This modification moves entirely away from one thread of control and requires complex scheduling decision to choose granularity and migrate data between processors dynamically. The changes to the application in stage three are systemic and rely heavily on complex expressions in the summary information

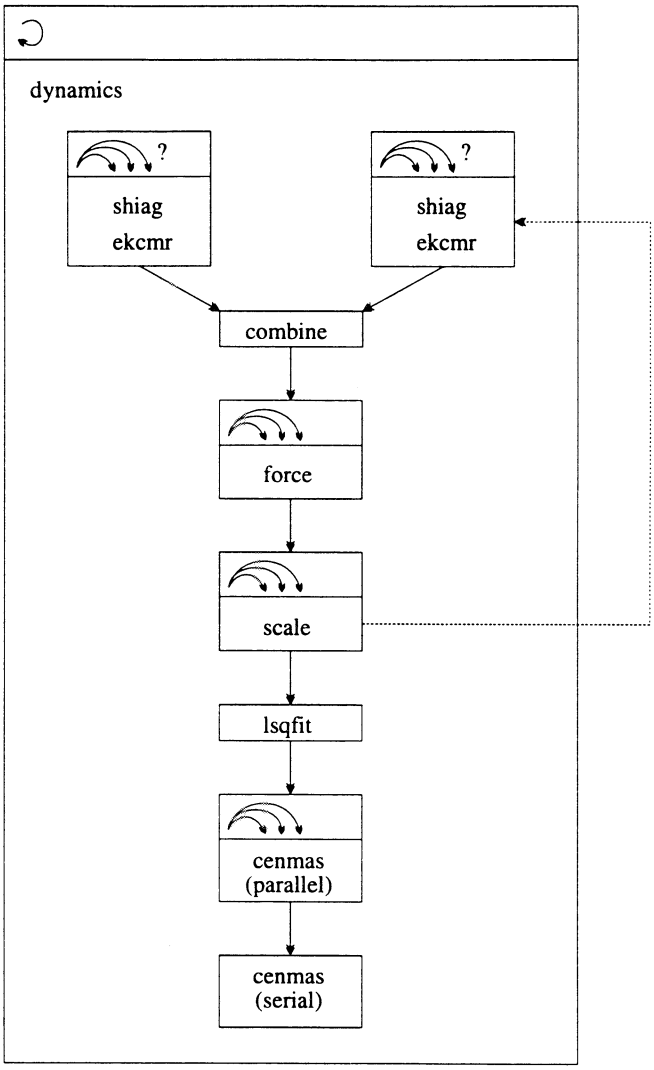


Figure 8.14: Amber Stage 3: WEB Representation

	nasa7	hydro2d	psirrfan	amber
full/complex	4.6	28.2	31.6	83.1
full/simple	3.0	17.6	12.0	40.6
summarize/simple	3.0	10.0	6.2	18.0
none/simple	1.6	3.6	2.6	4.8

Table 8.3: Execution Cost of Analysis (in seconds)

available from the descriptor.

The graph in Figure 8.11 demonstrates that the added concurrency has a dramatic effect on performance.

8.5 Cost of Analysis

This section measures the cost of executing MAGNIFY to demonstrate that its algorithms are not prohibitively expensive to use. The process of transforming an application using MAGNIFY consists of a series of steps, but the most time consuming by far is the initial annotation of an application with its memory usage descriptors.

As the application is modified, some reanalysis is usually necessary after each transformation, but it is always less time-consuming than the initial pass. In practice, we find that the time is usually negligible (no more than a few seconds); the longest reanalysis we have encountered executed for approximately one-fifth as long as the original summarization.

Consequently, this section focuses entirely on the cost of the initial analysis pass. The first set of results, shown in Table 8.3, were run on a SPARCstation 10 from Sun Microsystems [114]. They measure the amount of time that MAGNIFY requires to analyze four different applications completely. The analysis involves the construction of an SDD for every program construct of interest — every loop, conditional construct, basic block, and procedure.

We chose these four programs to demonstrate a range of execution times:

- `nasa7` — a program from the SPEC92 suite that contains the NAS Kernel benchmarks
- `hydro2d` — also from SPEC92, an astrophysics simulation based on the hydrodynamical Navier Stokes equations
- `psirrfan` — from the first case study
- `amber` — from the third case study

The simplest is `nasa7`; it is fairly short and has a simple control structure. Amber is the most expensive program to analyze in our application test suite.

The leftmost column in the table indicates the level of interprocedural and symbolic analysis:

- full — reanalyze every procedure for each call site
- summarize — analyze each procedure once and use that summary at each call site
- none — perform no interprocedural analysis
- simple — use only simple symbolic expressions, as explained in the introduction to the case studies (Section 8.1).
- complex — allow arbitrarily complex symbolic expressions

In the worst case shown here, the total time for a full analysis was a bit more than one minute and hence is easily acceptable for an interactive environment. In fact, given that the analysis is performed once and can have a substantial impact on the efficiency of the resulting parallel implementation, in most cases a programmer should be willing to wait for a much longer period of time. However, the cost might become burdensome for larger applications. If so, we have demonstrated two simple strategies that substantially reduce the analysis time at the price of reduced accuracy.

The first is to restrict analysis to simpler symbolic expressions. As was shown in the case studies, in practice most transformations can be applied without the full complexity. The code CS next to a transformation step indicates otherwise, and appears relatively infrequently. However, some of the optimizations that were used to achieve peak performance would not be possible with simple expressions.

The second option is to reduce the aggressiveness of the interprocedural analysis. Choosing to do none at all permits very fast analysis, but prevents almost any interesting transformation — few complex scientific computations are expressed entirely within the boundaries of one procedure. Moving to summarization is significantly more time consuming, but still much less so than the full analysis.

The difference in cost between summarization and full interprocedural analysis is largely determined by programming style. For the examples shown in Table 8.3 and for other codes that we have examined, full/complex analysis costs between two and ten times as much as summarize/simple.

If the call tree structure is uncomplicated, where there are one or a few paths to most leaf procedures, full interprocedural analysis is not much more costly than using a summary. The **nasa7** program is an extreme example, because it does not have any procedure calls at all except for one main routine that invokes the various kernels. Adding interprocedural analysis costs a small amount and there is no difference between full and summary analysis because every procedure has exactly one call site. Amber, on the other hand, is extensively divided into procedures, so the effect is more noticeable.

Table 8.4 provides a detailed evaluation of the execution time required to build the descriptors. Section 5.3 analyzed the cost of assembling symbolic descriptors in terms of a set of average quantities (they are listed in Table 5.1 on page 40). The table in Table 8.4 gives

	full/complex			full/simple			summarize/simple		
	hydro	psirr	amber	hydro	psirr	amber	hydro	psirr	amber
$reanalyze_{avg}$	.03	.12	.1	.03	.12	.1	.02	.09	.07
$proc_{avg}$	.15s	.6s	.38s	.1s	.25s	.19s	.06s	.12s	.08s
$triples_{avg}$	5.2	8.1	5.7	5.2	8.1	5.7	5.2	7.0	5.7
$expr\_union_{avg}$	7ms	10ms	8ms	5ms	4ms	2ms	3ms	3ms	2ms
num_exprs_{avg}	10.2	17.4	10.0	4.5	3.0	3.2	4.0	2.9	3.1
$dirty\_var_{avg}$	2.1	4.1	2.2	.9	1.6	.8	.7	1.0	.8
ops_clean_{avg}	2.9	6.1	4.0	2.1	2.7	.9	1.1	2.4	1.0
op_cost_{avg}	.6ms	.9ms	.7ms	.3ms	.5ms	.2ms	.2ms	.6ms	.2ms

Table 8.4: Breakdown of Cost of Assembling Descriptors

a value for each of those quantities under different circumstances. Some of the values are unitless numbers; $triples_{avg}$, for example, is the average number of triples in a descriptor. Where the value represents elapsed time, it is followed by "ms" for milliseconds or "s" for seconds.

There are three sets of numbers: one for full analysis with complex expressions, one for full analysis with simple expressions, and one for summary analysis with simple expressions. In each set, the value of the average is given for the `hydro2d` SPEC benchmark, for PSIRRFAN, and for Amber. We presented results earlier for the SPEC program `nasa7`, to show the effect of an unrealistically simple computational structure, but a detailed breakdown is not useful and is not shown.

As the table demonstrates, the introduction of full analysis has its most concentrated effect on the average procedure analysis cost ($proc_{avg}$). The other numbers are more sensitive to the complexity of the symbolic analysis. In particular, the number of expressions (num_expr_{avg}) rises sharply because it is more often the case that an expression in the source code has a corresponding representation. The time required to compute the union of an expression and a descriptor ($expr\_union_{avg}$) similarly increases. There are more dirty variables in a computed descriptor and more operations are required to clean them. Each symbolic operation also becomes more expensive.

Chapter 9

Conclusion

This dissertation describes MAGNIFY, a tool that embodies a new analysis and transformation strategy to assist programmers in parallelizing scientific applications. The key advantage over existing systems is that MAGNIFY does not restrict itself to individual loop nests with a forced synchronization step after each one. Instead, MAGNIFY is selectively guided by the programmer in applying transformations that manage the interactions among the major sub-computations in the application.

The most important contribution of MAGNIFY is that it allows programmers to employ new and more powerful complex parallelization strategies that are prohibitively difficult to implement by hand. A program that incorporates the scheduling mechanisms provided by the run-time system bears little resemblance to its original form. Using MAGNIFY, the programmer continues to work from the original source code and can manipulate the control structure at a higher level. The tedious and error-prone modifications to the code are carried out automatically.

The novel set of transformations applied by MAGNIFY relies on an accurate summary of the memory access behavior of the application. This summary is provided by the symbolic data descriptor, which incorporates extensive symbolic and conditional information.

Once it has revealed opportunities for concurrency and for tuning the granularity of the application as it executes, MAGNIFY expresses the control structure in an intermediate form used by the adaptive run-time system to make efficient scheduling decisions. We have demonstrated that the system is effective on a number of real applications.

9.1 Observations

In building and working with MAGNIFY, we have come to three conclusions about the parallelization of array-based scientific applications. The first is that there is useful information available within an application that current techniques ignore but that is practical to unearth. By pursuing aggressive symbolic analysis, we have presented a strategy that discovers

opportunities for parallelism hidden from traditional analysis algorithms. Although symbolic analysis is expensive, we have shown that it is easily practical within an interactive system.

The second observation is that loop-level parallelism is necessary but often insufficient for achieving very high efficiency on large numbers of processors. While it is appropriate that automatic parallelization environments focus a great deal of attention on loops, the interactions between loop nests often contain additional sources of concurrency. The case studies demonstrate that considering only individual computations yields restricted parallelism. In order to maintain efficiency as the number of processors is increased, it is essential to manage the interactions among those computations as well.

Finally, MAGNIFY demonstrates that high-level restructuring transformations can be applied automatically under programmer guidance and achieve very high efficiency on large numbers of processors. We are convinced that such techniques are necessary to exploit massively parallel machines. For many widely used application domains, individual operations will not provide enough concurrency to use such machines efficiently.

9.2 Future Directions

There are a number of avenues of exploration that could improve both MAGNIFY and the Delirium environment in general. One interesting possibility is to rely more heavily on user assertions. The system could support both absolute declarations and guesses; the former would be automatically verifiable and the latter might be used for conditional specialization of operators. If a programmer guess about the value of a variable is found to be correct, perhaps a more efficient parallel operation could be used instead of the one that only relies on statically known information.

Another avenue to explore is the use of a transformation framework. Frameworks are being actively investigated in the parallel compiler community. The idea is to develop a systematic descriptive form that encodes many of the commonly used transformations. Then the optimizer can efficiently search the space of transformations that can be applied to the program, considering the effect of application order as it does so. Although the split-based transformations are not suitable for inclusion in such a framework, MAGNIFY could use the two strategies in tandem.

In addition, the Delirium programming environment would benefit greatly from a better user interface. Other interactive parallel programming systems like Parafrase-2[98] and Parascope[15] provide a graphical code browsing environment. A similar addition would represent a major improvement in the usability of the system.

Bibliography

[1] Walid Abu-Sufah, David J. Kuck, and D. Lawrie. "On the Performance Enhancement of Paging Systems through Program Analysis and Transformations,". *IEEE Transactions on Computers*, C-30(5):341–356, May 1981.

[2] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, Massachussetts, 1986.

[3] F. E. Allen, J. L. Carter, W. H. Harrison, P. G. Loewner, R. P. Tapscott, L. H. Trevillyan, and M. N. Wegman. "The Experimental Compiling Systems Project,". Technical report, IBM Thomas J. Watson Research Center, August 1977.

[4] Frances E. Allen and John Cocke. "A Catalogue of Optimizing Transformations,". In R. Rustin, editor, *Design and Optimization of Compilers*, pages 1–30. Prentice-Hall, Englewood Cliffs, New Jersey, 1971.

[5] John Randy Allen and Ken Kennedy. "Automatic Loop Interchange,". In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 233–246, Montreal, Quebec, June 1984. ACM Press, New York, New York.

[6] Randy Allen, Donn Baumgartner, Ken Kennedy, and Allen Porterfield. "PTOOL: A Semi-Automatic Parallel Programming Assistant,". In K. Hwang, S. M. Jacobs, and E. E. Swartzlander, editors, *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 164–170, St. Charles, Illinois, August 1986. IEEE Computer Society Press, Washington, D.C.

[7] Randy Allen and Ken Kennedy. "Automatic Translation of FORTRAN Programs to Vector Form,". *ACM Transactions on Programming Languages and Systems*, 9(4):491–542, October 1987.

[8] Steven Anderson and Paul Hudak. "Compilation of Haskell Array Comprehensions for Scientific Computing,". In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 137–149, White Plains, New York, June 1990. ACM Press, New York, New York.

[9] A. Arakawa and V. R. Lamb. "Computation Design of the UCLA General Circulation Model,". *Methods in Computational Physics*, 17:173–265, 1977.

[10] Arvind and Kim P. Gostelow. "An Asynchronous Programming Language and Computing Machine,". Technical Report TR114a, Department of Information and Computer Science, University of California, Irvine, December 1978.

[11] N.S. Asaithambi, Shen Zuhe, and R.E. Moore. "On Computing the Range of Values,". *Computing*, 28:225–237, 1982.

[12] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. "Compiler Transformations for High-Performance Computing,". *ACM Computing Surveys*, 26(4):345–420, December 1994.

[13] Henri Bal, Jennifer Steiner, and Andrew Tanenbaum. "Programming Languages for Distributed Computing Systems,". *ACM Computing Surveys*, 21(3):261–322, September 1989.

[14] Vasanth Balasundaram. "A Mechanism for Keeping Useful Internal Information in Parallel Programming Tools: the Data Access Descriptor,". *Journal of Parallel and Distributed Computing*, 9(2):154–170, June 1990.

[15] Vasanth Balasundaram, Ken Kennedy, Ulrich Kremer, Kathryn McKinley, and Jaspal Subhlok. "The ParaScope Editor: An Interactive Parallel Programming Tool,". In *Proceedings of Supercomputing '89*, pages 540–550, Reno, Nevada, November 1989. ACM Press, New York, New York.

[16] J. E. Ball. "Predicting the Effects of Optimization on a Procedure Body,". In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 214–220, Denver, Colorado, August 1979. ACM Press, New York, New York.

[17] J. E. Ball. *Program Improvement by the Selective Integration of Procedure Calls.* PhD thesis, University of Rochester, 1982.

[18] Uptal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, Boston, Massachussetts, 1988.

[19] Uptal Banerjee. "Unimodular Transformations of Double Loops,". In Alexandru Nicolau, editor, *Advances in Languages and Compilers for Parallel Processing*, Research Monographs in Parallel and Distributed Computing, chapter 10. MIT Press, Cambridge, Massachussetts, 1991.

[20] Uptal Banerjee, S. C. Chen, D. J. Kuck, and R. A. Towle. "Time and Parallel Processor Bounds for FORTRAN-Like Loops,". *IEEE Transactions on Computers*, C-28(9):660–670, September 1979.

[21] Utpal Banerjee. *Speedup of Ordinary Programs.* PhD thesis, Computer Science Department, University of Illinois at Urbana-Champaign, October 1979. Technical Report 79-989.

[22] Utpal Banerjee. "An Introduction to a Formal Theory of Dependence Analysis,". *Journal of Supercomputing*, 2(2):133–149, October 1988.

[23] John P. Banning. "An Efficient Way to Find the Side-Effects of Procedure Calls and the Aliases of Variables,". In *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages (POPL)*, pages 29–41, San Antonio, Texas, January 1979. ACM Press, New York, New York.

[24] Thomas Bemmerl. "An Integrated and Portable Tool Environment for Parallel Computers,". In F. A. Briggs, editor, *Proceedings of the International Conference on Parallel Processing (ICPP)*, volume II, pages 50–53, University Park, Pennsylvania, August 1988. Pennsylvania State University Press.

[25] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. "Distribution and Abstract Types in Emerald,". *IEEE Transactions on Software Engineering*, 13(1):65–76, January 1987.

[26] R. S. Boyer, B. Elspas, and K. N. Levitt. "SELECT — A Formal System for Testing and Debugging Programs by Symbolic Execution,". In *Proceedings of the International Conference on Reliable Software*, pages 234–244, April 1975.

[27] Michael Burke and Ron Cytron. "Interprocedural Dependence Analysis and Parallelization,". In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 162–175, Palo Alto, California, June 1986. ACM Press, New York, New York. Extended version available as IBM Thomas J. Watson Research Center Technical ReportRC 11794.

[28] D. Callahan, J. Cocke, and K. Kennedy. "Estimating interlock and improving balance for pipelined architectures,". *Journal of Parallel and Distributed Computing*, 5(4):334–358, August 1988.

[29] David Callahan, Keith Cooper, Ken Kennedy, and Linda Torczon. "Interprocedural Constant Propagation,". In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 152–161, Palo Alto, California, June 1986. ACM Press, New York, New York.

[30] David Callahan and Ken Kennedy. "Analysis of Interprocedural Side-Effects in a Parallel Programming Environment,". *Journal of Parallel and Distributed Computing*, 5(5):517–550, October 1988.

[31] Jeffery S. Chase, Franz G. Amador, Edward D. Lazowska, Henry M. Levy, and Richard J. Littlefield. "The Amber System: Parallel Programming on a Network of Multiprocessors,". In *Proceedings of the Twelfth Symposium on Operating Systems Principles (SOSP)*, pages 147–158, Litchfield Park, Arizona, December 1989. ACM Press, New York, New York.

[32] Thomas E. Cheatham, Glenn H. Holloway, and Judy A. Townley. "Symbolic Evaluation and the Analysis of Programs,". *IEEE Transactions on Software Engineering*, SE-5(4):402–417, July 1979.

[33] Marina Chen. "A Parallel Language and Its Compilation to Multiprocessor Machines or VLSI,". In *Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages (POPL)*, pages 131–139, St. Petersburg Beach, Florida, January 1986. ACM Press, New York, New York.

[34] Lori Clarke and Debra Richardson. "Applications of Symbolic Evaluation,". *The Journal of Systems and Software*, 5:15–35, 1985.

[35] Lori A. Clarke. "A System to Generate Test Data and Symbolically Execute Programs,". *IEEE Transactions on Software Engineering*, SE-2(3), September 1976.

[36] John Cocke and Jacob T. Schwartz. *Programming Languages and Their Compilers (Preliminary Notes)*. Courant Institute of Mathematical Sciences, New York University, New York, New York, second revised edition, April 1970.

[37] K. D. Cooper, M. W. Hall, and K. Kennedy. "Procedure cloning,". In *Proceedings of the International Conference on Computer Languages*, pages 96–105, Oakland, California, April 1992. IEEE Computer Society Press, Los Alamitos, California.

[38] K. D. Cooper, M. W. Hall, and L. Torczon. "An experiment with inline substitution,". *Software – Practice and Experience*, 21(6):581–601, June 1991.

[39] Keith D. Cooper and Ken Kennedy. "Efficient Computation of Flow Insensitive Interprocedural Summary Information,". In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 247–258, Montreal, Quebec, June 1984. ACM Press, New York, New York.

[40] Keith D. Cooper and Ken Kennedy. "Fast Interprocedural Alias Analysis,". In *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages (POPL)*, pages 49–59, Austin, Texas, January 1989. ACM Press, New York, New York.

[41] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. "Efficiently Computing Static Single Assignment Form and the Control Dependence Graph,". *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[42] G. B. Dantzig and B. C. Eaves. "Fourier-Motzkin Elimination and its Dual with Application to Integer Programming,". In B. Dordrecht Roy, editor, *Combinatorial Programming: Methods and Applications*, pages 93–102, Versailles, France, September 1974.

[43] K. M. Dixit. "New CPU Benchmarks from SPEC,". In *Digest of Papers, Spring COMPCON 1992, Thirty-Seventh IEEE Computer Society International Conference*, pages 305–310, San Francisco, California, February 1992. IEEE Computer Society Press, Los Alamitos, California.

[44] Jack Dongarra and A. R. Hind. "Unrolling loops in FORTRAN,". *Software – Practice and Experience*, 9(3):219–226, March 1979.

[45] Jack J. Dongarra and Danny C. Sorenson. "SCHEDULE: Tools for Developing and Analyzing Parallel FORTRAN Programs,". Technical Report 86, Argonne National Laboratory, November 1986.

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

[47] R. E. Fairley. "An Experimental Program-Testing Facility,". *IEEE Transactions on Software Engineering*, SE-1:350–357, December 1975.

[48] Gary Fielland. "The Balance Multiprocessor System,". *IEEE Micro*, 1(8):57–69, February 1988.

[49] Joseph A. Fisher, John R. Ellis, John C. Ruttenberg, and Alexandru Nicolau. "Parallel Processing: A Smart Compiler and a Dumb Machine,". In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 37–47, Montreal, Quebec, June 1984. ACM Press, New York, New York.

[50] J. Flower and A. Kolawa. "Express is Not Just a Message Passing System: Current and Future Directions in Express,". *Parallel Computing*, 20(4):597–614, April 1994.

[51] Ian Foster, Robert Olson, and Steven Tuecke. "Productive Parallel Programming: The PCN Approach,". *Scientific Programming*, 1:51–66, 1992.

[52] Dennis Gannon, William Jalby, and Kyle Gallivan. "Strategies for Cache and Local Memory Management by Global Program Transformation,". *Journal of Parallel and Distributed Computing*, 5(5):587–616, October 1988.

[53] D. Gelernter and N. Carriero. "Coordination languages and their significance,". *Communications of the ACM*, 35(2):97–107, February 1992.

[54] David Gelernter. "Parallel Programming in Linda,". In *Proceedings of the International Conference on Parallel Processing*, pages 255–263, August 1985.

[55] David K. Gifford and John M. Lucassen. "Integrating Functional and Imperative Programming,". In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 28–38, Cambridge, Massachussetts, August 1986. ACM Press, New York, New York.

[56] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. "Practical Dependence Testing,". In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 15–29, Toronto, Ontario, June 1991. ACM Press, New York, New York.

[57] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick. "An Execution Profiler for Modular Programs,". *Software–Practice & Experience*, 13:671–685, August 1983.

[58] M. Haghighat and C. Polychronopoulos. "Symbolic program analysis and optimization for parallelizing compilers,". In U. Banerjee, D. Gelernter, A. Nicolau, and D. A. Padua, editors, *Proceedings of the Fifth International Workshop on Languages and Compilers for Parallel Computing*, volume 757 of *Lecture Notes in Computer Science*, pages 538–562, New Haven, Connecticut, August 1992. Springer-Verlag, Berlin, Germany.

[59] M. W. Hall, K. Kennedy, and K. S. McKinley. "Interprocedural transformations for parallel code generation,". In *Proceedings of Supercomputing '91*, pages 424–434, Albuquerque, New Mexico, November 1991. IEEE Computer Society Press, Los Alamitos, California.

[60] S. L. Hantler and J. C. King. "An Introduction to Proving the Correctness of Programs,". *ACM Computing Surveys*, 8:331–353, September 1976.

[61] William Harrison. "Compiler Analysis of the Value Ranges for Variables,". *IEEE Transactions on Software Engineering*, SE-3(3):243–250, May 1977.

[62] P. Havlak and K. Kennedy. "Experience with interprocedural analysis of array side effects,". In *Proceedings of Supercomputing '90*, pages 952–961, New York, New York, November 1990. IEEE Computer Society Press, Los Alamitos, California.

[63] Kaarlo Heiskanen. *Tomography with Limited Data in Fan Beam Geometry.* PhD thesis, University of California, Berkeley, February 1990.

[64] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. "Compiler Optimizations for FORTRAN D on MIMD Distributed Memory Machines,". In *Proceedings of Supercomputing '91*, pages 86–100, Albuquerque, New Mexico, November 1991. IEEE Computer Society Press, Los Alamitos, California.

[65] Paul Hudak and Lauren Smith. "Para-functional Programming: A Paradigm for Programming Multiprocessor Systems,". In *Conference Record of the Thirteenth ACM Symposium on Principles of Programming Languages (POPL)*, pages 243–254, St. Petersburg Beach, Florida, January 1986. ACM Press, New York, New York.

[66] Paul Hudak and P. Wadler. "Report on the Programming Language Haskell,". Technical Report YALU/DCS/RR666, Computer Science Department, Yale University, November 1988.

[67] Susan Hummel, Edith Schonberg, and Lawrence Flynn. "Factoring: A Practical and Robust Method for Scheduling Parallel Loops,". In *Proceedings of Supercomputing '91*, pages 610–619, Albuquerque, New Mexico, November 1991. IEEE Computer Society Press, Los Alamitos, California.

[68] Eric Jul, Henry Levy, Norman Hutchinson, and Andrew Black. "Fine-Grained Mobility in the Emerald System,". *ACM Transactions on Computer Systems*, 6(1):109–133, 1988.

[69] K. Kennedy, K. S. McKinley, and C.-W. Tseng. "Analysis and transformation in an interactive parallel programming tool,". *Concurrency: Practice and Experience*, 5(7):575–602, October 1993.

[70] James King. "Symbolic Execution and Program Testing,". *Communications of the ACM*, 19(7):385–394, July 1976.

[71] C. Koelbel and P. Mehrotra. "Programming data parallel algorithms on distributed memory machines using Kali,". In *Proceedings of the ACM International Conference on Supercomputing*, pages 414–423, Cologne, Germany, June 1991. ACM Press, New York, New York.

[72] C. Kruskal and A. Weiss. "Allocating Independent Subtasks on Parallel Processors,". *IEEE Transactions on Software Engineering*, SE-11, October 1985.

[73] Monica S. Lam. "Software Pipelining: An Effective Scheduling Technique for VLIW Machines,". In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 318–328, Atlanta, Georgia, June 1988. ACM Press, New York, New York.

[74] Monica S. Lam, Edward E. Rothberg, and Michael E. Wolf. "The Cache Performance and Optimization of Blocked Algorithms,". In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 63–74, Santa Clara, California, April 1991. ACM Press, New York, New York.

[75] Thomas Lengauer and Robert Endre Tarjan. "A Fast Algorithm for Finding Dominators in a Flowgraph,". *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[76] K. Li and P. Hudak. "Memory Coherence in Shared Virtual Memory Systems,". In *Proceedings of the Fifth Annual ACM Symposium on Principles of Distributed Computing*, pages 229–239, 1986.

[77] Zhiyuan Li and Pen-Chung Yew. "Interprocedural Analysis for Parallel Computing,". In F. A. Briggs, editor, *Proceedings of the International Conference on Parallel Processing (ICPP)*, volume 2, pages 221–228, University Park, Pennsylvania, August 1988. Pennsylvania State University Press.

[78] Zhiyuan Li and Pen-Chung Yew. "Program Parallelization with Interprocedural Analysis,". *Journal of Supercomputing*, 2(2):225–244, October 1988.

[79] Zhiyuan Li, Pen-Chung Yew, and C. Zhu. "Data Dependence Analysis on Multi-Dimensional Array References,". *IEEE Transactions on Parallel and Distributed Systems*, 1(1):26–34, January 1990.

[80] A. Lichnewsky and F. Thomasset. "Introducing Symbolic Problem Solving Techniques in the Dependence Testing Phases of a Vectorizer,". In *Proceedings of the ACM*

International Conference on Supercomputing, pages 396–406, St. Malo, France, July 1988. ACM Press, New York, New York.

[81] David B. Loveman. "Program Improvement by Source-to-Source Transformation,". *Journal of the ACM*, 1(24):121–145, January 1977.

[82] Steven Lucco. "Parallel Programming in a Virtual Object Space,". In *Proceedings of the 1987 ACM Conference on Object Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Orlando, Florida, October 1987. ACM Press, New York, New York.

[83] Steven Lucco. "A Dynamic Scheduling Method for Irregular Parallel Programs,". In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 200–211, San Francisco, California, June 1992. ACM Press, New York, New York.

[84] Steven Lucco. *Adaptive Parallel Programs.* PhD thesis, Computer Science Division, University of California, Berkeley, 1994.

[85] Steven Lucco and Oliver Sharp. "Delirium: An Embedding Coordination Language,". In *Proceedings of Supercomputing '90*, pages 515–524, New York, New York, November 1990. IEEE Computer Society Press, Los Alamitos, California.

[86] Steven Lucco and Oliver Sharp. "Parallel Programming With Coordination Structures,". In *Conference Record of the Eighteenth ACM Symposium on Principles of Programming Languages (POPL)*, Orlando, Florida, January 1991. ACM Press, New York, New York.

[87] D. E. Maydan, J. L. Hennessy, and M. S. Lam. "Efficient and Exact Data Dependence Analysis,". In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 1–14, Toronto, Ontario, June 1991. ACM Press, New York, New York.

[88] James R. McGraw. "The VAL Language: Description and Analysis,". *ACM Transactions on Programming Languages and Systems*, 4(1):44–82, January 1982.

[89] James R. McGraw. "SISAL: Streams and Iteration in a Single Assignment Language,". Technical Report M-146, Lawrence Livermore National Laboratory, March 1985.

[90] Michael Metcalf and John Reid. *FORTRAN 90 Explained.* Oxford University Press, New York, 1992.

[91] Ravi Mirchandaney, Joel H. Saltz, Roger M. Smith, David M. Nicol, and Kay Crowley. "Principles of Runtime Support for Parallel Processors,". In *Proceedings of the ACM International Conference on Supercomputing*, pages 140–152, St. Malo, France, July 1988. ACM Press, New York, New York.

[92] Y. Muraoka. *Parallelism Exposure and Exploitation in Programs.* PhD thesis, University of Illinois at Urbana-Champaign, February 1971. Technical Report 71-424.

[93] Eugene W. Myers. "A Precise Interprocedural Data Flow Algorithm,". In *Conference Record of the Eighth ACM Symposium on Principles of Programming Languages (POPL)*, pages 219–230, Williamsburg, Virginia, January 1981. ACM Press, New York, New York.

[94] Rishiyur S. Nikhil. "ID Reference Manual, version 88.0,". Technical Report 284, Laboratory for Computer Science, Massachussets Institute of Technology, 1988.

[95] David Notkin, Lawrence Snyder, David Socha, Mary L. Bailey, Bruce Forstall, Kevin Gates, Ray Greenlaw, William G. Griswold, Thomas J. Holman, Richard Korry, Gemimi Lasswell, Robert Mitchell, and Philip A. Nelson. "Experiences with Poker,". In *Proceedings of the ACM/SIGPLAN PPEALS Symposium on Parallel Programming: Experience with Applications, Languages, and Systems*, pages 10–20, New Haven, Connecticut, September 1988. ACM Press, New York, New York.

[96] Constantine D. Polychronopoulos. "Loop Coalescing: A Compiler Transformation for Parallel Machines,". In Sartaj K. Sahni, editor, *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 235–242, University Park, Pennsylvania, August 1987. Pennsylvania State University Press.

[97] Constantine D. Polychronopoulos. *Parallel Programming and Compilers.* Kluwer Academic Publishers, Boston, Massachussetts, 1988.

[98] Constantine D. Polychronopoulos, Milind Girkar, Mohammad Reza Haghighat, Chia Ling Lee, Bruce Leung, and Dale Schouten. "Parafrase-2: An Environment for Parallelizing, Partitioning, Synchronizing, and Scheduling Programs on Multiprocessors,". In *Proceedings of the International Conference on Parallel Processing (ICPP)*, volume II, pages 39–48, University Park, Pennsylvania, August 1989. Pennsylvania State University Press.

[99] Constantine D. Polychronopoulos and David J. Kuck. "Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers,". *IEEE Transactions on Computers*, C-36(12):1425–1439, December 1987.

[100] Terrence W. Pratt. "The Pisces 2 Parallel Programming Environment,". In Sartaj K. Sahni, editor, *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 439–445, University Park, Pennsylvania, August 1987. Pennsylvania State University Press.

[101] W. Pugh. "A practical algorithm for exact array dependence analysis,". *Communications of the ACM*, 35(8):102–115, August 1992.

[102] William Pugh. "Uniform Techniques for Loop Optimization,". In *Proceedings of the ACM International Conference on Supercomputing*, Cologne, Germany, June 1991. ACM Press, New York, New York.

[103] Jonathan Rees, William Clinger, et al. "Revised^3 Report on the Algorithmic Language SCHEME,". *SIGPLAN Notices*, 21(12):37–79, December 1986.

[104] S. K. Reinhardt, M. D. Hill, J. R. Larus, A. R. Lebeck, J. C. Lewis, and D. A. Wood. "The Wisconsin Wind Tunnel: virtual prototyping of parallel computers,". In *1993 ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*, pages 48–60, Santa Clara, CA, May 1993.

[105] Martin C. Rinard, Daniel J. Scales, and Monica S. Lam. "Jade: A High-Level Machine-Independent Language for Parallel Programming,". *Computer*, 26(6):28–38, June 1993.

[106] Gary Sabot and Skef Wholey. "CMAX: A FORTRAN Translator for the Connection Machine System,". In *Proceedings of the ACM International Conference on Supercomputing*, Tokyo, Japan, July 1993. ACM Press, New York, New York.

[107] Vivek Sarkar. "Determining Average Program Execution Times and Their Variance,". In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 298–312, Portland, Oregon, June 1989. ACM Press, New York, New York.

[108] Vivek Sarkar and John Hennessy. "Partitioning Parallel Programs for Macro Dataflow,". In *Proceedings of the 1986 ACM Conference on LISP and Functional Programming*, pages 202–211, Cambridge, Massachussetts, August 1986. ACM Press, New York, New York.

[109] Vivek Sarkar and Radhika Thekkath. "A General Framework for Iteration-Reordering Transformations,". In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 175–187, San Francisco, California, June 1992. ACM Press, New York, New York.

[110] R. W. Scheifler. "An Analysis of Inline Substitution for a Structured Programming Language,". *Communications of the ACM*, 20(9):647–654, September 1977.

[111] Oliver Sharp. "Pythia: A Parallel Compiler for Delirium,". Master's thesis, Computer Science Division, University of California, Berkeley, 1991.

[112] S. Sobek, M. Azam, and J. C. Browne. "Architectural and Language Independent Parallel Programming: A Feasibility Demonstration,". In F. A. Briggs, editor, *Proceedings of the International Conference on Parallel Processing (ICPP)*, volume II, pages 80–83, University Park, Pennsylvania, August 1988. Pennsylvania State University Press.

[113] K. Sridharan, M. McShea, C. Denton, B. Eventoff, J. C. Browne, P. Newton, M. Ellis, D. Grosshard, T. Wise, and D. Clemmer. "An Environment for Parallel Structuring of FORTRAN Programs,". In *Proceedings of the International Conference on Parallel Processing (ICPP)*, volume II, pages 98–106, University Park, Pennsylvania, August 1989. Pennsylvania State University Press.

[114] Sun Microsystems. *SPARC Architecture Manual, Version 8*, 1991. Part No. 800-1399-08.

[115] V. S. Sunderam. "PVM: A Framework for Parallel Distributed Computing,". *Concurrency: Practice and Experience*, 2(4):315–339, December 1990.

[116] Peiyi Tang and Pen-Chung Yew. "Dynamic Processor Self-Scheduling for General Parallel Nested Loops,". *IEEE Transactions on Computers*, C-39(7):919–929, July 1990.

[117] Ross A. Towle. *Control and Data Dependence for Program Transformations.* PhD thesis, Computer Science Department, University of Illinois at Urbana-Champaign, March 1976. Technical Report 76-788.

[118] Remi Triolet, Francois Irigoin, and Paul Feautrier. "Direct Parallelization of Call Statements,". In *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 176–185, Palo Alto, California, June 1986. ACM Press, New York, New York.

[119] United States Department of Defense. *Reference Manual for the Ada Programming Language*, ANSI/MIL-STD-1815-1983 edition, February 1983.

[120] D. W. Walker. "The Design of a Standard Message Passing Interface for Distributed Memory Concurrent Computers,". *Parallel Computing*, 20(4):657–673, April 1994.

[121] P. K. Weiner and P. A. Kollman. "AMBER: Assisted model building with energy refinement. A general program for modeling molecules and their interactions,". *Journal of Computational Chemistry*, 2(3):287–303, Fall 1981.

[122] M. Weiss, C. R. Morgan, P. Belmont, and Z. Fang. "Dynamic Scheduling and Memory Management for Parallel Programs,". In F. A. Briggs, editor, *Proceedings of the International Conference on Parallel Processing (ICPP)*, pages 161–165, University Park, Pennsylvania, August 1988. Pennsylvania State University Press.

[123] Michael E. Wolf and Monica S. Lam. "A Loop Transformation Theory and an Algorithm to Maximize Parallelism,". *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[124] Michael J. Wolfe. "More Iteration Space Tiling,". In *Proceedings of Supercomputing '89*, pages 655–664, Reno, Nevada, November 1989. ACM Press, New York, New York.

[125] Michael J. Wolfe. *Optimizing Supercompilers for Supercomputers.* Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Massachussetts, October 1989. Based on the author's Ph.D. thesis at the University of Illinois at Urbana-Champaign, 1982.

[126] Michael J. Wolfe and C. Tseng. "The Power Test for Data Dependence,". *IEEE Transactions on Parallel and Distributed Systems*, 3(5):591–601, September 1992.