PARALLELIZING COMPILER TECHNIQUES BASED ON LINEAR INEQUALITIES

A DISSERTATION SUBMITTED TO
THE DEPARTMENT OF ELECTRICAL ENGINEERING
AND THE COMMITTEE ON GRADUATE STUDIES
OF STANFORD UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

Saman Prabhath Amarasinghe

January 1997

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Monica S. Lam
(Principal Adviser)

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

John L. Hennessy

I certify that I have read this dissertation and that in my opinion it is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

Anoop Gupta

Approved for the University Committee on Graduate Studies:

Dean of Graduate Studies

Abstract

Shared-memory multiprocessors, built out of the latest microprocessors, are becoming a widely available class of computationally powerful machines. These affordable multiprocessors can potentially deliver supercomputer-like performance to the general public.

To effectively harness the power of these machines it is important to find all the available parallelism in programs. The Stanford SUIF interprocedural parallelizer we have developed is capable of detecting coarser granularity of parallelism in sequential scientific applications than previously possible. Specifically, it can parallelize loops that span numerous procedures and hundreds of lines of codes, frequently requiring modifications to array data structures such as array privatization. Measurements from several standard benchmark suites demonstrate that aggressive interprocedural analyses can substantially advance the capability of automatic parallelization technology.

However, locating parallelism is not sufficient in achieving high performance. It is critical to make effective use of the memory hierarchy. In parallel applications, false sharing and cache conflicts between processors can significantly reduce performance. We have developed the first compiler that automatically performs a full suite of data transformations (a combination of transposing, strip-mining and padding). The performance of many benchmarks improves drastically after the data transformations.

We introduce a framework based on systems of linear inequalities for developing compiler algorithms. Many of the whole program analyses and aggressive optimizations in our compiler employ this framework. Using this framework general solutions to many compiler problems can be found systematically.

To my late father

Dharmasiri Amarasinghe

Acknowledgments

I would like to thank my advisor Monica Lam, without whose unbounded energy, immense enthusiasm, demand for excellence and intellectual vigor (and many all-nighters) this research would not be possible. Krishna Saraswat graciously agreed to be my orals chairman. John Hennessy and Anoop Gupta provided invaluable comments as members of my thesis and orals committees. It was a deep honor and a privilege to receive wisdom and guidance from John Hennessy.

My time at Stanford was spent on several joint projects with many talented and brilliant individuals. The early work on array privatization was done with Dror Maydan, who is a pleasure to work with. The locality optimization work was done with Jennifer Anderson, a valued friend and a great collaborator. The interprocedural parallelizer project was a joint effort with Mary Hall, Brian Murphy, and Shih-Wei Liao. Mary is a great motivator and it was lively and fun to work with her. I greatly enjoyed and learned from my work with the SUIF compiler team, especially Jennifer Anderson, Robert French, Mary Hall, David Heine, Shih-Wei Liao, Amy Lim, Dror Maydan, Brian Murphy, Jason Nieh, Martin Rinard, Patrick Sathyanathan, Dan Scales, Alex Seibulescu, Mike Smith, Steve Tjiang, Chau-Wen Tseng, Bob Wilson, Chris Wilson, and Michael Wolf.

Jeff Erickson, Mary Hall, David Heine, Michael Margolis, and Mary McDevitt read many drafts of this thesis and provided me with invaluable comments.

I was fortunate to have a great collection of family, friends, and Hammaraskjöld buddies who kept me relatively sane during my time at Stanford. Chand Samaratunga, Nalin Kulatilaka, Hemantha Jayawardana and the rest of the crew at Lanka Internet Services, Ltd. provided me with a challenging distraction during my last few years at Stanford. I am especially in debt to my wife Pranita who provided constant support, strength and companionship when it was most needed.

This work was supported in part by DARPA contracts DABT63-91-K-0003, and DABT63-94-C-0054 and a graduate fellowship from Intel Corporation. Beatrice Fu was my mentor at Intel.

Table of Contents

List of Figures

1 Introduction

Shared-memory multiprocessors, built out of the latest microprocessors, are now becoming widely used as medium- and high-powered servers. These affordable multiprocessors can potentially deliver supercomputer-like performance to the general public. Today, these machines are mainly used in a multi-programming mode, increasing system throughput by running several independent applications in parallel. The multiple processors can also be used together to accelerate the execution of single applications. Automatic parallelization is a promising technique that allows ordinary sequential programs to take advantage of multiprocessors [24,43,71,87].

Multiprocessors present more difficult challenges to parallelizing compilers than do vector machines, their initial target. Effective use of a vector architecture involves parallelizing repeated arithmetic operations on large data streams (*e.g.*, innermost loops in array-oriented programs). On a multiprocessor, however, parallelizing innermost loops typically does not provide sufficiently large *granularity of parallelism* —not enough work is performed in parallel to overcome the overhead of synchronization and communication among processors. To utilize a multiprocessor effectively, the compiler must exploit *coarse-grain parallelism*, locating large computations that can execute independently in parallel.

Locating coarse-grain parallelism is not sufficient to obtain parallel performance. It is critical to make effective use of the memory hierarchy to achieve high performance. Over the last decade, microprocessor speeds have been steadily improving at a rate of 50% to 100% every year [82]. Meanwhile, memory access times have been improving at the rate of only 7% per year [82]. A common technique used to bridge this gap between processor and memory speeds is to employ one or more levels of caches. However, it has been notoriously difficult to use caches effectively for numeric applications. In fact, various past machines built for scientific computations—such as the Cray C90, Cydrome Cydra-5 [126] and the Multiflow Trace [42]—were all built without caches. However, current multiprocessor systems include complex memory hierarchies and multiple levels of caches. Given that the processor-memory gap continues to widen, exploiting the memory hierarchy is critical to achieving high performance on modern architectures.

1.1. Thesis Overview

1.1.1. Framework for Compiler Development

A successful parallelizing compiler needs to perform many whole program analyses and aggressive optimizations. Creating a compiler that is capable of performing these analyses and optimizations on an arbitrary program, written in one of many programming styles, is a daunting task for compiler writers. One important method used by compiler writers to tackle the complexity of the development process is to take advantage of proven frameworks. We introduce one such framework for parallelizing compilers based on systems of linear inequalities. Many of the problems in parallelizing compilers for scientific applications involve comprehensive analysis and aggressive optimizations on loop nests and data arrays. The iteration space of the loop nests, the data space of the arrays, and the index space of the processors are multi-dimensional integer spaces, and thus can be represented using systems of linear inequalities. We show the usefulness of this framework by applying it in developing many advanced analysis and optimization techniques.

1.1.2. Locating Coarse-Grain Parallelism

Finding coarse-grain parallelism requires major improvements over standard analysis for parallelization. A loop is often not parallelizable unless the compiler modifies the data structures it accesses. For example, it is very common for each iteration of a loop to define and use the same variable. The compiler must give each processor a private copy of the variable for the loop to be parallelizable. The compiler needs to perform array data-flow analysis to determine if an array is privatizable [52,113]. We have developed a unified array analysis algorithm using an array summary representation based on the linear inequalities framework. Using this representation, we calculate data-flow information more accurately than any other previous analysis, and we also perform the data-dependence analysis at the same precision as the exact data dependence test.

Furthermore, the existence of array reshapes in FORTRAN, where the same memory locations are accessed using different array shapes, further complicates interprocedural array analysis. In order to perform the aggressive whole program analysis, required to find coarse-grain parallelism, the compiler must analyze the programs in the presence of array reshapes to determine their effect on the rest of the analysis. Previously, array reshapes were handled only within a limited domain [137]. We have developed a linear inequalities-based algorithm that can analyze a large class of array reshapes.

Using these advanced array analysis techniques we have developed a fully functional interprocedural parallelizer in the Stanford SUIF compiler system that is capable of detecting coarse-grain parallelism. We show that automatic parallelization can succeed with many existing sequential dense matrix scientific applications by applying our compiler to more than 115,000 lines of FORTRAN code in 39 programs from four benchmark suites.

1.1.3. Optimizing for the Memory Hierarchy

The effective utilization of the memory hierarchy is critical to achieving high performance. Recent work on code transformations to improve cache performance has been shown to improve uniprocessor system performance significantly [33,147]. Making effective use of the memory hierarchy on multiprocessors is even more important to performance but also more difficult to achieve.

We have developed the first compiler that automatically performs a full suite of data transformations on the original array layouts to improve memory system performance of cache-coherent multiprocessors. Our algorithm restructures the layout of the data in the shared address space such that each processor is assigned a contiguous segment of memory. We ran our compiler on a set of application programs and measured their performance. Our results show that the compiler can effectively optimize for parallelism and memory subsystem performance.

1.1.4. Optimizing Communication

We have developed a systematic approach, based on the linear inequalities framework, for code generation and optimization of communication for distributed memory machines. This problem involves manipulation of all three spaces: iteration, data and processor. It also demonstrates the flexibility and usefulness of the linear inequalities framework. This framework can handle a large class of computation and data decompositions as well as complex array access functions. We represent data decompositions, computation decompositions, and inter-processor communication as systems of linear inequalities. We have also developed several communication optimizations within the same unified framework. These optimizations include eliminating redundant messages, aggregating messages, and hiding communication latency by overlapping communication with computation.

1.2. Organization of the Thesis

The organization of this thesis is as follows. In Chapter 2, we introduce our framework for parallelizing compilers based on systems of linear inequalities. We discuss the need for coarse-grain parallelism and the requirements for obtaining it in Chapter 3. We present our array data-flow algorithm in Chapter 4 and the linear inequalities-based summary representation in Chapter 5. In Chapter 6, we introduce a linear inequalities-based algorithm that can analyze a large class of array reshapes. In Chapter 7, we show that automatic parallelization can succeed with many existing sequential dense matrix scientific applications by applying our compiler to more than 115,000 lines of FORTRAN code in 39 programs from four benchmark suites. The unique problems posed by multiprocessor caches are discussed in Chapter 8. We introduce a data transformation algorithm that changes the original array layouts to improve memory system performance. A collection of communication code generation and communication optimization algorithms for distributed address-space machines is defined in Chapter 9. We conclude in Chapter 10.

2 The Linear Inequalities Framework

The first generation of compilers was capable only of a simple translation of programs written in a high-level programming language into a low-level machine language. However, modern compilers perform many complex transformations that are necessary to optimize programs to obtain good performance from today's complex computers. Creating a compiler that is capable of performing these complex transformations on an arbitrary program, written in one of many programming styles, is a daunting task for the compiler writer. Compilers have become very large and complex software systems that require highly skilled compiler writers and many people-years of development. One important method used by compiler writers to tackle the complexity of the development process is to take advantage of proven frameworks. Use of tools such as parser generators [96] and data-flow frameworks [90] can help create robust and powerful compilers with relative ease.

The next generation of compilers, aimed at parallel architectures, such as shared memory multiprocessors, needs to perform even more complex whole program analysis techniques and aggressive optimizations. A framework that can be used to develop many of the new analyses and optimizations is essential to the success of these parallelizing compilers. The framework should be robust and applicable to a wide class of input programs. Compiler writers should be able to use this framework to create effective general solutions in a systematic manner. In this chapter, we introduce one such framework based on systems of linear inequalities.

Many of the critical requirements of parallelizing compilers for scientific applications involve comprehensive analyses and aggressive optimizations on loop nests and data arrays. By representing the iteration space of the loop nests and the data space of the arrays as multi-dimensional integer spaces, we can perform these novel analyses and optimizations through a mathematical manipulation of the spaces. The compiler can analyze an input program by creating index sets associated with the spaces and perform optimizations by manipulating these index sets. Representing arbitrary sets of coordinates accurately is not practical in a compiler. However, many of the iteration and data spaces found in practice are multi-dimensional convex regions. Thus, we focus on the domain of index sets that can be represented using convex polyhedrons.

This chapter is organized as follows. In the next section we will define the linear inequality representation used throughout this thesis. We introduce the use of this framework by describing a code generation algorithm in Section 2.2. We have extended linear inequalities, as described in Section 2.3, to handle simple non-linear systems with symbolic coefficients. The data and processor spaces used in this thesis are introduced in Section 2.4. We present related work in Section 2.5.

2.1. Systems of Linear Inequalities

We use a unified framework based on linear inequalities to handle multi-dimensional integer spaces such as iteration, data and processor spaces that are used in analyses and optimization techniques for next-generation compilers [9]. We represent all possible values of a set of integer variables $(v_1, \ldots, v_n) \in Z^n$ as an n-dimensional discrete cartesian space, where the k-th axis corresponds to variable v_k. Coordinate $[x_1, \ldots, x_n] \in Z^n$ corresponds to the value $v_1 = x_1, \ldots, v_n = x_n$.

A parameterized convex polyhedron in the n-dimensional space of the variables $v_1, \ldots, v_n$, parameterized by symbolic constants $u_1, \ldots, u_k$, is represented by a system of linear inequalities with the variables $v_1, \ldots, v_n$ and the symbolic constants $u_1, \ldots, u_k$. All the solutions satisfying the inequalities correspond to the integer points within the polyhedron.

Definition 2-1: *A parameterized convex polyhedron* $S^n : Z^k \rightarrow P\left(Z^n \right)$ *of n dimensions and k parameters is represented by the system of inequalities*

$$S^n(u_1, \ldots, u_k) = \left\{ (v_1, \ldots, v_n) \;\middle|\; \begin{array}{c} a^1 + b_1^1 u_1 + \ldots + b_k^1 u_k + c_1^1 v_1 + \ldots + c_n^1 v_n \geq 0 \\ \ldots\ldots\ldots\ldots\ldots \\ a^m + b_1^m u_1 + \ldots + b_k^m u_k + c_1^m v_1 + \ldots + c_n^m v_n \geq 0 \end{array} \right\}$$

where all a's, b's and c's are integers, $u_1, \ldots, u_k$ *are integer symbolic constants and* $v_1, \ldots, v_n$ *are integer variables.*

In our compiler algorithms, we use projection as one of the key transformations in manipulating systems of linear inequalities [47]. Suppose we project an*n*-dimension polyhedron, S^n, onto the $(n-1)$-dimensional subspace orthogonal to the axis representing variable v_n. The resulting polyhedron in the $(n-1)$-dimensional subspace, S^{n-1}, is derived by eliminating the variable v_n from the system of inequalities of S^n.

Projection of an *n*-dimensional polyhedron onto an $(n-1)$-dimensional space can be achieved using a single step of Fourier-Motzkin elimination [48,127]. Fourier-Motzkin elimination can produce a large number of superfluous constraints. We can determine if a constraint is superfluous as follows. We replace the constraint in question with its negation, and if the new system does not have an integer solution then the constraint is superfluous. To check if a system has an integer solution, we again use Fourier-Motzkin elimination. Since the Fourier-Motzkin elimination algorithm checks if a real solution exists for a system, a branch-and-bound technique is needed to check for the existence of an integer solution [127].

Each integer point in the original polyhedron is mapped to an integer point in the polyhedron created by the projection operation. However, the projected polyhedron may contain integer points with no corresponding points in the original polyhedron. If $[x_1, \ldots, x_n] \in S^n$ then $[x_1, \ldots, x_{n-1}] \in S^{n-1}$. But, given $[x_1, \ldots, x_{n-1}] \in S^{n-1}$, there may or may not exist an x_n such that $[x_1, \ldots, x_n] \in S^n$. Consider the example where S^n has a single constraint involving v_n: $v_1 = 2v_n$. We know that v_1 must be even. However, this constraint is not captured in the projected polyhedron S^{n-1} and v_1 can be an odd number in S^{n-1}.

2.2. Code Generation

As a simple example of how linear inequalities framework can be used in compilers, we present a code generation algorithm based on linear inequalities [9]. Ancourt and Irigoin presented an algorithm for generation of loop nests after loop transformation by a series of projections of the transformed iteration space [10,11]. In the following, we briefly describe their algorithm, and our heuristics for finding tight and efficient loop bounds.

2.2.1. Iteration Space

The iterations of an n-deep loop nest are given by an iteration set where each element is an iteration in the iteration space $\mathfrak{I} \subseteq Z^n$.

A parameterized convex polyhedron can be used to represent the iteration space of a loop nest when the loop bounds of the nest are affine expressions of outer loop indices and symbolic constants. Within this scope of a loop nest, the convex polyhedron representing the iteration space is formally defined as follows:

Definition 2-2: *For the n-deep loop nest*

```
DO $i_1$ = $l_1(v_1, \ldots, v_m)$ , $h_1(v_1, \ldots, v_m)$
  DO $i_2$ = $l_2(v_1, \ldots, v_m, i_1)$ , $h_2(v_1, \ldots, v_m, i_1)$
   ....
     DO $i_n$ = $l_n(v_1, \ldots, v_m, i_1, \ldots, i_{n-1})$ , $h_n(v_1, \ldots, v_m, i_1, \ldots, i_{n-1})$
```

where $v_1, \ldots, v_m$ are symbolic constants (variables unchanged within the loop), $i_1, \ldots, i_n$ are the loop index variables and l_k, h_k are affine functions, the iteration set, $I^n(v_1, \ldots, v_m)$, is given by the parameterized convex polyhedron

$$I^n(v_1, \ldots, v_m) = \left\{ (i_1, \ldots, i_n) \in \mathfrak{I} \;\middle|\; \bigwedge_{k=1, \ldots, n} \begin{array}{l} i_k \geq l_k\left(v_1, \ldots, v_m, i_1, \ldots, i_{k-1}\right) \\ i_k \leq h_k\left(v_1, \ldots, v_m, i_1, \ldots, i_{k-1}\right) \end{array} \right\}$$

Figure 2-2 shows the system of inequalities describing the iteration space of the example loop nest in Figure 2-1. Each integer point within the convex polyhedron corresponds to a valid iteration of the loop nest. The graphical representation of the convex polyhedron of the iteration space is illustrated in Figure 2-3.

```
DO I = 1, N
     DO J = 1, I
          DO K = J, 2N-I
               .....
```

Figure 2-1. Example loop nest

$$I^3(N) = \left\{ (I, J, K) \;\middle|\; \begin{array}{cc} I-1 \geq 0 & N-I \geq 0 \\ J-1 \geq 0 & I-J \geq 0 \\ K-J \geq 0 & 2N-I-K \geq 0 \end{array} \right\}$$

Figure 2-2. System of inequalities describing the iteration space

2.2.2. Scanning a Polyhedron

The iteration space representation of a loop nest does not specify the order of execution of the iterations. When generating a loop nest from an iteration space, we need to provide a *lexicographical* order for execution of the iterations.

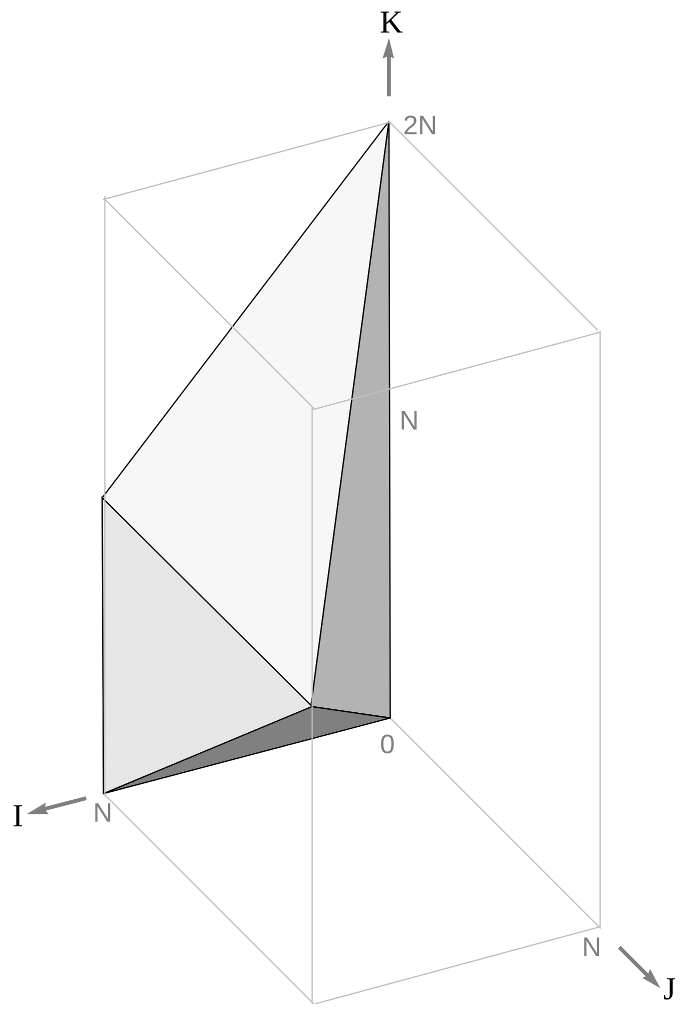


Figure 2-3. Convex polyhedron representing the iteration space

Definition 2-3: *Given iterations* $(i_1, \ldots, i_n)$, $(j_1, \ldots, j_n) \in Z^n$, $(i_1, \ldots, i_n)$ *is* ***lexicographically less than*** $(j_1, \ldots, j_n)$ *iff there exists* $k \le n$ *such that* $\forall 0 \le l < k$ $i_l = j_l$ *and* $i_k < j_k$.

We generate a loop nest from a parameterized convex polyhedron, $S^n(u_1, \ldots, u_m)$, with symbolic constants $u_1, \ldots, u_m$ and unknowns $v_1, \ldots, v_n$. The integer points within the polyhedron are converted to iterations of the loop nest. The order in which the points are visited, the lexicographical ordering of the iterations, is given by the*scanning order.* The scanning order $(v_1, \ldots, v_n)$ indicates that the index variable of the*k*-th outermost loop is v_k. The index of a loop is incremented by one every iteration, and has a finite lower and upper bound. The loop bounds are expressions of symbolic constants$u_1, \ldots, u_m$ and outer loop indices. The problem that remains is, what should the bounds of the loops be such that the loop nest contains an iteration with indices $[x_1, \ldots, x_n]$ iff $[x_1, \ldots, x_n]$ is a solution to S^n?

We find the bounds of the loop nest in the reverse scanning order. To find the bounds for loop index v_n, we rewrite the constraints in the form of$c_l^k v_n \geq l^k(u_1, \ldots, u_m, v_1, \ldots, v_{n-1})$ and $c_h^k v_n \leq h^k(u_1, \ldots, u_m, v_1, \ldots, v_{n-1})$. Any inequalities not involving v_n need not be considered here. The integer lower and upper bounds for v_n are given simply by

$$\underset{k}{MIN}\left\lceil \frac{l^k(u_1, \ldots, u_m, v_1, \ldots, v_{n-1})}{c_l^k} \right\rceil \leq v_n \leq \underset{k}{MIN}\left\lfloor \frac{h^k(u_1, \ldots, u_m, v_1, \ldots, v_{n-1})}{c_h^k} \right\rfloor$$

We next project the original polyhedron onto the $(v_1, \ldots, v_{n-1})$ space to obtain an $(n-1)$-dimensional parameterized convex polyhedron represented by a set of constraints involving the symbolic constants $u_1, \ldots, u_m$ and the variables $v_1, \ldots, v_{n-1}$. We can then repeat the process in the reverse scanning order for variables $v_{n-1}, \ldots, v_1$.

The system of inequalities in Figure 2-2 represents a three-dimensional iteration space. For this iteration space, six possible scanning orders can be used to generate a loop nest. The projections necessary for generating loop nests for all the six scanning orders are illustrated in Figure 2-4. The six loops nests generated are given in Figure2-5. Each loop is marked with the projection number that created the loop bounds. The three-dimensional polyhedron of the original iteration space has three possible projections, resulting in three two-dimensional polyhedrons(planes). The inequalities of the projected variable are the bounds of the inner loops. Each of the three planes have two possible projections, creating six one-dimensional polyhedrons(lines). These six lines provide the bounds for the outer loop. Note that the lexicographical order of the original loop nest in Figure2-1 is given by the scanning order (I, J, K). A loop nest with the same scanning order is generated by the projec-

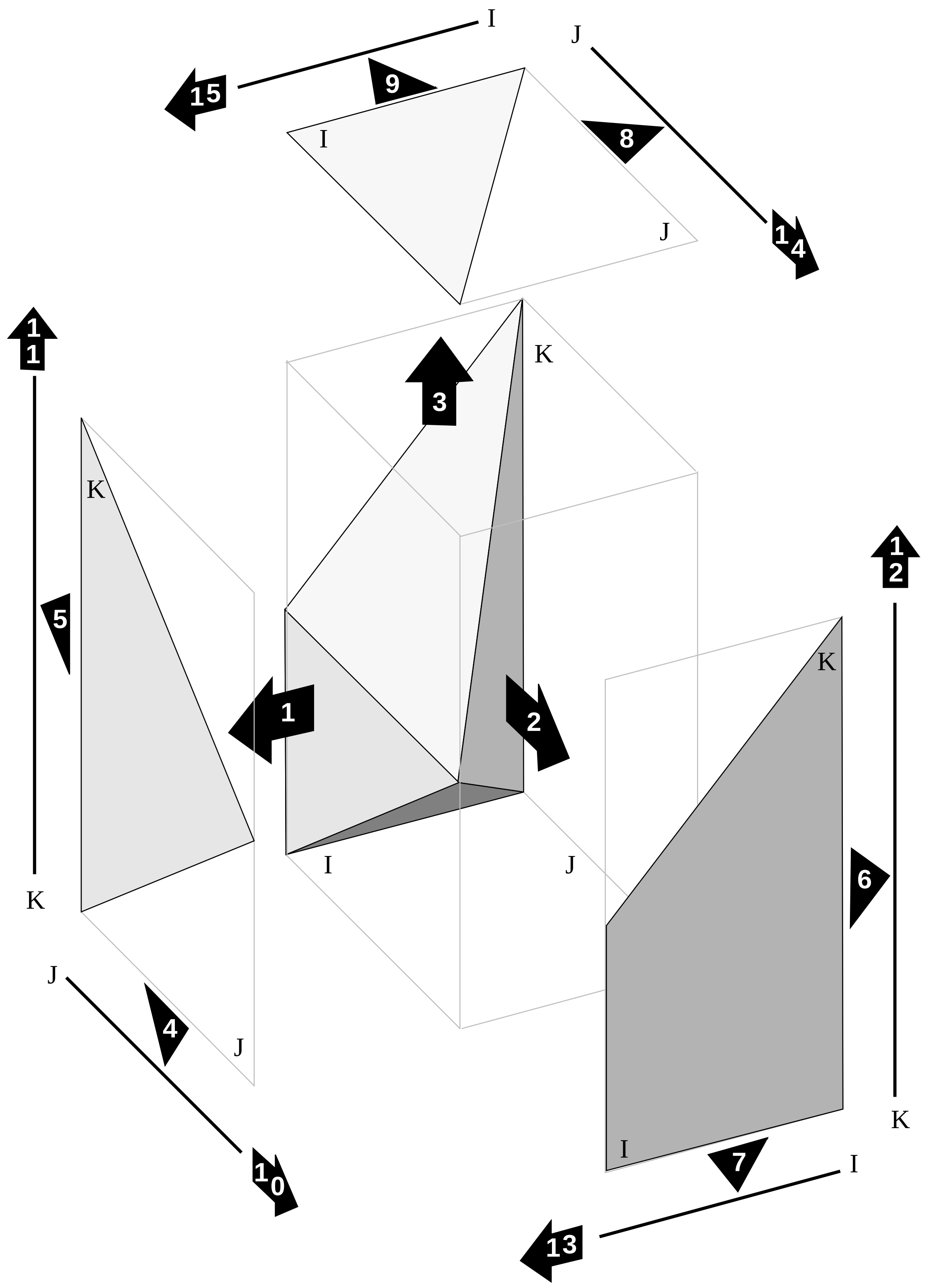


Figure 2-4. Projecting for all the possible scanning orders

Number	Third Projection	Second Projection	First Projection
15	`DO i = 1, N`		
9		`DO j = 1, i`	
3			`DO k = j, 2N-i`
13	`DO i = 1, N`		
7		`DO k = 1, 2N-i`	
2			`DO j = 1, min(k, i)`
14	`DO j = 1, N`		
8		`DO i = j, N`	
3			`DO k = j, 2N-i`
10	`DO j = 1, N`		
4		`DO k = j, 2N-j`	
1			`DO i = j, min(N, 2N-k)`
12	`DO k = 1, 2N-1`		
6		`DO i = 1, min(N, 2N-k)`	
2			`DO j = 1, min(k, i)`
11	`DO k = 1, 2N-1`		
5		`DO j = 1, min(k, 2N-k)`	
1			`DO i = j, min(N,2N-k)`

Figure 2-5. Loop nests generated by projecting the polyhedron

tions numbered $3 \rightarrow 9 \rightarrow 15$, and the bounds of the loop nest generated by our algorithm are identical to the original loop nest.

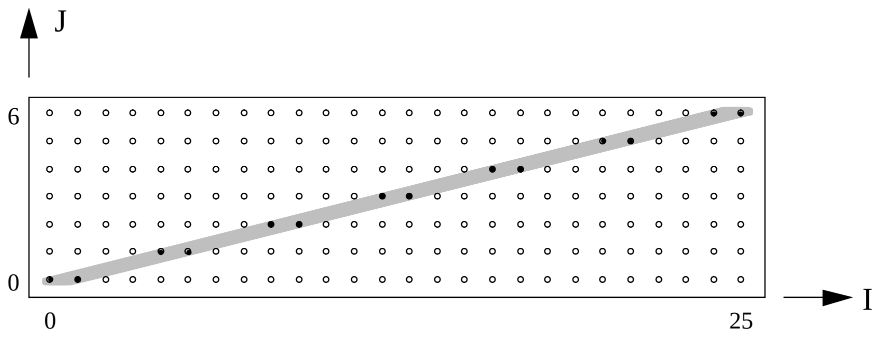
While the algorithm described above is correct, the generated code can be inefficient. Although the iteration space is dense, the presence of tight bounds may create gaps in the iterations. We demonstrate this using the example loop, in Figure2-6(a), with the iteration space given in Figure 2-6(b). The iteration space is graphically represented in Figure2-6(c), where the valid iterations are the dark dots in the shaded region. Although the dimen-

```
DO J = 0, 6
      DO I = 4*J, 4*J + 1
            .....
```

(a) Example loop nest

$$I^2(\ ) = \left\{ (I, J) \;\middle|\; \begin{array}{cc} J - 0 \geq 0 & 6 - J \geq 0 \\ I - 4J \geq 0 & 4J + 1 - I \geq 0 \end{array} \right\}$$

(b) Iteration Space



(c) Graphical representation of the iteration space

Figure 2-6. Example with tight bounds on the iteration space

sion I has iterations between 0 and 25, not all of them are valid. An inefficient loop nest is generated when we transpose the dimensions. In the new loop nest, given in Figure 2-7(a), the outer loop contains iterations that do not have any useful computation; they simply compute the bounds of the inner J loop just to find that the inner loop has no iterations.

```
DO I = 0, 25
      DO J = (2+I)/4, I/4
            .....
```

(a) Loop nest with empty iterations

```
DO II = 0, 25, 4
      DO I = II to MIN(II+1, 25)
            J = II/4
            .....
```

(b) Optimized loop nest

Figure 2-7. Transposed loop nests

This form of inefficiency can be eliminated as follows. We need not create a loop nest for v_n, when the bounds on v_n can be expressed as $v_k - \beta \le \alpha v_n \le v_k - \gamma$, where α, β and γ are integers such that $|\alpha| > 1$ and $0 \le \beta - \gamma < \alpha$, and v_k is an induction variable of an outer loop. Then, we can simply eliminate the loop for v_n from the loop nest by replacing all references to v_n by $\left\lfloor \frac{v_k - \gamma}{\alpha} \right\rfloor$ and replacing the loop v_k with:

```
DO $v_k' = \alpha\left\lfloor \frac{l - \gamma + \alpha - 1}{\alpha} \right\rfloor + \gamma$, $h$, $\alpha$
      DO $v_k = v_k'$, $min(v_k' + \beta - \gamma, h)$
```

where l and h are the lower and upper bounds of the original loop v_k, and v_k' is a new loop index. Furthermore, when $\beta = \gamma$, the loop v_k does not need strip-mining and can be replaced with:

$$\texttt{DO}\ v_k = \alpha\left\lfloor \frac{l-\gamma+\alpha-1}{\alpha} \right\rfloor + \gamma,\ \ h,\ \ \alpha$$

Note that the floor functions can be eliminated using integer division. The example loop nest in Figure 2-7(a) can be optimized, as given in Figure 2-7(b), since the bounds fit this definition with $\alpha = 4$, $\beta = 1$ and $\gamma = 0$.

2.2.3. Generating Efficient Loop Bounds

The Fourier-Motzkin elimination step used to generate the loop bounds produces a large number of redundant constraints. We iterate over all the constraints created by the Fourier-Motzkin elimination step, removing as many redundant constraints as possible. The order in which the constraints are checked for elimination determines the constraints that will be left at the end, which will constitute the bounds of the loop. We have developed a set of heuristics to simplify the system of inequalities, and to pick the order for eliminating the constraints so that the loop bounds generated are simple and efficient [9]. The outline of the algorithm is given in Figure 2-8. First, we simplify the system of inequalities. Then we attempt to eliminate the constraints in the given order. To check if a constraint is redundant, we replace the constraint in question with its negation. If the new system does not have an integer solution, then the constraint is redundant and can be eliminated.

2.2.3.1. Simplifying the inequalities

First, we simplify the inequalities by dividing all the coefficients by the greatest common divisor and finding the smallest integer offset. It is valid to round off the offset since we are interested only in integer solutions. The algorithm for normalizing the inequalities is given in Figure 2-9.

$EfficientBounds(S, (i_1, \ldots, i_n)) \to S'$
where S is a system of inequalities with index variables $(i_1, \ldots, i_n)$, from outer to inner loops respectively, and S' is a system of inequalities containing the loop bounds for i_n.

```
for each inequality $I \in S$ do
    $I = Simplify(I)$
$S = EliminateRedundant(S)$
$W = CalculateWeights\left(S, \{i_1, \ldots, i_n\}\right)$
for each inequality $I \in S$ in the order of weights $W$ do
    Remove inequality $I$ from $S$
        if $S \cap \{\neg I\} \neq \emptyset$ then
            $S = S \cap \{I\}$
for each inequality $I \in S$ do
    if the variable $i_n$ is not used in inequality $I$ then
        Remove inequality $I$ from $S$
return $S$
```

Figure 2-8. Algorithm for creating efficient loop bounds

2.2.3.2. Eliminating simple redundant inequalities

Next, we eliminate some of the redundant inequalities using a simple algorithm such that no two inequalities with identical coefficients exist in the system. Figure2-10 contains the algorithm.

$Simplify(I) \rightarrow I'$
where I, I' are inequalities such that $I = \{a_0 + a_1 i_1 + \ldots + a_k i_k \geq 0\}$ and $a_0, \ldots, a_n$ are integers

```
$g = gcd(a_1, \ldots, a_k)$
**return** $\left\lfloor \frac{a_0}{g} \right\rfloor + \frac{a_1}{g} i_1 + \ldots + \frac{a_k}{g} i_k \geq 0$
```

Figure 2-9. Algorithm for simplifying an inequality

$EliminateRedundant(S) \rightarrow S'$
where S and S' are systems of inequalities

```
**for** all pairs of inequalities $\{c_1 + a_1 i_1 + \ldots + a_k i_k \geq 0\} \in S$ and
$\{c_2 + a_1 i_1 + \ldots + a_k i_k \geq 0\} \in S$ where $c_1, c_2, a_1, \ldots, a_n$ are
integer constants **do**
    **if** $c_1 \geq c_2$ **then**
        Remove $c_1 + a_1 i_1 + \ldots + a_k i_k \geq 0$ from $S$
    **else**
        Remove $c_2 + a_1 i_1 + \ldots + a_k i_k \geq 0$ from $S$
**return** $S$
```

Figure 2-10. Algorithm for eliminating simple redundant inequalities

2.2.3.3. Determining the elimination order

Finally, we order the elimination of redundant inequalities, such that complex inequalities that can generate expensive loop bounds are eliminated before the inequalities that generate efficient loop bounds. The elimination order is determined by weights generated in the algorithm in Figure 2-11. The inequalities with higher weights will become candidates for elimination before the inequalities with lower weights.

$CalculateWeights(S, (i_1, \ldots, i_n)) \to W$
where S is a system of inequalities with index variables $(i_1, \ldots, i_n)$ from outer to inner loops respectively, and W is the set of weights for the elimination ordering

```
for all inequalities $\{a_o + a_1v_1 + \ldots + a_kk_k + b_1i_1 + \ldots + b_ni_n \geq 0\} \in S$
where $a_0, \ldots, a_k$ and $b_1, \ldots, b_n$ are integer constants and
$v_1, \ldots, v_k$ are loop invariant variables do
    $c = n - 1$
    while $c > 1$ and $b_c = 0$ do
        $c = c - 1$
    if there exist $\{-a_o - a_1v_1 - \ldots - a_kk_k - b_1i_1 - \ldots - b_ni_n \geq 0\} \in S$ then
        $c = c - 2n$
    if $|b_n| \neq 1$ then
        $c = c + n$
    $W\left(\{a_o + a_1v_1 + \ldots + a_kk_k + b_1i_1 + \ldots + b_ni_n \geq 0\}\right) = c$
Return $W$
```

Figure 2-11. Algorithm for calculating the weights that order the elimination of redundant inequalities

---

The least expensive loop bounds are the pairs of inequalities that form an equality, because the loop can be replaced by a single assignment statement. The inequalities that create loop bounds with floor and ceiling calculations are the most expensive and are therefore assigned highest weights. Otherwise, bound expressions with only outer or no loop index variables receive higher weights since loop invariant bound expressions can be moved out of the inner loops.

2.3. Linear Inequalities with Symbolic Coefficients

We have extended Fourier-Motzkin elimination to handle simple non-linear systems [9]. The variables of the linear inequalities can have a restricted form of symbolic coefficients.

Definition 2-4: *A linear inequality with symbolic coefficients is of the form*

$$\pm a_o^o \pm a_1^o u_1 \dots \pm a_m^o u_m \pm \left(a_o^1 + a_1^1 u_1 \dots + a_m^1 u_m \right) v_1 \dots \pm \left(a_o^n + a_1^n u_1 \dots + a_m^n u_m \right) v_n \geq 0$$

where $v_1, \dots, v_n$ *are integer variables,* $u_1, \dots, u_m > 0$ *are symbolic constants and* $\forall x, y \quad (0 \leq x \leq n)\ and\ (0 \leq y \leq m)\ ,\ a_y^x \geq 0$ *are integers.*

The scope of the systems that are allowed is limited to cases where the result of the Fourier-Motzkin elimination also creates inequalities that conform to Definition 2-4.

Theorem 2-1: *For the two inequalities with symbolic coefficients*

$$\pm a_o^o \pm a_1^o u_1 \dots \pm a_m^o u_m \pm \left(a_o^1 + a_1^1 u_1 \dots + a_m^1 u_m \right) v_1 \dots - \left(a_o^n + a_1^n u_1 \dots + a_m^n u_m \right) v_n \geq 0 \quad and$$
$$\pm b_o^o \pm b_1^o u_1 \dots \pm b_m^o u_m \pm \left(b_o^1 + b_1^1 u_1 \dots + b_m^1 u_m \right) v_1 \dots + \left(b_o^n + b_1^n u_1 \dots + b_m^n u_m \right) v_n \geq 0 \quad ,$$

where $v_1, \dots, v_n$ *are integer variables,* $u_1, \dots, u_m > 0$ *are symbolic constants and* $\forall x, y \quad (0 \leq x \leq n)\ and\ (0 \leq y \leq m)\ ,\ a_y^x, b_y^x \geq 0$ *are integers, the Fourier-Motzkin elimination step to eliminate the variable* i_n *creates another inequality conforming to Definition 2-4 iff either*

i) *There exist integers p and q such that* $\forall y \quad (0 \leq y \leq m)\ ,\ p a_y^n = q b_y^n$ *or*

ii) *(* $\forall y \quad (1 \leq y \leq m)\ ,\ a_y^n = 0$ *or*

$\forall x, y \quad (0 \leq x \leq n-1)\ and\ (1 \leq y \leq m)\ ,\ b_y^x = 0$ *) and*

$$(\forall y \quad (1 \le y \le m) \, , \, b_y^n = 0 \; or$$
$$\forall x, y \quad (0 \le x \le n-1) \; and \; (1 \le y \le m) \, , \, a_y^x = 0 \;)$$

This class of systems is important because it enables the compiler to handle symbolic block sizes in Single Program Multiple Data(SPMD) code generation as shown below. Otherwise, the number of iterations attached to each processors has to be determined at compile time.

2.3.1. SPMD Code Generation Example

We illustrate the use of linear inequalities with symbolic coefficients by an example, where we generate an SPMD loop nest after parallelization. The loop nest, given in Figure 2-12, has the inner loop J marked parallel. We need to generate an SPMD loop nest to execute

```
DO I = 0, U
     DOALL J = 0, MIN(2*I, V)
          .....
```

Figure 2-12. Example DOALL loop nest

this loop in parallel. The Figure 2-13 shows the iteration space of the loop nest. The shaded area represents the iterations that need to be executed. Iterations of the J loop are distributed across processors such that each processor is assigned an equal size block of iterations. We need to create a system of inequalities to represent the iteration space. Using this iteration space, we can generate a single program that will assign blocks of iterations of the loop J and execute the correct iterations in the corresponding processor.However, neither the number of processors nor the number of iterations of the loop J is known at compile-time. Therefore, we cannot create a linear system with integer coefficients to represent the iteration space. However, a system of inequalities with symbolic coefficients can be cre-

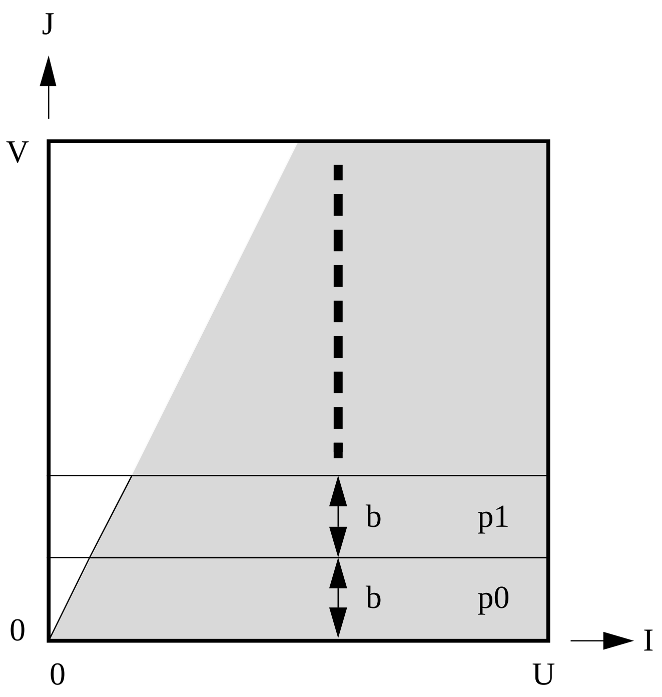


Figure 2-13. Iteration space

ated, as given by Figure 2-14, to represent the iteration space. The first five inequalities are the loop bounds of the input program. The last two inequalities distribute b iterations of the J loop across the processors. The processor identification number is the variable x. Now, applying Fourier-Motzkin elimination with the scanning order (I, J), we generate the SPMD loop nest given in Figure 2-15. Using the number of processors, P, which is a run-time constant, we generate code to calculate the appropriate block size at run-time.

2.4. Linear Inequality Representation for Data and Processor Spaces

The two other important multi-dimensional integer spaces used in our compiler are the array data space A and the processor space P.

$$I^2(U, V, b, x) = \left\{ (I, J) \;\middle|\; \begin{array}{cc} I \ge 0 & U - I \ge 0 \\ J \ge 0 & 2I - J \ge 0 \\ & V - J \ge 0 \\ J - bx \ge 0 & bx + b - 1 - J \ge 0 \end{array} \right\}$$

Figure 2-14. System of inequalities describing the iteration space

```
P = NumProcs()
x = MyProcID()
b = (min(V, 2*U)+P-1)/P
DO I = max((1+b*x)/2, 0), U
      DO J = max(b*x, 0), min(2*I, V, -1+b+b*x)
            .....
```

Figure 2-15. Compiler generated SPMD loop nest

2.4.1. Data Space

Manipulating arrays is critical for analysis and optimizations when compiling dense matrix scientific applications. Accesses to multi-dimensional arrays can be represented using systems of linear inequalities, providing a convenient framework for array analysis and optimization. For example, the array summary representation defined in Chapter 5 is based on systems of linear inequalities.

Definition 2-5: *The index set of an m-dimensional data array, with the declaration* $(l_1 : u_1, \ldots, l_m : u_m)$ *, is given by*

$$A = \left\{ (a_1, \ldots, a_m) \in \mathsf{A} \;\middle|\; \forall_{k = 1, \ldots, m} \;\; l_k \le a_k \le u_k \right\}.$$

2.4.2. Processor Space

When generating code for multiprocessors as well as when performing communication optimizations, compilers need to operate on the processor space. Again, it is convenient to represent the processor space as a system of linear inequalities. The SPMD code generation example in Section 2.3.1 introduced a simple one-dimensional processor space. Chapter 9 uses virtual and physical processor spaces extensively in communication code generation and communication optimization algorithms.

Definition 2-6: *For a q-dimensional processor space, where* $r_1, \ldots, r_q$ *are the number of processors in each dimension, the index set of the processor space is*

$$P = \left\{ (p_1, \ldots, p_q) \in \mathsf{P} \;\middle|\; \forall_{k = 1, \ldots, q} \;\; 0 \le p_k < r_k \right\}.$$

2.5. Related Work

Researchers have used integer and linear programming techniques to solve many individual problems in parallelizing compilers. For example, compiler problems such as exact data-dependence analysis [110,119], array analysis based on array summary information [137], instruction scheduling for superscalars [5], automatic data layout for minimizing communication [21], and code generation after loop transformations [10,11] have been solved using linear and integer programming. In this thesis, we introduce a framework based on linear inequalities that is used formany purposes, such as representing array summaries in interprocedural data-flow analysis, solving for the array reshapes, identifying modulo and division optimizations, and generating and optimizing communication code for distributed address-space machines.

Ancourt and Irigoin used a series of projections to generate loop nests after loop transformation [10,11]. We have introduced a set of heuristics to simplify the loop bounds generated by their algorithm. We have also extended their algorithm to handle simple non-linear systems.

2.6. Chapter Summary

In this chapter, we describe the linear inequalities framework used for developing advanced compiler analyses and optimizations. The framework is used in compiler algorithms to represent and manipulate iteration, array and processor spaces. We introduce the use of this framework by describing an algorithm for code generation. We have extended the linear inequalities framework to handle simple non-linear systems with symbolic coefficients.

3 Coarse-Grain Parallelism

Shared-memory multiprocessors are now a widely available class of computationally powerful machines. As hardware technology advances make pervasive parallel computing a possibility, it is ever more important that tools be developed to simplify parallel programming. Parallelizing compilers that automatically parallelize sequential applications are critical tools, because they free programmers from the difficult task of explicitly managing parallelism. A large body of research and development effort has focused on developing parallelizing compilers for scientific applications. In Section 3.1 we focus on the major issues involved in detecting parallelism in sequential scientific applications.

Current parallelizing compilers have not succeeded in obtaining good parallel performance on symmetric shared-memory multiprocessors. These parallelizers, based on vectorization technology, are generally capable of finding only inner loop parallelism. The inability to parallelize computation that occurs outside the inner loops reduces the effectiveness of these compilers. Furthermore, multiprocessors need to perform an expensive synchronization operation after executing each parallel region. Parallelizing the inner loops creates parallel regions with relatively small amounts of computation; thus the cost of synchronization can easily overwhelm the benefits of the parallel execution. For parallelizing compilers to target multiprocessors effectively, it is necessary to locate *coarse-grain parallelism*; that is, to find outer loops with independent computations that can perform a significant amount of work without any synchronization.

This chapter provides an overview of parallelizing sequential scientific applications for shared-memory multiprocessors. In Section 3.2 we will introduce coarse-grain parallelism. We focus on the advanced array analyses required for obtaining coarse-grain parallelism in Section 3.3.

3.1. Parallelism in Sequential Scientific Programs

Scientific applications are typically computationally intensive, and thus can benefit immensely from parallelization. The domain of applications we are interested is dense matrix scientific applications written in FORTRAN [150]. Within this domain, loop nests dominate the computation and multi-dimensional arrays hold most of the data structures. A parallelizing compiler determines loops that can be parallelized by analyzing accesses to scalar and array variables within the loops. One of the most difficult parts of this parallelization process is array analysis. For the references to each array data structure, array analysis determines if executing the iterations of a loop in parallel does not violate the semantics of the original serial ordering. Current parallelizers accomplish this using *data dependence analysis*.

3.1.1. Data Dependence Analysis

Current parallelizing compilers use data dependence analysis to check whether the parallel execution of a loop violates serial ordering constraints between any write operation and any other write or read operation to the same memory location [151]. Data dependence analysis is performed on each pair of references to the same array, where two references are said to be *dependent* if any of the locations accessed by one reference is also accessed by the other [149]. A dependence is said to be *loop-carried* by a loop if the dependence occurs between two iterations for the same instance of the loop. Thus, according to the data dependence test, a loop can be parallelized if there are no *loop-carried data dependences.*

Definition 3-1: *For an m-deep loop nest with two array accesses X, X' to the same array in the loop body, if there exist iterations $(i_1, \ldots, i_m)$ and $(i'_1, \ldots, i'_m)$ such that $\forall_{j = 1 \ldots k-1}\ i_j = i'_j$ and $i_k < i'_k$, and array access a at iteration $(i_1, \ldots, i_m)$ accesses the same memory location as a' at $(i'_1, \ldots, i'_m)$, and*

i) X is a write access and X' is a read access, then there exists a true-dependence carried at the k-th loop.

ii) X is a read access and X' is a write access, then there exists an anti-dependence carried at the k-th loop.

iii) both X and X' are write accesses, then there exists an output-dependence carried at the k-th loop.

The compilers perform a data dependence test on each pair of accesses within the candidate loop for parallelization. Determining the data dependences of loop nests where the loop bounds and array indices are affine functions of the loop indices is equivalent to integer programming [110]. Many practical algorithms have been devised to find the data dependence information exactly [19,110,120,149,151].

3.1.2. Extracting Fine-Grain Parallelism

The first-generation parallelizers start the parallelization process by performing a series of symbolic analyses such as constant propagation, induction variable identification and loop-invariant code motion. These analyses increase the ability of finding parallel loops. Next, each loop nest is analyzed to identify parallelizable loops. These analyses are performed intraprocedurally; thus a procedure call within the loop will eliminate it as a candidate for parallel execution. Since the presence of any scalar definition within the loop creates a loop-carried dependence, the parallelizer attempts to eliminate this dependence by applying *scalar privatization* or *scalar reduction* [149]. When the scalar value used in each iteration is created within the same iteration, the loop-carried data dependence can be eliminated by giving each processor a private copy of the variable. Further, a reduction (*e.g.,* computation of a commutative and associative operation such as sum, product, or maximum of the scalar) can be parallelized by having each processor compute a partial reduction locally and update the global result at the end. Finally, the parallelizer performs data dependence analysis on all pairs of accesses to the same array within the loop. After these analyses and optimizations, if the compiler does not find any loop-carried array dependences and can eliminate all the loop-carried scalar dependences, then the loop can be parallelized.

3.2. Coarse-Grain Parallelism

The first generation of parallelizing compilers targeted for vector supercomputers focused on finding inner-most parallel loops with only a few simple operations that can be converted into vector operations [4,32,125].

Multiprocessors are more powerful than vector machines in that they can execute different threads of control simultaneously. The processors can execute different segments of code in parallel, each of which can be arbitrarily complex.

The current parallelizing compilers, developed from vectorizing compiler technology, do not effectively obtain good performance on multiprocessors [23,131]. Parallelizing just inner loops is not adequate for multiprocessors for two reasons. First, inner loops may not make up a significant portion of the computation, thus limiting the parallel speedup by limiting the amount of parallelism. Second, multiprocessors need to perform an expensive synchronization operation at the end of each parallel region. When parallelizing inner loops, which normally contain only small amounts of computation, the cost of frequent synchronization and load imbalance can potentially overwhelm the benefits of parallelization. Thus, for a parallelizing compiler to target a multiprocessor effectively, it must identify outer parallelizable loops to extract coarse-grain parallelism.

However, detecting coarse-grain parallelism is much more complicated than finding inner loop parallelism, requiring whole program analyses and aggressive optimizations. Interprocedural analysis is necessary for obtaining coarse-grain parallelism. If programs are written in a modular style, it is natural that coarse-grain parallel loops will span multiple procedures. For this reason, procedure boundaries must not pose a barrier to analysis [23]. This can be accomplished by applying data-flow analysis techniques across procedure boundaries using an interprocedural framework. Although many interprocedural analyses for parallelization have been proposed [70,80,81,86,108], they have rarely been adopted in practice. The primary obstacle to progress in this area has been the fact that effective interprocedural compilers are substantially harder to build than their intraprocedural counterparts. Moreover, there is an inherent trade-off between performing analysis efficiently and obtaining precise results. A successful interprocedural compiler must handle the complexity of the compilation process, while maintaining reasonable efficiency without sacrificing too much precision.

It is also critical to go beyond the restrictive data-dependence test in analyzing arrays when detecting coarse-grain parallelism. We need to perform parallelism-enhancing optimizations on arrays, such as array privatization, to expose the inherent parallelism in the algorithms of the program.

3.3. Advanced Array Analyses

Much of the inherent parallelism that is available in an abstract algorithm can be hidden by the implementation of the algorithm. We can expose some of these inherent parallelism by using complex analyses such as array data-flow analysis and by performing aggressive optimizations such as array privatization and array reductions.

3.3.1. Beyond Location-Based Dependences

Traditional data dependence analysis checks if two iterations of the loop access the same *memory location*. However, the only dependence that requires data to be communicated between iterations of a loop is a *flow-dependence,* where a *data value* used by an iteration is defined in a previous iteration.

Definition 3-2: *For an m-deep loop nest with a write access* X *and a read access* X' *to the same array, there exists a loop-carried flow-dependence at the k-th loop iff there exist iterations* $(i_1, \ldots, i_m)$ *and* $(i'_1, \ldots, i'_m)$ *such that* $\forall_{j = 1 \ldots k-1}\ i_j = i'_j$ *and* $i_k < i'_k$*, and read* X' *at iteration* $(i'_1, \ldots, i'_m)$ *uses the data written by* X *at iteration* $(i_1, \ldots, i_m)$.

Thus, when there are no value-based flow-dependences between a write and a read access, a loop can be parallelized even if there exist location-based dependences. However, this requires us to assign a private copy of the array to each processor.

Array privatization is crucial for parallelizing ordinary scientific applications because programmers tend to reuse the same array space for multiple purposes. This creates memory-based dependences while there are no value-based dependences requiring sequential execution. A simple example motivating the need for value-based dependences is shown in Figure 3-1. It is a 160-line loop taken from the Nas sample benchmark appbt. Figure 3-2 shows both location-based and value-based dependences for the read accesses to the array TM in the 4-th iteration of the outermost loop. All iterations of the outermost loop write to the same *locations* of the array TM that are read in the 4-th iteration. Thus, as shown in

```
DO K = 2, NZ-1
     DO M = 1, 5
          DO N = 1, 5
               TM(1:5,M) = ...
     DO M = 1, 5
          DO N = 1, 5
               ... = TM(N,M)
```

Figure 3-1. Example from appbt

Figure 3-2(a), the location-based data dependence analysis will find a loop-carried true-dependence at the outermost loop, suppressing parallelization of the loop. However, the data *values* read in the 4-th iteration are defined by the write instructions of the same iteration, as shown in Figure 3-2(b). Consequently, we can parallelize the outermost loop by allocating a private copy of TM to each processor.

Finding privatizable arrays can be achieved by *array data-flow analysis*, which extends scalar data-flow analysis to individual array elements [28,53,55,111,112,121,122]. Using array data-flow analysis, if we can determine that an array in a loop, with loop-carried true-, anti- or output-dependences, does not contain any loop-carried flow-dependences, we can still parallelize the loop by allocating a private copy of the array to each processor. When privatizing an array, we need to perform *initialization* at the beginning of the parallel region and *finalization* at the end of the parallel region. We need to initialize the array by copying all values, that are used within the loop but are defined outside, from the original array to each private copy. We finalize the array by copying those values that are defined within the loop from the private copies to the original array.

An example of array privatization with initialization is shown in Figure 3-3. The figure shows a portion of a 1002-line interprocedural loop in the Perfect benchmark spec77.

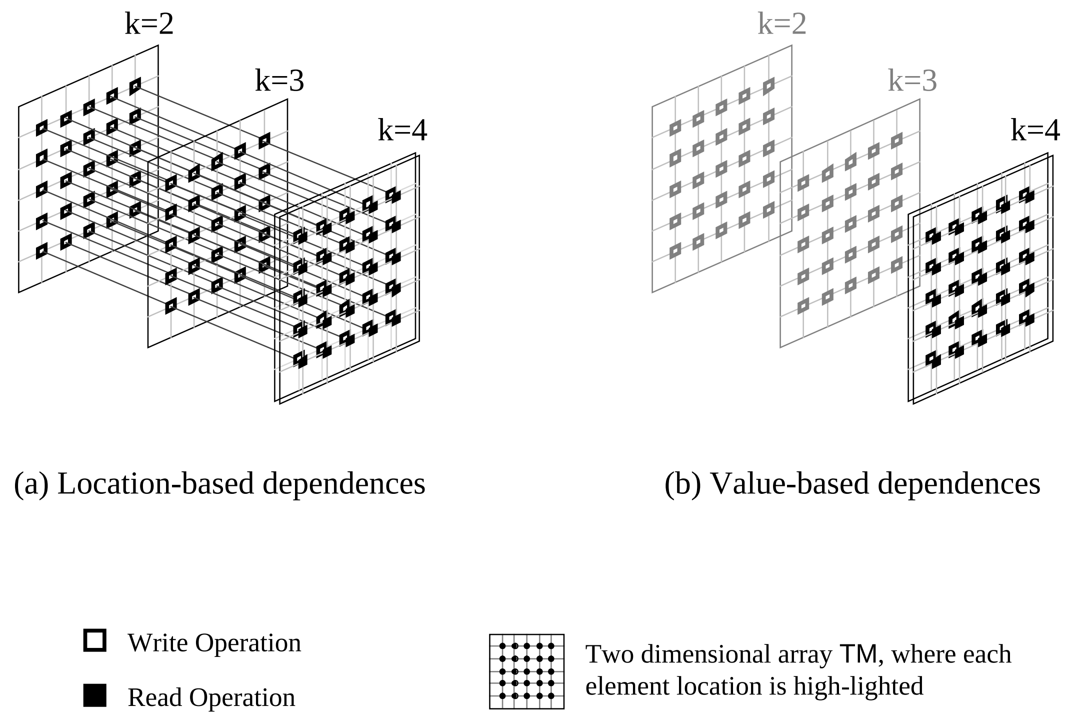

k=2
k=3
k=4
(a) Location-based dependences
k=2
k=3
k=4
(b) Value-based dependences
□ Write Operation
■ Read Operation
Two dimensional array TM, where each element location is high-lighted

Figure 3-2. Dependences for the elements read by the 4-th iteration of the `k` loop

Here, part of array `ZE`, the second row, is modified before it is referenced; the remainder of the array is not modified at all in the loop. Array `ZE` is privatizable in the outer loop by giving each processor a private copy with all but the second row initialized with the original values.

3.3.2. Multiple Array Accesses

To locate coarse-grain parallelism successfully, we must analyze very large interprocedural loops with numerous array accesses. When numerous reads and writes to an array are interleaved in multiple loop nests, it is more difficult to keep track of the order of the elements accessed. Figure 3-4, which is extracted from `spec77`, illustrates the complexity of the

```
DO LAT = 1, 38
     DO K=1, 12
          ZE(2,K) = RELVOR(K)
          // UVGLOB reads the entire
          // Kth column of array ZE
          CALL UVGLOB(...,ZE(1,K),...)
          ZE(2,K) = ABSVOR(K)
```

Figure 3-3. Example of an array privatization from spec77

```
DO LAT = 1, 38
     W(1:2,1:UB)   = ...
     W(3:36,1:UB)  = ...
     W(62:96,1:UB) = ...
     W(37:48,1:UB) = ...
     W(51:61,1:UB) = ...
     W(49:50,1:UB) = ...
     ...= W(1:2,1:UB)+
          W(33:34,1:UB)+
          W(65:66,1:UB)
     ... =W(3:32,1:UB)+
          W(35:64,1:UB)+
          W(67:96,1:UB)
```

Figure 3-4. Example of multiple regions across loops from spec77

problem. Each statement in array notation here corresponds to a doubly nested loop. In order to determine that the array W is privatizable, we need to infer from the collection of write operations that W defines all the elements before they are read.

3.3.3. Array Reshapes Across Procedure Boundaries

The existence of array reshapes further complicates interprocedural analysis. An example of an array reshape is given in Figure 3-5. The code segment is a part of an interprocedural loop from the TURB3d program of the SPEC95fp benchmark suite. In this segment, a three-dimensional array U in the caller is treated as a vector in DCFT. The FORTRAN-77 standard allows array reshapes with equivalence statements, in parameter passing, and with different common block definitions [150]. To perform the aggressive whole program anal-

```
DIMENSION U(66,64,64)
...
DO K=1,64
      CALL DCFT(U(1,1,K),33)
...
DO J=1,64
      CALL DCFT(U(1,J,1),33*64)
...

SUBROUTINE DCFT(X, INCX)
REAL*8 X(*)
DO I=1,33
      DO II=1,64
            ... = X((I-1)*2+(II-1)*2*INCX+1)
            ... = X((I-1)*2+(II-1)*2*INCX+2)
...
```

Figure 3-5. An example with two array reshapes from turb3d

ysis required in finding coarse-grain parallelism, it is necessary for the compiler to analyze the programs in the presence of these features and determine their effect on the rest of the analysis.

3.4. Chapter Summary

It is necessary to locate coarse-grain parallelism for compilers to target multiprocessors effectively. However, obtaining coarse-grain parallelism requires many advanced analyses and optimizations such as interprocedural analysis, array privatization and array reshape analysis.

4 Interprocedural Array Analysis

Automatic detection of coarse-grain parallelism is challenging as it requires a large suite of robust analysis techniques to work together. Furthermore, the original program may need to be transformed before it can be parallelized. Detecting and enabling parallelism require sophisticated analyses on array and scalar variables. These analysis techniques need to operate across procedural boundaries seamlessly. For a parallelizing compiler to work in practice it must not only be sufficiently powerful, but also robust and efficient.

Most of these analyses are formulated as interprocedural data-flow problems. As solving interprocedural data-flow problems is very complex, we have developed a framework, described in Section 4.1, to cope with the complexity involved in building such a system.

The parallelization process is composed of multiple phases. The first set of phases consists of a large suite of interprocedural symbolic analyses on scalar variables. These analyses include constant propagation, common sub-expression recognition, loop invariant code motion, and induction variable detection. These symbolic analyses provide detailed and accurate information about the input program, which enhances the effectiveness of array analyses and parallelization. The next set of phases includes parallelization analyses on scalar variables. These analyses identify scalar dependences, and perform optimizations, such as scalar privatization and scalar reductions. More information on scalar analyses can be found in [74]. The scalar analyses are followed by the array analyses and parallelization, which include the following four phases:

- The first phase propagates loop context information to the nested loops.
- The second phase performs the array data-flow analyses necessary for dependence testing and parallelization.

- The third phase performs data-dependence and privatization tests to determine which loops can be executed in parallel.

- The last phase identifies the outermost parallel loops and transforms the code to execute them in parallel.

The outline of this chapter is as follows. We introduce the interprocedural framework in Section 4.1 The four phases of array analysis are described in Sections 4.2, 4.3, 4.4 and 4.5 respectively. We compare our approach to related works in Section4.6, and we summarize in Section 4.7.

4.1. Interprocedural Framework

Traditional intraprocedural data-flow analysis frameworks help reduce development time and improve correctness by capturing the common features in a single module [90]. In an interprocedural setting, a framework is even more important because of the increased complexity in collecting and managing information about all the procedures in a program. Thus, when building the parallelizer, we rely on an interprocedural framework that encapsulates the common features of interprocedural analysis [74,76].

Traditional intraprocedural data-flow frameworks are *flow-sensitive*. That is, they derive data-flow results along each possible control flow path through the procedure. Straightforward interprocedural adaptation of flow-sensitive intraprocedural analysis is not sufficient to maintain the same precision over the entire program. For example, interprocedural analysis using the *supergraph* [117] program representation, where individual control flow graphs for the procedures in the program are linked together at procedure call and return points, loses context sensitivity by propagating information along *unrealizable paths* [104]. This occurs when the analysis propagates calling context information from one caller through a procedure and returns the side-effect information to a different caller. Furthermore, iterative analysis over this structure is slow because of the large number of control flow paths through which information flows.

Full interprocedural precision was previously obtained either by inline substitution or by tagging data-flow values with a path history through the call graph [79,117,128,130].

However, these methods do not exploit the common case in which many calls to a procedure have the same context. Thus, they require excessive space and are expensive and impractical. Inlining the procedure bodies at all the call sites in the program can result in code explosion, and tagging of the data-flow values with all possible paths can result in rapid multiplication of the tags. Our interprocedural framework utilizes path-specific information only when it can provide opportunities for improved optimization. The system incorporates *selective procedure cloning*, a program restructuring technique in which the compiler replicates the analysis results in the context of distinct calling environments [44]. By applying cloning selectively according to the unique data-flow information it exposes, the interprocedural system can obtain the same precision as full inlining without unnecessary replication.

4.1.1. The Regions Graph

Region-based analysis collects information at the boundaries of program regions: basic blocks, loop bodies and loops (restricted to DO loops), procedure calls, procedure bodies, and procedures. The interprocedural framework represents a program as a set of*regions*, one for each loop body and procedure body in the program. Within a region, inner loop nests are represented by “loop” nodes, procedure call sites by “procedure” nodes, and the remaining basic blocks in the region by “basic block” nodes. These nodes and their control flow edges necessarily define a directed acyclic graph. The regions have a single entry and a single exit defined by the “start” node and the “end” node, respectively. Therefore, a region R is a four-tuple (N, E, s, e) where N is the set of nodes, E is the set of edges, s is the start node and e is the end node. Associated with each “loop” or “procedure” node is the region representing the corresponding loop body or procedure body, respectively. We say that a region R is an *immediate subregion* of another region Q if R represents the body of a node in region Q. A program’s regions and their immediate subregion relationships define a *regions graph* of the program. The regions graph of any FORTRAN-77 program, which by definition does not contain recursive function calls, is also a directed acyclic graph.

The regions graph of the example program segment in Figure4-1 is shown in Figure 4-2. In Figure 4-2, the regions are enclosed in gray polygons and the immediate subregion relationship is represented by gray dotted lines. For each region, the start and end nodes are represented by black ovals, the loop and procedure nodes by gray ovals and basic block nodes by white ovals. Control flow between the nodes within each region is represented by arrows. Each node is annotated with the corresponding code.

```
     ...
     IF X > 0 THEN
          DO I = 2, X
               CALL FOO(I,A)
               IF A(I) > 0 THEN
                    Y = Y + A(I)
               ELSE
                    Y = Y - A(I)
     ELSE
          DO J= 2, -X
               CALL FOO(J,A)
          DO K= 2, -X
               CALL FOO(K,A)

SUBROUTINE FOO(I,B)
     B(I) = B(I-1) + B(I) + B(I+1)
     ...
```

Figure 4-1. Example program

4.1.2. Data-Flow Analysis

Data-flow analysis is composed of one or more traversals through the regions graph where each traversal propagates the flow values in a single sweep over the nodes of the graph. We can choose the order in which to visit the regions and the nodes within each region inde-

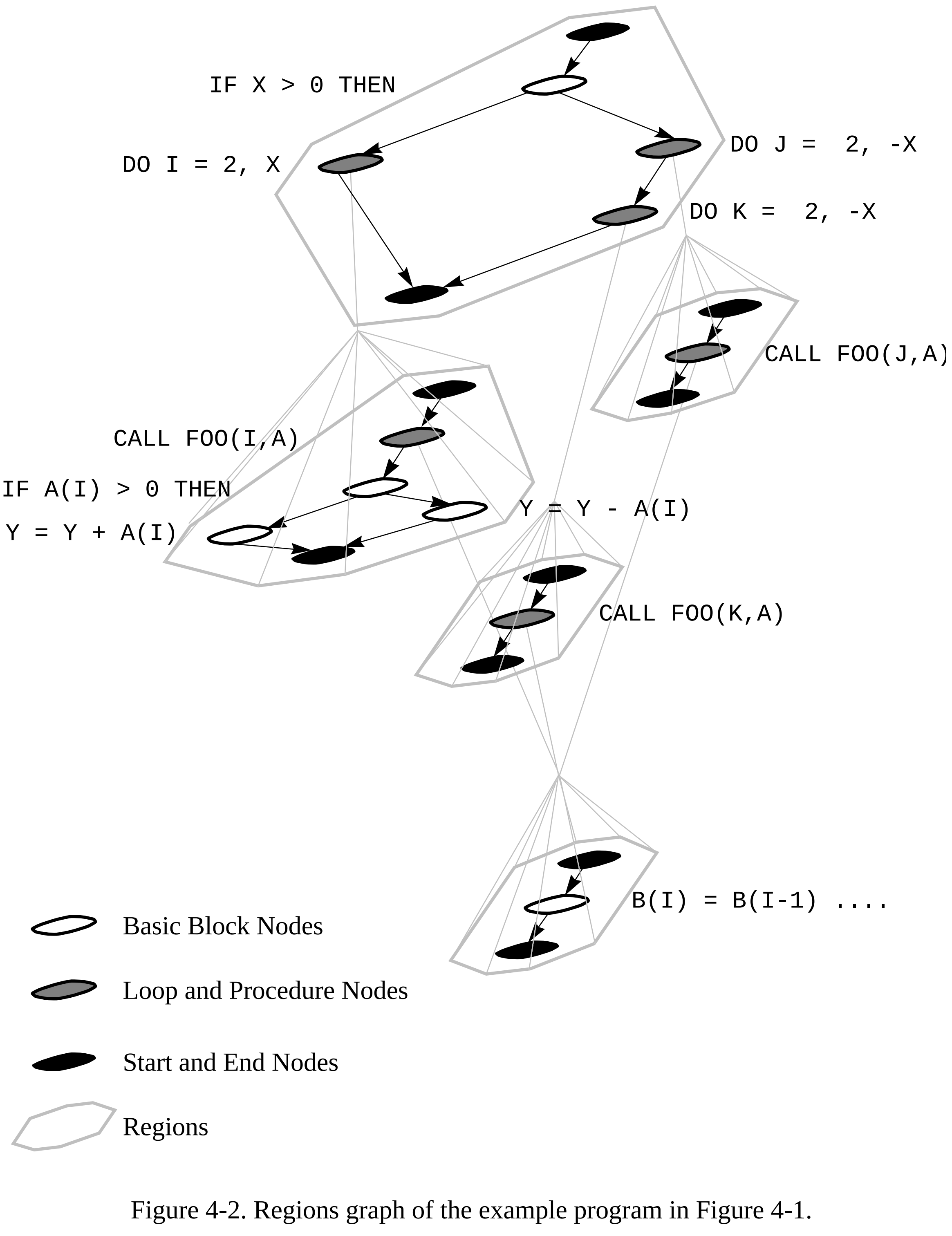


Figure 4-2. Regions graph of the example program in Figure 4-1.

pendently. The regions can be visited either in a *top-down* or a *bottom-up* order. In the top-down order, we visit a region before its immediate subregions, and vice versa in a bottom-up traversal. Each region is a directed acyclic graph and the nodes in the region can be visited in a *forward-flow* or *backward-flow* order. In a forward-flow order, we visit a node before its successors in the control-flow graph, and vice versa in a backward-flow order. In addition to these two flow-sensitive methods of propagation, nodes can be visited in a *flow-insensitive* method which ignores the control flow within the region and treats all the nodes as a single summary node. Next, we define bottom-up traversal for forward-flow and backward-flow order, and top-down traversal for flow-insensitive order.

4.1.2.1. Bottom-up traversal for forward and backward flow order

In a bottom-up traversal of a regions graph, we analyze the program starting from the leaf procedures in the call graph. Each procedure is analyzed once, after all its callee procedures have been analyzed. Within each procedure, analysis is performed from inner loops to outer loops. In the regions graph representation of the program, this is achieved by visiting the regions in the directed acyclic graph in a post-order traversal. Thus, each node is visited only once in a bottom-up traversal pass. Figure4-3 shows the bottom-up, forward-flow propagation for the regions graph example in Figure4-2. The ordering of the traversal is given by the arrows in the diagram.

In a forward-flow pass, the analysis calculates the *flow value* at each node, summarizing the cumulative effect of traversing through all the possible paths between two points of the program represented by the start node of the region and the current node. In a backward-flow pass, the flow value of a node summarizes the cumulative effect of traversing, in reverse direction, through all the possible paths between the program points represented by the current node and the end node of the region. We calculate the *local values* as an intermediate step in calculating the flow values. The local value of a node summarizes the cumulative effect of traversing through all the paths of the program segment represented by the node.

The algorithm for bottom-up traversal is given in Figure4-4. We start by calculating the local values for each node. We define the function Loc that provides the local value for

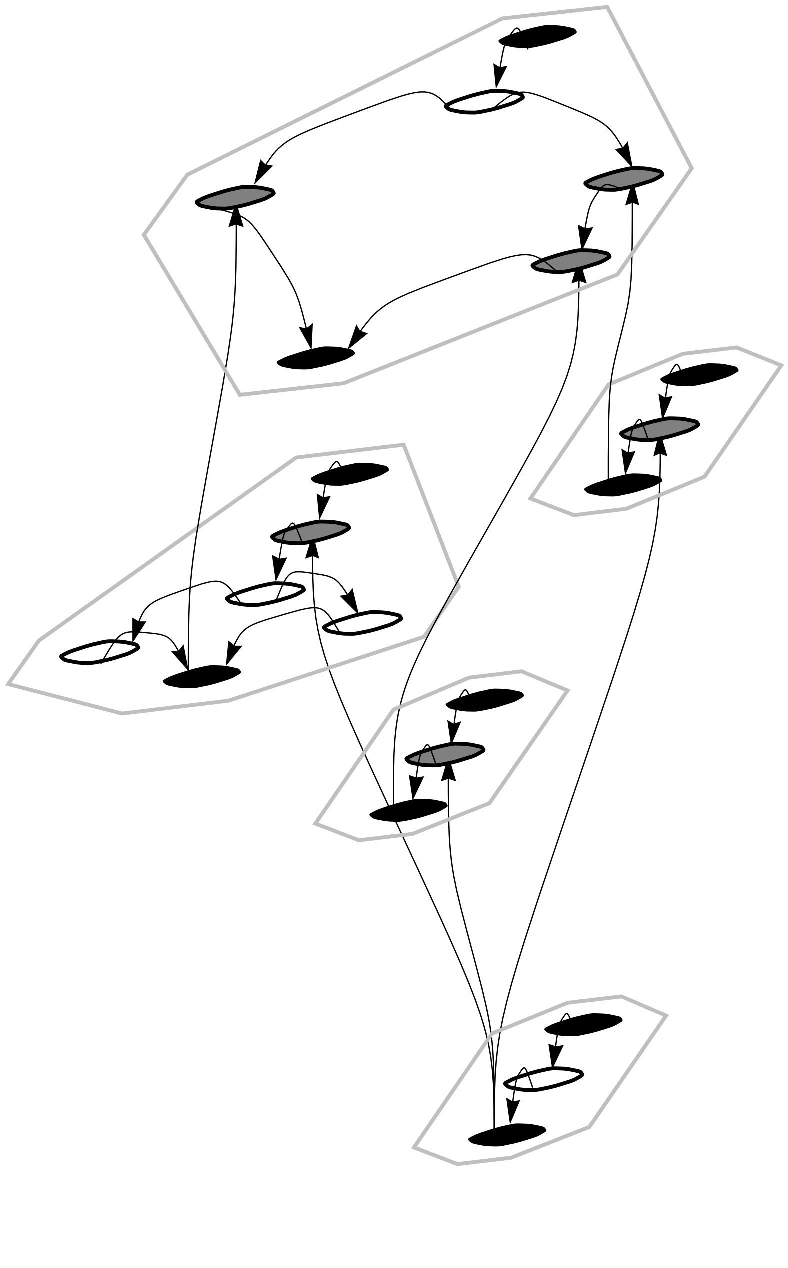


Figure 4-3. Bottom-up, forward-flow traversal

```
for each procedure $P$ from leaf procedures to main (bottom to top) do
    for each region $R = (N, E, s, e)$ from innermost to outermost of $P$ do
        for all nodes $n \in N$ do
            if $n$ is a basic block node then
                $l_n = Loc(n)$
            if $n$ is a loop node then
                $R' = ImmediateSubregion(n)$
                $l_n = V_{R'}{}^*$
            if $n$ is a call-site node then
                $R' = ImmediateSubregion(n)$
                $l_n = \underset{n}{\Uparrow} V_{R'}$
            if $n$ is the start node or the end node then
                $l_n = \bot$

        if forward-flow problem then
            for each node $n \in N$ do
                $V_n = \mathrm{T}\left( \bigwedge_{(n', n) \in E} V_{n'}, l_n \right)$
            $V_R = V_e$
        else if backward-flow problem then
            for each node $n \in N$ do
                $V_n = \mathrm{T}\left( \bigwedge_{(n, n') \in E} V_{n'}, l_n \right)$
            $V_R = V_s$
```

Figure 4-4. Algorithm for bottom-up regions-based data-flow analysis

each basic block node. For loop nodes and call site nodes, we derive the local values from the flow values of the immediate subregions. At each loop node, we apply the closure operator to the flow value of the immediate subregion (the loop body) and create a local-value that describes the effect of the entire loop. At a procedure call node, we apply the map operator that maps the flow value of the immediate subregion (the procedure body) from the callee space to the caller space by mapping the formals to actuals.

The local value of the start or end nodes is set to the initial flow value. Next, we calculate the flow value, V_n, at each node by using the meet operator to combine the incoming flow values from multiple control-flow edges, and then applying the transfer function to the combined incoming flow-value and the local value of the node. After propagating the flow value through all the nodes, we find the flow value, V_R, for the region.

4.1.2.2. Top-down traversal for flow-insensitive order

The only top-down traversal used in this thesis is flow-insensitive. Thus, we omit the forward-flow and backward-flow orders from the description. The general algorithm for a top-down traversal can be found in [76].

A flow-insensitive, top-down pass is used to propagate information into loop bodies and down the call chain. A flow value at a node is the cumulation of the local information of all the enclosing loops and procedure calls. The top-down analysis starts at the “main” procedure and moves toward the leaf procedures by following the call chains. Within each procedure, the analysis is performed from outer loops to inner loops. We analyze each region by propagating the incoming flow value through the nodes. Then, for loop and procedure nodes, we propagate the flow value to the immediate subregion. When the immediate subregion is a loop body, we analyze that region using this flow value. However, if the immediate subregion is a procedure body and if the procedure is called by multiple call sites, we may need procedure cloning. A procedure is cloned only if none of the clones created thus far has the same incoming flow value. Figure 4-5 shows the top-down, flow-insensitive propagation for the example code segment in Figure 4-1. The ordering of the traversal is given by the arrows in the diagram. We assume the flow values propagated to the subroutine FOO from the last two call sites are the same. Thus they both share a single clone,

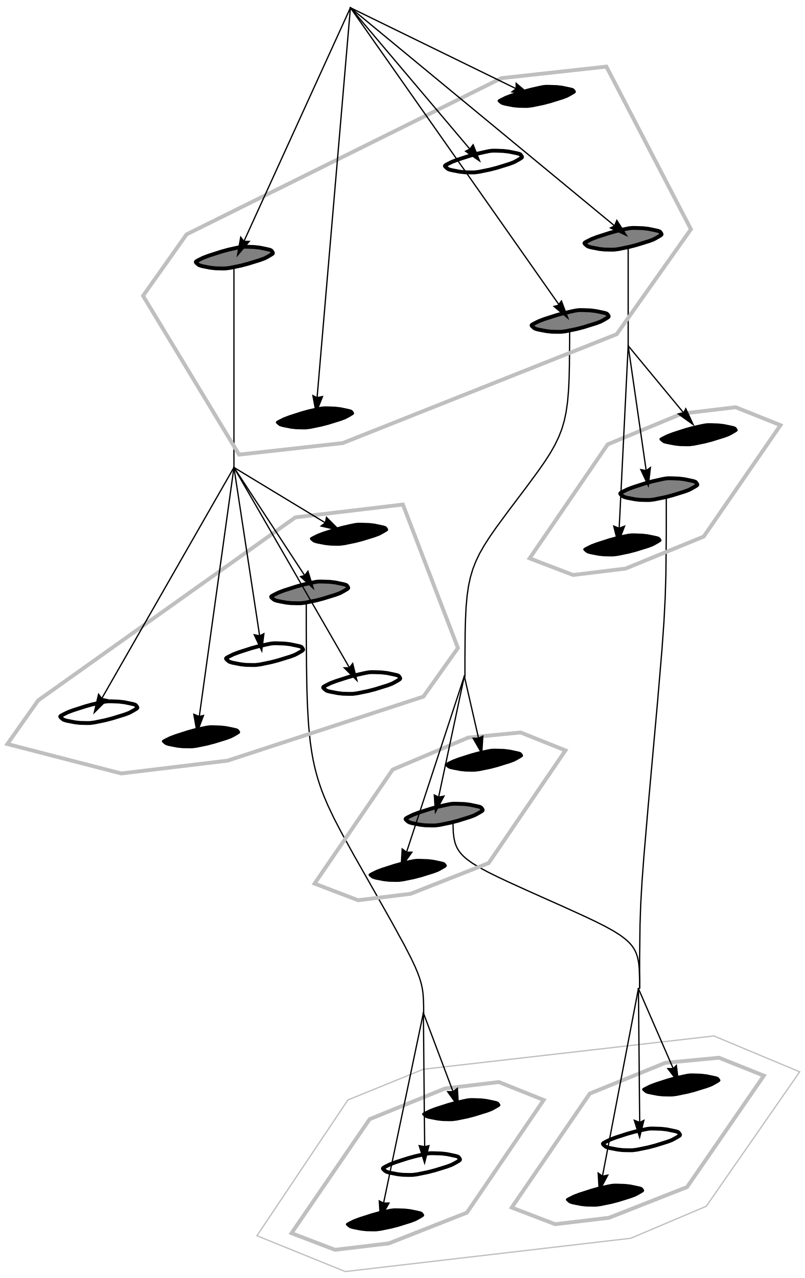


Figure 4-5. Top-down, flow-insensitive pass needing selective procedure cloning

which will be analyzed once when the flow value is propagated by the first call-site. The first call to subroutine FOO has a separate clone since its flow value is different.

The algorithm for top-down traversal is given in Figure4-6. The subroutine $TopDown$ recursively descends through the regions graph starting from the outermost region of the "main" procedure. The flow value of the region, V_R, is the incoming flow value. We find

```
$TopDown(R, V)$ : where region $R = (N, E, s, e)$ , and $V$ is the initial flow value
    for all nodes $n \in N$ do
        $l_n = Loc(n)$
        $V_n = \mathrm{T}(V, l_n)$

    for all loop nodes $n \in N$ do
        $TopDown(ImmediateSubregion(n), V_n)$
    for all call site nodes $n \in N$ do
        Let $p$ be the procedure called by $n$
        Let $C_p$ be the set of cloned regions for procedure $p$
        if there exist an $R' \in C_p$ such that $V_{R'} = \underset{n}{\Downarrow} V_n$ then
            Make $R'$ the immediate subregion of $n$
        else
            $R' = Clone(p)$
            $TopDown\left(R', \underset{n}{\Downarrow} V_n\right)$
            Add $R'$ to $C_p$
            Make $R'$ the immediate subregion of $n$
```

Figure 4-6. Algorithm for top-down analysis. Initiated with the call $TopDown(R_{MAIN}, \perp)$

the flow value at each node by applying the transfer function,T, to the flow-value of the region and the local value, $Loc(n)$, of the node. Next, at loop and procedure call nodes, we propagate the flow value to the immediate subregion. At a loop node, we analyze the immediate subregion with this flow value. At a procedure call node, we use the map operator to map the flow value of the call site node from the caller space to the callee space. Then we check the set of clones for the procedure to see if reanalysis is needed. If no clone exists with the same flow value, we create a new clone for the procedure and analyze it with the incoming flow value.

4.2. Loop Context Propagation

Each static instance of the program, denoted by a node in the regions graph, can have multiple dynamic invocations due to the execution of the enclosing loop nest. These invocations have different values for the index variables of the enclosing loop nest, which are often used in array access functions and inner loop bounds. The loop context captures the values of these index variables for each dynamic instance. We use the loop context information in array analyses, such as in data dependence analysis, to derive more accurate results.

The loop context at a node describes the bounds of the loop index variables in the node's enclosing loops. We represent the loop context concisely using a system of linear inequalities, a representation that is precise within the domain of affine loop bound expressions. The example in Figure 4-7 shows the system of inequalities representing the loop context of a loop nest.

4.2.1. The Data-Flow Problem

Loop context propagation is a single top-down, flow-insensitive traversal over the regions graph of the program. The analysis starts with an empty context at the outermost region of the "main" procedure and propagates the context in a top-down manner. At each loop node, we include the bounds of the loop index variable in the current context and propagate them to the loop body. If multiple call sites propagate different contexts to a procedure, the procedure is cloned. To reduce the number of clones created, we simplify the context by elim-

```
DO i = 1, M
```

$$\{\ (i) \mid 1 \le i \le M\ \}$$

```
    DO j = i, N
```

$$\left\{ (i, j) \mid \begin{array}{c} 1 \le i \le M \\ i \le j \le N \end{array} \right\}$$

Figure 4-7. An example of loop contexts for a loop nest

inating information that has no effect on the array analyses. The analyses of the caller's array accesses cannot be improved by knowing the bounds of the variables in the callee that are not accessible to the caller. Thus, we eliminate from the context the variables that are not accessible to the caller. We define the algorithm for loop context propagation by providing the functions used in the top-down pass algorithm in Section 4.1.2.2.

4.2.1.1. The local value function

The local value is the empty system for all but the loop nodes. The system of inequalities representing the loop bounds provides the local value at each loop node. For the loop `DO i = l to u`, with the corresponding node *n*, the local value is:

$$Loc(n) = \begin{cases} \{l \le i \le u\} & n \text{ is a loop node} \\ \{\ \} & \text{otherwise} \end{cases}$$

4.2.1.2. The transfer function

The transfer function incorporates the local constraints into the system of inequalities of the incoming flow-value from the enclosing loop nest. Let C be the context of the outer loop nest and t be the local constraints of the loop, then the transfer function to derive the flow value for the body of the loop is:

$$\mathrm{T}(C, t) = C \wedge t$$

4.2.1.3. The map operator

The map operator $\underset{n}{\Downarrow}$ first eliminates the inaccessible variables and then transforms the context across the procedure boundary. In the elimination step, loop index variables that are not accessible at the procedure body are projected away from the context using Fourier-Motzkin elimination. Next, the variables of the context that are actual variables of the callee space are renamed to the corresponding formal variables in the caller space.

4.3. Array Data-Flow Analysis

Array data-flow analysis is the most important phase in the parallelization process. This analysis summarizes the array accesses of the sub-region at each loop node. We use this information to parallelize loops by identifying arrays without any data dependences. The information also helps increase the available parallelism by identifying privatizable arrays that would otherwise prevent the parallelization of a loop. We perform the analysis using a single traversal over the regions graph of the program. For simplicity, in the following discussion we assume that the program contains only a single n-dimensional array. The analysis can be easily extended to the general case with multiple arrays.

4.3.1. Array Index Sets

In array data-flow analysis, we are required to summarize the effects of multiple dynamic instances of many different accesses to the array. The array summaries need to capture the access information at the granularity of array indices. This can be achieved by using an *index set* representation for the array summaries.

Definition 4-1: *The index set of the array is denoted by* A *. Each array index of the array is an n-tuple* $(a_1, \ldots, a_n) \in Z^n$ *such that* $(a_1, \ldots, a_n) \in \mathsf{A}$ *if and only if* $\underset{i = 1, \ldots, n}{\forall} l_i \le a_i \le u_i$ *where, from innermost to outermost dimension,* $l_1, \ldots, l_n$ *are the lower bounds and* $u_1, \ldots, u_n$ *are the upper bounds of the array.*

The set of all the possible array summaries is the power set of A.

In array data-flow analysis, we generate a summary for the array at each node of the regions graph. These summaries represent the accesses to the array within the subregion of the node. The accesses in the subregion, enclosed in loop nests and procedure bodies, can have multiple dynamic invocations at each invocation of the node. Moreover, the nodes are also enclosed within loop nests and procedure calls, and thus have multiple dynamic instances themselves. Therefore, at each node we need the ability to summarize not only the effect of all dynamic invocations within the subregion but also differences among the multiple invocations of the node. Creating a separate summary for each dynamic instance of a node is not viable. Therefore, we generate a summary at a node that is valid for all the dynamic instances of that node. However, the use of a simple array index set is not sufficient to produce an accurate summary that is valid for all the dynamic instances of the node. This requires the summary information to be parameterized by the instances of the loop context of the node. Thus, we define a*parameterized index set*, a set of array indices that are parameterized by the variables defining the dynamic instance.

Definition 4-2: *A parameterized index set of an array at a node with a context C is a function r where, for all instances $(i_1, \ldots, i_x) \in C$, the function $r(i_1, \ldots, i_x) \subseteq \mathsf{A}$.*

One or more parameters can be eliminated from a parameterized index set by assigning them actual integer values. We define a*projection function* that eliminates a parameter by assigning a value range to that parameter. For each integer value in the range, a new parameterized index set is created. The projection function returns the union of these parameterized index sets.

Definition 4-3: *The projection function maps a parameterized index set r to a parameterized index set r' such that $r' = proj(r, k, l, u)$, where*

$$r'(i_1, \ldots, i_{k-1}, i_{k+1}, \ldots, i_x) = \bigcup_{l \le i_k \le u} r(i_1, \ldots, i_x)$$

$i_1, \ldots, i_k, \ldots, i_x$ are integer variables, l and u are upper bound and lower bound expressions.

We also define a reshape operator to transform an array index set across procedure boundaries. The operator $reshape_n(r)$ transforms an array index set r of a formal array variable at the callee procedure to the corresponding array index set of the actual array at the call site n. The reshape operator is an injective mapping from indices of a formal array at a called procedure to the indices of the corresponding actual array at the call site. An inverse mapping from call site in the caller to callee can also be defined.

4.3.2. The Flow Value of the Array Data-Flow Problem

The array data-flow analysis calculates four parameterized index sets at each node. These sets, at a node n, contain the indices that are accessed in a program section defined by the node n and its subgraph. The four sets are:

- The *read set* R, which contains all the array indices that may be used by a read access in a valid execution path of the program section.

- The *exposed read set* E, which contains all the array indices that may be used by a read array access in a valid execution path but have no preceding write array access in the same path. These exposed read accesses use values that were defined outside the program section.

- The *write set* W, which contains all the array indices that may be defined by a write access in some valid execution path of the program section.

- The *must write set* M, which contains the array indices that are definitely defined by a write access in all the valid execution paths of the program section.

Thus, array data-flow analysis calculates a four-tuple $\langle R, E, W, M \rangle$ at node n, where:

$$\begin{aligned}
R &= \{a \mid \text{array element } a \text{ may be used in } n\} \\
E &= \{a \mid \text{array element } a \text{ may be an outward exposed use in } n\} \\
W &= \{a \mid \text{array element } a \text{ may be defined in } n\} \\
M &= \{a \mid \text{array element } a \text{ must be defined in } n\}
\end{aligned}$$

We have defined the above four sets such that these values do not need to be exact. It is not always possible to find the exact set of indices that are accessed when the corresponding code is executed since that information may be undecidable at compile time. Therefore, the exact parameterized index set for many of the array access functions, loop bound expressions, etc., cannot be created at compile time. Furthermore, the operators on array index sets in a given representation may not be exact. Thus, we calculate a valid approximation of the exact value in our algorithm. Let R_{exact} be the exact parameterized index set for all the reads in a program segment, E_{exact} for all the exposed reads, W_{exact} for all the writes and M_{exact} for all the must writes. The value $\langle R, E, W, M\rangle$ is a valid approximation for that program segment if and only if R, E, W are over approximations of the exact value and M is an under approximation of the exact value. That is, $R \supseteq R_{exact}$, $E \supseteq E_{exact}$, $W \supseteq W_{exact}$ and $M \subseteq M_{exact}$.

4.3.3. The Data-Flow Problem

The array data-flow analysis is defined as a single top-down, backward-flow problem in the regions-based data-flow framework. The algorithm for array data-flow analysis is defined by providing the functions needed by the top-down pass in Section 4.1.2.2.

4.3.3.1. The local value function

The local value function Loc generates a four-tuple $\langle R, E, W, M\rangle$ for the array accesses in each basic block. While the conversion of array accesses with affine access functions to a parameterized index set is exact, we also need to generate conservative approximations when exact information is not available. If multiple array accesses are present in a basic block, we create multiple nodes with a single array access per node when building the regions graph. The local value for the array at a basic block n with a context descriptor C is the four-tuple $\langle R, E, W, M\rangle$. When the basic block has:

i) no accesses to the array, the four-tuple is $\langle \emptyset, \emptyset, \emptyset, \emptyset\rangle$.

ii) a read access $A(f_1, \ldots, f_n)$ where $f_1, \ldots, f_n$ are functions known at compile-time and parameterized by the context C, the four-tuple is $\langle \{ (f_1, \ldots, f_n) \}, \{ (f_1, \ldots, f_n) \}, \emptyset, \emptyset\rangle$.

iii) a write access $A(f_1, \ldots, f_n)$ where $f_1, \ldots, f_n$ are functions known at compile-time and parameterized by the context C, the four-tuple is $\langle \emptyset, \emptyset, \{(f_1, \ldots, f_n)\}, \{(f_1, \ldots, f_n)\} \rangle$.

iv) a read access $A(f_1, \ldots, f_n)$ where at least one of $f_1, \ldots, f_n$ is not known at compile time, the four-tuple is $\langle \mathsf{A}, \mathsf{A}, \emptyset, \emptyset \rangle$.

v) a write access $A(f_1, \ldots, f_n)$ where at least one of $f_1, \ldots, f_n$ is not known at compile time, the four-tuple is $\langle \emptyset, \emptyset, \mathsf{A}, \emptyset \rangle$.

vi) an unknown access, the four-tuple is $\langle \mathsf{A}, \mathsf{A}, \mathsf{A}, \emptyset \rangle$.

The local value, $Loc(n)$, is always a valid approximation of the array indices accessed by the program segment in node n and its subgraph. For W, R and E of part i, R and E of part ii and iv and W of part iii and v, the empty set $\emptyset$ is the exact result since there are no accesses to the array. In all other cases of W, R and E, the result is either the exact array index or an over approximation given bythe entire index set A. The set M is either the exact array index or an under approximation given by the empty set $\emptyset$. Thus, the local value $Loc(n)$ is a valid approximation of the array accesses in the basic block n.

4.3.3.2. The transfer function

The transfer function T takes the local value $\langle R_{loc}, E_{loc}, W_{loc}, M_{loc} \rangle$ at a node and an incoming flow value $\langle R_{in}, E_{in}, W_{in}, M_{in} \rangle$ as input and creates the outgoing flow value

$$\langle R_{loc} \cup R_{in}, E_{loc} \cup (E_{in} - M_{loc}), W_{loc} \cup W_{in}, M_{loc} \cup M_{in} \rangle.$$

If the local value and incoming flow value are valid approximations, the outgoing flow value is also a valid approximation. The resulting parameterized index sets $R_{loc} \cup R_{in}$ and $W_{loc} \cup W_{in}$ are supersets of the exact value and $M_{loc} \cup M_{in}$ is a subset of the exact value. Since E_{in} is a superset of the exact value and M_{loc} is a subset, $E_{in} - M_{loc}$ is also a superset of the exact value. Thus $E_{loc} \cup (E_{in} - M_{loc})$ is a superset of the exact value. This conforms to the definition of the flow value.

4.3.3.3. The meet operator

The meet operator of two flow values, $\langle R_1, E_1, W_1, M_1\rangle$ and $\langle R_2, E_2, W_2, M_2\rangle$, produces the flow value

$$\langle R_1 \cup R_2, E_1 \cup E_2, W_1 \cup W_2, M_1 \cap M_2\rangle.$$

4.3.3.4. The closure operator

The closure operator takes the flow value $\langle R, E, W, M\rangle$ at the immediate subgraph of the loop node of the loop `DO i = l to u` and returns the flow value for the loop node

$$\left\langle \begin{matrix} proj(R, i, l, u), proj\left(E - proj\left(M|_i^{i'}, i', l, i-1\right), i, l, u\right), \\ proj(W, i, l, u), proj(M, i, l, u) \end{matrix} \right\rangle,$$

where i' is a new variable. The projection operator, $proj$, is given in Definition 4-3.

4.3.3.5. The map operator

The map operator takes a flow value $\langle R, E, W, M\rangle$ at a procedure node and returns the flow value for the call-site node n

$$\langle reshape_n(R), reshape_n(E), reshape_n(W), reshape_n(M)\rangle.$$

The function $reshape_n$ is defined in Section 4.3.1.

4.4. Parallel Loop Detection

At each loop, we need to determine if that loop can be executed in parallel. We are only interested in the variables declared outside the scope of the loop and modified inside the loop body. We test these variables for loop-carried dependences and perform optimizations to eliminate the loop-carried dependences when possible. In this presentation we assume that testing of the scalar variables has already been done. All the scalar variables should be candidates for either privatization or reduction optimizations. For all the array variables, we use the results of the data-flow problem to identify if the loop can execute in parallel with respect to each array.

4.4.1. Location-Based Dependences

First we perform a location-based data-dependence test to identify the existence of loop-carried true-, anti-, and output-dependences in the array.

Theorem 4-1: *At the loop* `DO i`$_n$ `= 0 to u` *with the flow value* $\langle R(i_1, \ldots, i_n), E(i_1, \ldots, i_n), W(i_1, \ldots, i_n), M(i_1, \ldots, i_n) \rangle$*, there is a loop-carried true-dependence iff* $\exists k, l \;\; 1 \le k < l \le u$ *such that* $W(i_1, \ldots, i_{n-1}, k) \cap R(i_1, \ldots, i_{n-1}, l) \neq \emptyset$.

Theorem 4-2: *At the loop* `DO i`$_n$ `= 0 to u` *with the flow value* $\langle R(i_1, \ldots, i_n), E(i_1, \ldots, i_n), W(i_1, \ldots, i_n), M(i_1, \ldots, i_n) \rangle$ *there is a loop-carried anti-dependence iff* $\exists k, l \;\; 1 \le k < l \le u$ *such that* $R(i_1, \ldots, i_{n-1}, k) \cap W(i_1, \ldots, i_{n-1}, l) \neq \emptyset$.

Theorem 4-3: *At the loop* `DO i`$_n$ `= 0 to u` *with the flow value* $\langle R(i_1, \ldots, i_n), E(i_1, \ldots, i_n), W(i_1, \ldots, i_n), M(i_1, \ldots, i_n) \rangle$*, there is a loop-carried output-dependence iff* $\exists k, l \;\; 1 \le k < l \le u$ *s. t.*
$W(i_1, \ldots, i_{n-1}, k) \cap W(i_1, \ldots, i_{n-1}, l) \neq \emptyset$.

4.4.2. Value-Based Dependences

If a location-based loop-carried dependence exists for any array, the loop may still be parallelized if there are no value-based, loop-carried flow-dependences or if the dependences are due to a reduction operation. When there are no loop-carried flow-dependences, the location-based dependences can be eliminated by giving each processor a private copy of the array. When a memory location updated using a commutative and associative reduction operation, the accesses will create a loop-carried dependence. However, we can safely parallelize the loop by replacing the reduction with a parallel version since the ordering of the commutative updates need not be preserved. The updates are applied to a local copy during the parallel execution of the loop. The program performs a global accumulation following the parallel loop execution. Array reductions are futher described in [75,76].

Theorem 4-4: *At the loop* `DO i`$_n$ `= 0 to u` *with the flow value* $\langle R(i_1, \ldots, i_n), E(i_1, \ldots, i_n), W(i_1, \ldots, i_n), M(i_1, \ldots, i_n) \rangle$*, the array cannot be privatized iff* $\exists k, l \;\; 1 \le k < l \le u$ *such that* $W(i_1, \ldots, i_{n-1}, k) \cap E(i_1, \ldots, i_{n-1}, l) \neq \emptyset$.

However, it may not be possible to produce efficient code for a privatized array due to the need for *finalization*. At the end of the loop, the original copy of the privatized array must have the most up-to-date values. But all the updates within the loop nests were made to the private copy. Thus, we need to identify the last update to each location and copy it to the original array. In general this is a very expensive operation. Hence, we restrict privatization to arrays where the last iteration of the loop will be overwriting all the indices updated by any previous iteration. In this case, we can create the correct final values for the original array by allowing the processor that executes the last iteration to use the original array while all the other processors use a private copy.

Theorem 4-5: *At the loop* `DO i`$_n$ `= 0 to u` *with the flow value* $\langle R(i_1, \ldots, i_n), E(i_1, \ldots, i_n), W(i_1, \ldots, i_n), M(i_1, \ldots, i_n)\rangle$, *the array can be finalized after privatization, by assigning the original array to the processor executing the last iteration, iff* $\underset{k = 1, \ldots, u}{\forall} W(i_1, \ldots, i_{n-1}, k) \subseteq W(i_1, \ldots, i_{n-1}, u)$.

4.5. Determining the Outermost Parallel Loops

Determining the outermost parallel loops is defined as a single data-flow problem in a top-down, flow-insensitive pass using the regions-based interprocedural data-flow framework. After solving the data-flow problem, we assign each loop a value from the set $\{outerSeq, parallel, innerSeq\}$. The outermost parallel loops will be marked *parallel* while the outer sequential loops and loops inside parallel regions will be marked *outerSeq* and *innerSeq* respectively. The algorithm for determining the outermost parallel loops is defined by providing the functions needed by the top-down pass algorithm in Section 4.1.2.2.

- We define the local value at each node to be

$$Loc(n) = \begin{cases} parallel & n \text{ is a parallel loop} \\ outerSeq & \text{otherwise} \end{cases}$$

- The transfer function is defined as

- The closure operator $V^* = V$

$$T(V, l) = \begin{cases} outerSeq & (V = outeSeq) \wedge (l = outerSeq) \\ parallel & (V = outerSeq) \wedge (l = parallel) \\ innerSeq & (V = parallel) \vee (V = innerSeq) \end{cases}$$

- The map operator $\underset{n}{\Uparrow} V = V$

We parallelize the outermost parallel loops after the appropriate transformations to implement scalar and array privatization and reduction for the loops.

4.6. Related Work

Researchers have discovered that it is necessary to go beyond the traditional scalar data-flow and array data dependence analysis in automatic parallelization of sequential scientific applications. Successful parallelization requires advanced analysis techniques such as array data-flow analysis used for array privatization [52,113,131]. There have been two major approaches in finding data-flow information for array elements. The first approach builds on data dependence analysis, and the second on scalar data-flow analysis.

The first approach, pioneered by Feautrier, uses the same framework as the data dependence analysis. This approach finds the perfect data-flow information for arrays in the domain of loop nests where the loop bounds and array indices are affine functions of the loop indices [53,54,55]. We have devised a more efficient algorithm than Feautrier's for obtaining data-flow information that is applicable to many common cases found in practice [112]. Several other researchers have taken a similar approach to data-flow analysis [28,121,122].

However, none of these algorithms handle general control flow in a direct or efficient manner. Extending the pair-wise data dependence framework is not efficient in handling a large number of array accesses. Furthermore, the presence of multiple writes makes solving the exact data-flow problem very complex and prohibitively expensive. Thus, this approach is not practical for large coarse-grain loop nests with complex control flow and a multitude of array accesses.

The other approach, used in our algorithm, is based on extending the scalar data-flow framework. Array data-flow analysis is formulated as a problem in the data-flow framework. Instead of representing an array with a single bit, the set of data touched within a region/interval in the flow graph is approximated by an array index set. In this approach, we are able to efficiently handle arbitrary control flow by using conservative meet operators and multiple accesses by merging summary information. Many researchers have proposed this approach for array data-flow analysis [18,45,64,66,124,137,142]. The greatest improvement of our algorithm over previous work is the increased accuracy of our array region representation.

Our array summary representation, based on sets of convex polyhedrons, is most similar to the single convex polyhedron representation used in the PIPS project [45,46,86]. However, we will show in Chapter 5 that our representation is more accurate. Furthermore, their algorithm restricts write regions to be either a single over or an under approximation. In our algorithm, we calculate both an over approximation (write) and an under approximation (must write). Thus, we are not forced to lose under approximation information, required for array privatization, in the presence of over approximations.

Unlike many of these previous studies, we have implemented our array data-flow algorithm in a full interprocedural parallelizer. We demonstrate the applicability of our analysis in Chapter 7, by automatically parallelizing a large collection of benchmark programs. Another implementation of interprocedural array data-flow analysis can be found in the Polaris parallelizing compiler [22,25]. In their algorithm, array privatization is applied only to the cases where all the values used in an iteration are defined before they are used in the same iteration [142]. However, array privatization is also applicable to loops in which iterations use values computed outside the loop, where the private copies must be initialized with these values before parallel execution begins. Our algorithm identifies privatizable arrays that require initialization.

4.7. Chapter Summary

This chapter presents the array analyses used in our parallelizer. Our algorithm calculates both location-based and value-based dependences to locate parallelizable loops. We first introduce the interval-based interprocedural framework used by the algorithm. The array analysis is divided into four phases. The first phase propagates the loop context information used to increase the precision of array analysis. Next, we derive array data-flow information at each loop using an array summary representation. Then, we use the array data-flow information to identify data-dependences and privatizable arrays. Finally, we identify the outermost parallel loops.

5 Array Summary Representation

For the array analysis defined in the previous chapter to be practical, we must have an expressive, compact, and efficient array summary representation. In this chapter we define such a representation based on systems of linear inequalities.

Since no practical array summary representation can precisely represent any arbitrary access pattern, we need to find a compact representation with the ability to precisely represent many access patterns found in practice. The array summary representation should efficiently execute operations on array index sets such as union, intersection, difference, and projection. The cost of maintaining the array summaries as well as performing operations on them increases with the precision of the representation. Thus in designing the summary representation, we have to arrive at a balance between precision and cost. The array summaries need to maintain sufficient precision to perform the required analysis without losing information for most cases found in practice. But the cost of generating and maintaining the information should not be prohibitively expensive.

We have imposed an additional requirement on the precision of the array summary representation. We want the data dependence test based on the summary representation to be at least as precise as the pair-wise array data dependence test that it will replace [110]. The pair-wise data dependence test is exact over the *affine domain*. An array access is in the affine domain when the index function of the array access and lower and upper bounds of the enclosing loops of interest are affine expressions with respect to loop index variables and loop constants.

We have developed an array summary representation based on systems of linear inequalities that satisfy the above criteria. We represent convex regions of an array by a system of linear inequalities called a *convex array section*. We use a list of convex array sections, an *array section descriptor*, as the general representation of array summaries.

In this chapter we introduce the convex array sections in Section 5.1 and array section descriptors in Section 5.2. We show how to create a convex array section for a sparse region in Section 5.4. The operations on convex array sections and array section descriptors are defined in Sections 5.4. and 5.5. respectively. The related works are given in Section 5.6. As in Chapter 4, we simplify the following discussion by assuming that the program contains only a single *n*-dimensional array.

5.1. Convex Array Section

We use convex array sections as a practical representation for the parameterized index sets introduced by Definition 4-2. A convex array section can precisely represent the class of parameterized index sets, where all the indices of the index set can be represented as a set of integer points within a multi-dimensional convex polyhedron. We use a system of linear inequalities to describe this convex polyhedron. The inequalities are parameterized by the variables of the loop context associated with the parameterized index set as well as the set of *dimension variables* of the array. These special dimension variables hold the index values of each dimension of the array.

Definition 5-1: *A convex array section*

$$R = \left\{ (a_1, \ldots, a_n) \;\middle|\; \begin{array}{c} c_0^1 + c_1^1 a_1 + \ldots + c_n^1 a_n + c_{n+1}^1 i_1 + \ldots + c_{n+x}^1 i_x \geq 0 \\ \cdots\cdots\cdots \\ c_0^m + c_1^m a_1 + \ldots + c_n^m a_n + c_{n+1}^m i_1 + \ldots + c_{n+x}^m i_x \geq 0 \end{array} \right\}$$

defines a parameterized index set where $i_1, \ldots, i_x$ *are the variables of the loop context associated with the parameterized index set,* $a_1, \ldots, a_n$ *are variables representing each of the dimensions of the n-dimensional array, and all c's are integers.*

Figure 5-2 shows the different parameterized index sets of the regions graph for the example in Figure 5-1. The innermost region, with only one array access, is represented by a convex array section that denotes a single array index. The convex array section is param-

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

Figure 5-1. A loop nest with an array access

eterized by the loop index variables *I* and *J*, the loop invariant variables *M*, *N* and *C* and dimension variables a_1 and a_2. The region that includes the innermost loop, *J*, is represented by a convex array section containing the array elements accessed by all the iterations of the loop *J* for a given iteration of the loop *I*. The parameterized index set describing the entire region is a convex array section containing the array elements accessed by all iterations of both loops.

Next, we define the index set A and the empty set $\varnothing$ using the convex array section representation.

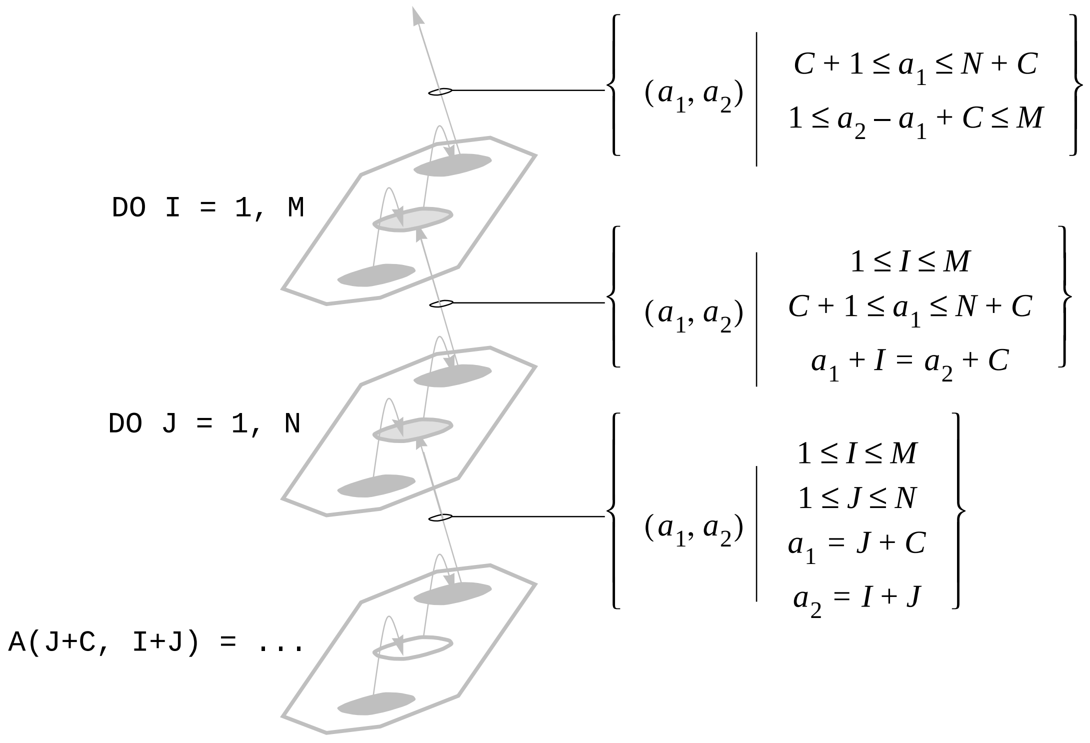
Definition 5-2: *The index set of all the indices of the array is given by the convex array section*

$$\mathsf{A} = \left\{ (a_1, a_2, \ldots, a_n) \;\middle|\; \begin{array}{c} u_1 \le a_1 \le l_1 \\ \ldots \\ u_n \le a_n \le l_n \end{array} \right\}$$

where integers $l_1, \ldots, l_n$ and $u_1, \ldots, u_n$ are the lower and upper bounds of the array dimensions.

Definition 5-3: *The empty set is given by the convex array section in canonical form where the system of inequalities is always false.*

$$\varnothing = \{ (a_1, a_2, \ldots, a_n) \mid 0 > 1 \}$$



$$\left\{ (a_1, a_2) \;\middle|\; \begin{array}{c} C + 1 \le a_1 \le N + C \\ 1 \le a_2 - a_1 + C \le M \end{array} \right\}$$

$$\left\{ (a_1, a_2) \;\middle|\; \begin{array}{c} 1 \le I \le M \\ C + 1 \le a_1 \le N + C \\ a_1 + I = a_2 + C \end{array} \right\}$$

$$\left\{ (a_1, a_2) \;\middle|\; \begin{array}{c} 1 \le I \le M \\ 1 \le J \le N \\ a_1 = J + C \\ a_2 = I + J \end{array} \right\}$$

Figure 5-2. Summarizing the array access patterns

Next, we express an affine array access function using the convex array section representation. This formulation is used in calculating the local values as defined in Section4.3.3.1.

Definition 5-4: *An affine array access* $A\left(c_0^1 + c_1^1 i_1 \ldots + c_x^1 i_x, \ldots\ldots, c_0^n + c_1^n i_1 \ldots + c_x^n i_x\right)$ *is represented by the convex array section*

$$\left\{ (a_1, \ldots, a_n) \;\middle|\; \begin{array}{c} a_1 = c_0^1 + c_1^1 i_1 + \ldots + c_x^1 i_x \\ \ldots \\ a_n = c_0^n + c_1^n i_1 + \ldots + c_x^n i_x \end{array} \right\}$$

where, all c's are integers and $i_1, \ldots, i_x$ *are the variables of the loop context associated with the array access.*

5.2. Array Section Descriptors

Although each affine array access can be represented using a convex array section, many operations on these convex array sections produce non-convex results. By using a single convex section to approximate a non-convex region, we lose a significant degree of precision. Because of this loss of accuracy, approximating non-convex regions using a convex section is unacceptable. Therefore, for parameterized index sets we need a more general representation than the convex array sections. We use an*array section descriptor*, a list of convex array sections, to represent a general array index set. Each array section descriptor can have one or more convex array sections. Thus, non-convex regions can be represented using multiple convex array sections. In theory, any arbitrary array index set can be represented using a list of convex array sections by dividing the index set into convex regions. However, in practice we avoid creating large lists.

Definition 5-5: *An array section descriptor D is a list of convex array sections $\{R_1, \ldots, R_i, \ldots, R_k\}$, where an array index $(a_1, \ldots, a_n) \in D$ iff $(a_1, \ldots, a_n) \in R_i$.*

We use a canonical form for the array section descriptor, where, for a given array section descriptor D,

i) there does not exist $R_i \in D$ such that $R_i \equiv \varnothing$
ii) there does not exist $R_i, R_j \in D$, $i \neq j$ such that R_i is contained in R_j

These properties do not affect the functionality of the array section descriptors but help make the implementation more concise. We allow overlapping convex sections in an array section descriptor. Requiring convex array sections of a descriptor to be non-overlapping would not increase the precision of the results. However, it would make the operations on array section descriptors more complicated and expensive.

5.3. Sparse Array Regions

A convex array section, as defined in Definition5-1, can only represent index sets which are *dense convex polyhedrons*, where all the integer points within the convex polyhedron are in the index set. However, in practice we need to represent sparse convex polyhedrons, where only a subset of integer points within a convex region are in the index set. The example loop in Figure 5-3 accesses only the even elements of the array, resulting in a sparse access pattern. There are two possible methods of representing a sparse pattern within our framework. One way is to fragment the sparse region into a set of dense convex regions. However, the number of regions required is not known at compile-time for many parameterized accesses. Furthermore, the number of dense convex regions is dependent on the size of the sparse region, which can be quite large. Instead, we choose to construct a single system using *auxiliary variables*, special variables used in creating linear constraints to represent the sparse nature of the access patterns. These variables can be viewed as additional dimensions of the parameterized convex polyhedron. The Figure5-4 shows how an auxiliary variable is used to represent a non-dense array region for the loop nest given in Figure 5-3. In this case, all the even indices of the one-dimensional array between indices 1 and *N* are represented using an auxiliary variable α.

```
DO I = 1, N
     A(2I)
```

Figure 5-3. A simple example creating sparse access pattern

When the same sparse pattern arises in multiple sections, each section introduces a unique auxiliary variable. Thus, union, intersection or subtraction operations on two of these sections will create a section that has multiple redundant auxiliary variables representing the same sparse pattern. In Section 5.4.8, we show how to simplify the resulting section by eliminating these redundant auxiliary variables. We also handle auxiliary variables as a special case in our union algorithm given in Section 5.4.3.

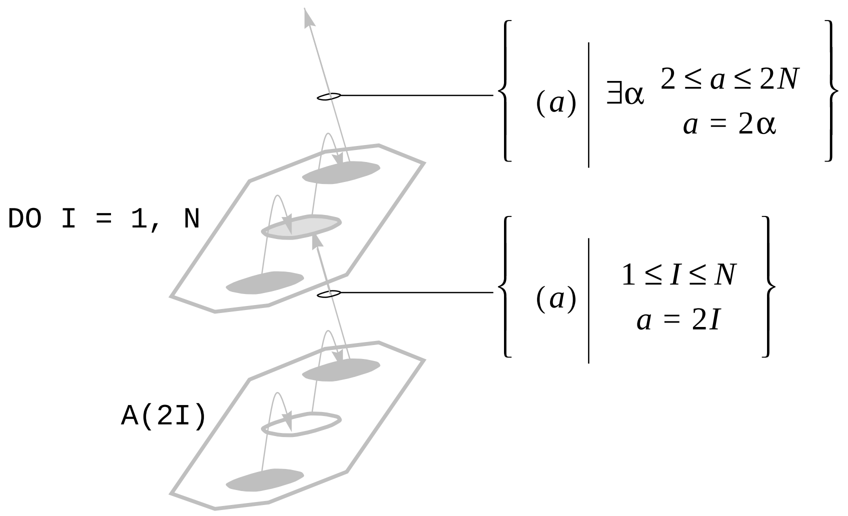

DO I = 1, N
A(2I)
$\left\{ (a) \,\middle|\, \exists\alpha \begin{array}{c} 2 \le a \le 2N \\ a = 2\alpha \end{array} \right\}$
$\left\{ (a) \,\middle|\, \begin{array}{c} 1 \le I \le N \\ a = 2I \end{array} \right\}$

Figure 5-4. An array summary with an auxiliary variable

5.4. Operations on Convex Regions

We define several operations useful for manipulating array index sets. Some of these operators, such as subtraction, are only approximations of the set operators.

5.4.1. Empty Test

The operator $IsEmpty$ is a boolean function that returns false if the convex array section contains any valid array indices. An empty array region implies that no integer solution exists for the system of inequalities. Therefore, the empty test is implemented using Fourier-Motzkin elimination, which finds the existence of an integer solution for the system.

5.4.2. Intersection Operator

The intersection operator finds the common array indices in multiple array sections. The function $Intersect\ (R_1, R_2)$ returns a convex array section $R_1 \cap R_2$, where R_1 and R_2 are convex array sections of the same array. The implementation of the intersection operator is very simple. The inequalities of both systems are combined to form a single system. Intersecting two sections with no common array indices will result in a system of inequalities with no solution. We eliminate these systems by checking for empty sections after the intersection. Moreover, the resulting system may have many redundant inequalities and auxiliary variables. The algorithms to eliminate the redundant inequalities and auxiliary variables are given in Section 5.4.8.

5.4.3. Union Operator

A union of two convex array sections contains all the array indices of both sections. However, as shown by the two examples in Figure5-5, the union of two convex sections may not be convex. In our algorithm, we keep both convex array sections to precisely represent the resulting region. Since this requires a list of convex array sections in the representation, the definition of the union operator will be deferred until Section 5.5.3.

Figure 5-5. Examples of unions of two convex sections resulting in a non-convex section

In many instances found in practice, the union of two convex regions is a single convex region. This is illustrated by the two examples in Figure5-6. The array index sets for the examples are given in Figure 5-7. In the first example, the odd and even indices of a one-dimensional array are written by two write statements. The two convex array sections of the write statements can be merged into a single convex array section. In the second example, the elements of the lower triangle and the diagonal of a two dimensional array are updated separately. The two sections can be merged into a single convex array section representing both regions.

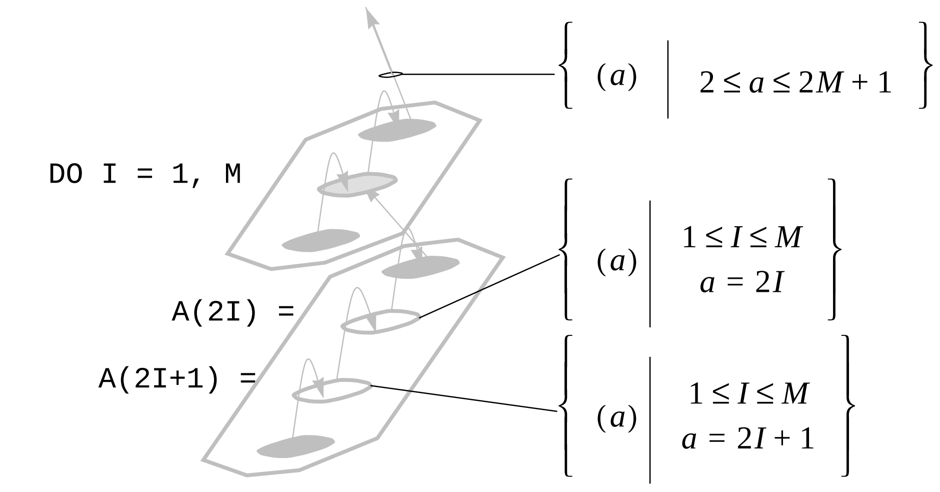
```
DO I = 1, M
      A(2I) = ..
      A(2I+1) = ...

DO I = 1, M
      A(I,I)
DO I = 1, M-1
      DO J = 1, I-I
```

Figure 5-6. Two examples of loop nests where the convex array sections can be merged after union operator.

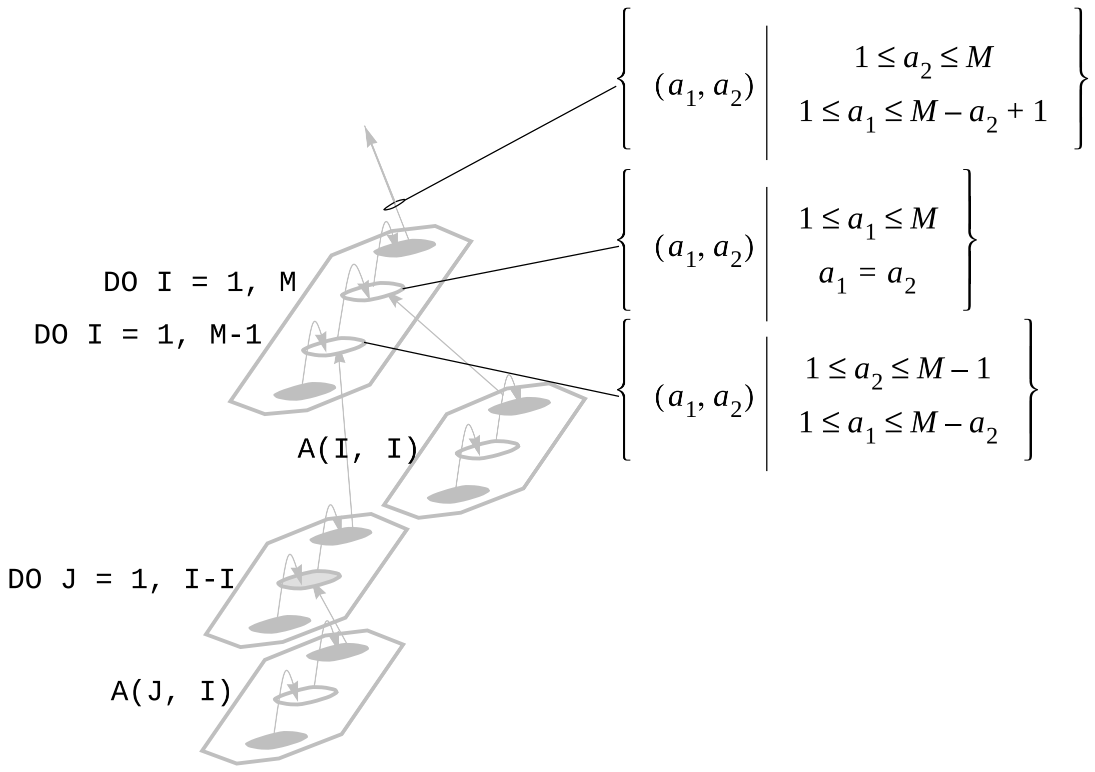
5.4.3.1. A simple merge algorithm

First, we describe a merge algorithm that attempts to merge two convex array sections without any special treatment of the auxiliary variables. Merging two convex array sections where one is contained in the other is trivial. The result of the merge is the convex array section that contains the other. However, merging two arbitrary convex array sections, even when the result is convex, is non-trivial. All the elements of each input convex array region are in the result of the merge.

We have developed an algorithm, presented in Figure 5-8, that will merge two convex array sections when the merge can be performed by eliminating exactly one inequality from each convex array section. The negation operator, $\neg$, used in the algorithm is implemented by negating all the coefficients and the offset of the inequality and subtracting one from the offset. Since it is not always possible to merge two convex array sections, the *mergeSimple* algorithm returns a tuple with a convex array section and a boolean. If a valid merge is found, a tuple with the convex array section and *true* will be returned; otherwise, the boolean value of the returned tuple will be *false*.



DO I = 1, M
A(2I) =
A(2I+1) =
$\{ (a) \mid 2 \le a \le 2M+1 \}$
$\{ (a) \mid 1 \le I \le M,\ a = 2I \}$
$\{ (a) \mid 1 \le I \le M,\ a = 2I+1 \}$



DO I = 1, M
DO I = 1, M-1
A(I, I)
DO J = 1, I-I
A(J, I)
$\{ (a_1, a_2) \mid 1 \le a_2 \le M,\ 1 \le a_1 \le M - a_2 + 1 \}$
$\{ (a_1, a_2) \mid 1 \le a_1 \le M,\ a_1 = a_2 \}$
$\{ (a_1, a_2) \mid 1 \le a_2 \le M-1,\ 1 \le a_1 \le M - a_2 \}$

Figure 5-7. Examples of convex array sections that can be merged after union operator.

```
$MergeSimple(R_1, R_2) \rightarrow \langle R, \{true, false\} \rangle$
where $R_1$, $R_2$ and $R$ are convex array sections

        if $IsContained(R_1, R_2)$ then
            return $\langle R_2, true \rangle$
        else if $IsContained(R_2, R_1)$ then
            return $\langle R_1, true \rangle$
        else
            $R_1' = R_1$
            for each linear inequality $I \in R_1$ do
                if $IsEmpty(Intersect(R_2, \{\neg I\}))$ then
                    Remove inequality $I$ from $R_1'$
            $R_2' = R_2$
            for each linear inequality $I \in R_2$ do
                if $IsEmpty(Intersect(R_1, \{\neg I\}))$ then
                    Remove inequality $I$ from $R_2'$
            if $R_1' \equiv \{I_1\}$ and $R_2' \equiv \{I_2\}$
            where $I_1$ and $I_2$ are single linear inequalities then
                remove $I_1$ from $R_1$
                remove $I_2$ from $R_2$
                if $IsEmpty\left(Intersect\left(R_1, R_2, \{\neg I_1\}, \{\neg I_2\}\right)\right)$ then
                    return $\langle Intersect(R_1, R_2), true \rangle$
        return $\langle \varnothing, false \rangle$
```

Figure 5-8. Attempts to merge two convex array sections without any special treatment on auxiliary variables

5.4.3.2. Merge algorithm in the presence of auxiliary variables

When both sections have auxiliary variables, the *mergeSimple* algorithm considers these auxiliary variables as separate variables, and thus does not succeed in merging the sparse access patterns. However, we can successfully merge many sparse patterns when the properties of the auxiliary variables are taken into account. The algorithm *mergeAuxVars* in Figure 5-9 attempts to merge two sparse patterns given by two different auxiliary variables into a single sparse pattern with a new auxiliary variable. The *merge* algorithm, in Figure 5-10, eliminates the inequalities of sparse patterns that are combined using the *mergeAuxVars* algorithm and merged the reduced system using the *mergeSimple* algorithm.

5.4.4. Projection Operator

We define a general projection operator, $Project\left(R, \{v_1, \ldots, v_n\} \right)$, that projects away a set of variables $v_1, \ldots, v_n$ from a given system of inequalities using Fourier-Motzkin elimination. The resulting system does not have any inequalities with the variables $v_1, \ldots, v_n$.

The projection operator defined in Definition 4-3 is implemented using this operator. The operator, *proj*, takes a system of inequalities R and eliminates the k-th index variable, i_k, from the system. The range of the k-th index variable is between the lower and the upper bound affine expressions l and u, respectively. Thus, the resulting system of $proj(R, k, l, u)$ is given by $Project\left(Intersect\left(R, \{l \le i_k \le u\} \right), \{i_k\} \right)$. The resulting system does not have any inequalities with the variable i_k. However, the result may not be exact since the index variable may have contributed to a sparse pattern. In that case, elimination of the variable creates a dense region, including the array indices not present in the original region. Therefore, we include the original inequalities with the index variable back into the result by changing the index variable to a new auxiliary variable. If there are no sparse patterns, the clean-up algorithm, given in Section 5.4.8, will eliminate the inequalities with the auxiliary variable.

```
$MrgeAuxVars(R_1, \alpha_1, R_2, \alpha_2) \rightarrow \langle R, \{true, false\}\rangle$
where
        $R_1 = \left\{ \begin{array}{l} -l_1 - c_1^1 i_1 - \ldots - c_1^k i_k + n_1\alpha_1 \geq 0 \\ u_1 + c_1^1 i_1 + \ldots + c_1^k i_k - n_1\alpha_1 \geq 0 \end{array} \right\},$

        $R_2 = \left\{ \begin{array}{l} -l_2 - c_2^1 i_1 - \ldots - c_2^k i_k + n_2\alpha_2 \geq 0 \\ u_2 + c_2^1 i_1 + \ldots + c_2^k i_k - n_2\alpha_2 \geq 0 \end{array} \right\},$

        all $c$'s, $l_1, l_2, u_1, u_2, n_1, n_2$ are integers and $i_1, \ldots, i_k$ are variables

    if $c_1^1 = c_2^1$ .... $c_1^k = c_2^k$ and $n_1 = n_2$ then
            $l = min(l_1, l_2)$
            $u = max(u_1, u_2)$
    else if $c_1^1 = c_2^1$ .... $c_1^k = c_2^k$ and $l_1 \leq l_2$ and $u_1 \geq u_2$ and $\exists k$ s.t. $kn_1 = n_2$ then
            $l = l_1$
            $u = u_1$
    else if $c_1^1 = c_2^1$ .... $c_1^k = c_2^k$ and $l_2 \leq l_1$ and $u_2 \geq u_1$ and $\exists k$ s.t. $kn_2 = n_1$ then
            $l = l_2$
            $u = u_2$
    else
            return $\langle \emptyset, false \rangle$
    $R = \left\{ \begin{array}{l} -l - c_1^1 i_1 - \ldots - c_1^k i_k + n_1\alpha \geq 0 \\ u + c_1^1 i_1 + \ldots + c_1^k i_k - n_1\alpha \geq 0 \end{array} \right\}$ where $\alpha$ is a new auxiliary variable

    return $\langle R, true \rangle$
```

Figure 5-9. Attempts to merge different sparse patterns into a single sparse pattern using a new auxiliary variable.

$Merge(R_1, R_2) \rightarrow \langle R, \{true, false\} \rangle$
where R_1, R_2 and R are convex array sections.

```
$R' = \{ \}$
**for** each auxiliary variable $\alpha_1$ used by the inequalities of $R_1$ **do**
    $R_1^{\alpha} = \{I | I \in R_1 and$ variable $\alpha_1$ is in $I\}$
    **for** each auxiliary variable $\alpha_2$ used by the inequalities of $R_2$ **do**
        $R_2^{\alpha} = \{I | I \in R_2 and$ variable $\alpha_2$ is in $I\}$
        $\langle R^{\alpha}, bool \rangle = mergeAuxVars\left( R_1^{\alpha}, \alpha_1, R_2^{\alpha}, \alpha_2 \right)$
        **if** $bool = true$ **then**
            $R_1 = \{I | I \in R_1 and$ variable $\alpha_1$ is not in $I\}$
            $R_2 = \{I | I \in R_2 and$ variable $\alpha_2$ is not in $I\}$
            $R' = Intersect\left( R', R^{\alpha} \right)$

$\langle R, bool \rangle = mergeSimple(R_1, R_2)$
**if** $bool = true$ **then**
    **return** $\langle Intersect(R, R'), true \rangle$
**else**
    **return** $\langle \emptyset, false \rangle$
```

Figure 5-10. Attempts to merge two convex array sections

5.4.5. Containment Test

The *IsContained* operator checks if all indices of one convex array section are included in the other convex array section. For two convex array sections, R_1 and R_2, the operator $IsContained(R_1, R_2)$ returns true if and only if $R_1 \subseteq R_2$. The implementation of the containment test is given in Figure 5-11.

```
$IsContained(R_1, R_2) \rightarrow \{true, false\}$
where $R_1$ and $R_2$ are convex array sections
        **if** $IsEmpty(R_1)$ **then**
            **return** $true$
        **if** $IsEmpty(Intersect(R_1, R_2))$ **then**
            **return** $false$
        **for** each inequality $r \in R_2$ **do**
            **if** $IsEmpty(Intersect(R_1, \neg r))$ **then**
                **return** $false$
        **return** $true$
```

Figure 5-11. Algorithm for the containment test

5.4.6. Equivalence Test

The $IsEquivalent$ operator returns true if the two convex array sections are equivalent. However, for two convex array sections to be equivalent, the systems do not have to be identical. Thus, the implementation of the equivalence test, given in Figure5-12, identifies equivalent array sections by examining the parameterized array indices of both sections.

```
$IsEquivalent(R_1, R_2) \rightarrow \{true, false\}$
where $R_1$ and $R_2$ are convex array sections
        return $(IsContained(R_1, R_2))$ $and$ $(IsContained(R_2, R_1))$
```

Figure 5-12. Algorithm for the equivalence test

5.4.7. Subtraction Operator

Subtraction of two convex array sections creates an array section containing the indices of the first section that are not present in the second section. Precise subtraction of two convex array sections can result in a non-convex section, as shown in Figure5-13. Thus, we define

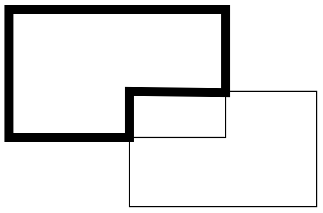


Figure 5-13. An example of a subtraction of two convex sections resulting in a single non-convex section

the subtraction operator, $Subtract(R_1, R_2)$, which is precise if the result can be represented by a single convex array section. When no precise single system exists, we create an approximate result that satisfies the property $R_1 - R_2 \subseteq Subtract(R_1, R_2) \subseteq R_1$. The algorithm for the subtraction operator is given in Figure5-14. The result of $Subtract(R_1, R_2)$ is the empty set if all indices of R_1 are also in R_2. Otherwise, we find a single inequality in R_2 that slices the part of R_1 that is contained in R_2. If there is more than a single inequality that slices the part of R_1 that is contained in R_2, then the result of the subtraction is non-convex.

```
$Substract(R_1, R_2) \rightarrow R$
where $R_1$, $R_2$ and $R$ are convex array sections
        **if** $IsEmpty(Intersect(R_1, R_2))$ **then**
            **return** $R_1$
        **else if** $IsContained(R_1, R_2)$ **then**
            **return** $\emptyset$
        **else**
            **for** each inequality $i \in R_2$ **do**
                **if** $IsContained(Intersect(R_1, \{i\}), R_2)$ **then**
                    **return** $Intersect(R_1, \{\neg i\})$
        **return** $R_1$
```

Figure 5-14. Algorithm for subtracting two convex array sections

5.4.8. Simplify and Clean-up

The above definitions of the operations—such as intersection, merge and projection—are fairly simple. Although these operators produce correct results, the resulting convex array sections are not simple and concise. In fact, the resulting convex array sections produced by these operators have many unnecessary inequalities and auxiliary variables. Thus, we use the algorithms, given in the next five sections, to simplify a convex array section. The algorithms are invoked in the order given in Figure 5-15.

```
$SimplifyCleanup(R) \rightarrow R'$
where $R$ and $R'$ are convex array sections.
        $R = ImproveBounds(R)$
        $R = RemoveUnusedAux(R)$
        $R = NormalizeAux(R)$
        $R = RemoveRedundantAux(R)$
        $R = RemoveSimpleRedundant(R)$
        **return** $R$
```

Figure 5-15. The driver for the simplify and clean-up algorithms

5.4.8.1. Simplify coefficients and tighten the bounds

Using the algorithm in Figure 5-16, we simplify the coefficients of each inequality by dividing the coefficients by the greatest common divisor. This also tightens the inequalities to the closest integer solution since the offset is moved to the closest integer.

5.4.8.2. Eliminate unused auxiliary variables

In creating a sparse pattern, the auxiliary variable should have a non-unit coefficient. Furthermore, there should be at least one inequality providing a lower bound for the auxiliary variable and another inequality providing an upper bound. Using the algorithm shown in Figure 5-17, we eliminate auxiliary variables and the associated inequalities if they do not contribute to a sparse pattern.

```
$ImproveBounds(R) \rightarrow R'$
where $R$ and $R'$ are convex array sections

        **for** all inequalities $I \in R$ such that $I = \{c + a_1 i_1 + \ldots + a_k i_k \geq 0\}$ where
        $c, a_1, \ldots, a_n$ are integer constants **do**

                $g = gcd(a_1, \ldots, a_k)$
                $I = \{\left\lfloor \frac{c}{g} \right\rfloor + \frac{a_1}{g} i_1 + \ldots + \frac{a_k}{g} i_k \geq 0\}$
        **return** $R$
```

Figure 5-16. Algorithm for tightening the integer bounds

```
$RemoveUnusedAux(R) \rightarrow R'$
where $R$ and $R'$ are convex array sections

        **for** all auxiliary variables $\alpha$ in $R$ **do**

                **if not** $\exists$ inequalities $I_1, I_2 \in R$ such that $I_1 = \{c_1 + r_1 + n_1\alpha \geq 0\}$
                and $I_2 = \{c_2 + r_2 - n_2\alpha \geq 0\}$ where $r_1$ and $r_2$ are linear expressions
                and $c_1, c_2, n_1, n_2$ are integers such that $n_1, n_2 > 1$ **then**

                        $R = project(R, \alpha)$

        **return** $R$
```

Figure 5-17. Algorithm for eliminating inequalities and auxiliary variables that do not create any sparse patterns

5.4.8.3. Normalize the offsets

We normalize the offsets of the inequalities with auxiliary variables using the algorithm in Figure 5-18. We find each pair of inequalities with the same linear expression that represents the lower bound and upper bound of a sparse pattern and normalize the offsets such that the offset of the upper bound is always between zero and the coefficient of the auxiliary variable. Again, we eliminate the pair of inequalities if they do not contribute to a sparse pattern.

5.4.8.4. Eliminate redundant auxiliary variables

When two or more convex array sections are combined using operations such as union and intersection, the same sparse pattern that occurred in multiple input sections is repeated in the result using multiple auxiliary variables. The algorithm in Figure5-19 removes these redundant inequalities and auxiliary variables.

```
$RemoveRedundantAux(R) \to R'$
where $R$ and $R'$ are convex array sections
    for all auxiliary variables $\alpha$ in $R$ do
        for all auxiliary variables $\beta$ in $R$ such that $\alpha \neq \beta$ do
            if $IsEmpty(Intersect(R, \{\alpha > \beta\}))$ and
             $IsEmpty(Intersect(R, \{\alpha < \beta\}))$ then
                $R = project(R, \beta)$
    return $R$
```

Figure 5-19. Algorithm for removing redundant auxiliary variables

```
$NormalizeAux(R) \to R'$
where $R$ and $R'$ are convex array sections

    **for** all auxiliary variables $\alpha$ in $R$ **do**
        $n$ is an integer such that there exist $\{t - n\alpha \geq 0\} \in R$ where $t$ is
        an affine expression
        **if not** there exist $\{t' - m\alpha \geq 0\} \in R$ such that $m \neq n$ and $t'$ is
        an affine expression **then**
            $ol = ou = -\infty$
            **for** all inequalities $\{c + r - n\alpha \geq 0\} \in R$ where integer $c > 0$,
            and $r$ is a linear expression **do**
                $ol = max\left(ol, n\left\lfloor \frac{c}{n} \right\rfloor\right)$
            **for** all inequalities $\{-c' + r' + n\alpha \geq 0\} \in R$ where integer
            $c' > 0$ and $r$ is a linear expression **do**
                $ou = max\left(ou, n\left\lfloor \frac{c'}{n} \right\rfloor\right)$
            **if** $ou = ol$ **then**
                **for** all inequalities $\{\pm c'' + r'' - n\alpha \geq 0\} \in R$ where
                integer $c'' \geq 0$ and $r''$ is a linear expression **do**
                    $c'' = c'' - ol$
    **return** $R$
```

Figure 5-18. Algorithm for normalizing the offsets of the inequalities with auxiliary variables

5.4.8.5. Eliminate redundant inequalities

Finally, we remove many redundant inequalities using the algorithm given in Figure5-20. However, the system may still retain redundant inequalities after this algorithm. These redundant inequalities can be found only by using the Fourier-Motzkin elimination technique. Since Fourier-Motzkin elimination is expensive, we do not use it in the clean-up code that gets called frequently.

```
$RemoveSimpleRedundant(R) \to R'$
where $R$ and $R'$ are convex array sections
        for all inequalities $\{c + r \geq 0\} \in R$ where $c$ is an integer and
        $r$ is a linear expression do
                for all inequalities $\{d + r \geq 0\} \in R$ where integer $d > c$ do
                        Remove the inequality $\{d + r \geq 0\}$ from $R$
        return $R$
```

Figure 5-20. Algorithm for removing inequalities that are obviously redundant

5.5. Operations on Array Section Descriptors

In this section, we define the operators that are used to manipulate array section descriptors. These operators are used in the array data-flow analysis algorithm given in Section 4.3. Unlike convex array sections, array section descriptors can represent non-convex array index sets. The operators on array section descriptors are built using the operators defined for convex array sections presented in the previous section. The operators *IsEmpty, Intersect, Union*, and *Project* are exact under the representation of array section descriptors, while our definitions of *Subtract* and *IsContained* produce approximate results. However, our algorithm is able to find the exact results for*Subtract* and *IsCon-*

tained operators for a large class of inputs found in practice. The algorithm in Figure 5-21 is used to insert a new convex array section into the list of convex array sections in an array section descriptor in order to maintain the properties of array section descriptors described in Section 5.2.

$Add(D, R) \rightarrow D'$
where D and D' are array section descriptors and R is a convex array section.

```
if not $IsEmpty(R)$ then
    for each convex array section $R' \in D$ do
        if $IsContained(R', R)$ then
            Remove $R'$ from $D$
        if $IsContained(R, R')$ then
            return $D$
    Insert $R$ into the list of convex array sections in $D$
return $D$
```

Figure 5-21. Algorithm for inserting a convex array section to an array section descriptor

5.5.1. Empty Test

The boolean function $IsEmpty$ determines if an array section descriptor has any valid indices. The empty test returns *true* when the list of convex array sections in the array section descriptor is empty.

5.5.2. Intersection Operator

The intersection operator obtains the common array indices in two array section descriptors for a given array. Figure 5-22 illustrates the implementation of the intersection operator.

```
Intersect $(D_1, D_2) \rightarrow D$
where $D_1$, $D_2$ and $D$ are array section descriptors.
        $D = \{ \}$
        **for** each convex array section $R_1 \in D_1$ **do**
            **for** each convex array section $R_2 \in D_2$ **do**
                $D = Add(D, Intersect(R_1, R_2))$
        **return** $D$
```

Figure 5-22. Algorithm for the intersection operator

5.5.3. Union Operator

The union of two array section descriptors contains all the array indices of both sections. The algorithm for the union operator is given in Figure5-23. The array section descriptor produced by this algorithm will have multiple convex array sections that may be merged into a single section. Merging convex array sections has two advantages. First, merging reduces the number of convex array sections in an array section descriptor, thus reducing the complexity and the storage requirements. This was found to be necessary in practice. Second, merging increases the precision of the *IsContained* and *IsEquivalent* operators. Merging is performed after the simple algorithm for the union operator using a post-pass, given in Figure 5-24, which iterates over the convex array sections in the array section

$Union(D_1, D_2) \rightarrow D$

where D_1, D_2 and D are array section descriptors.

```
$D = D_1$
**for** each convex array section $R_2 \in D_2$ **do**
    $D = Add(D, R_2)$
$D = Merge(D)$
**return** $D$
```

Figure 5-23. Algorithm for the union operator

descriptor until no two convex array sections can be merged. Multiple iterations are needed since merging two convex array sections can enable yet another merge that was not possible before the first merge. In the algorithm, D_N holds the convex regions created in the current iteration, D_C holds the regions created during the previous iteration of the merge, and D_F holds the rest of the regions. Each iteration of the merge first compares all pairs of convex regions in D_C for possible merges and includes any merged region in D_N. Next, each region in D_C is checked against the rest of the regions in D_F for possible merges. This is repeated until no more merging is possible.

5.5.4. Projection Operator

The projection operator projects away an index variable from all the convex array sections in the array section descriptor. Since the projection operator increases the size of each convex region, some of the resulting convex regions may become candidates for merging. The implementation of the projection operator, *proj*, is given in Figure 5-25.

$Merge(D) \rightarrow D'$
where D and D' are array section descriptors.

```
        $D_N = D, \quad D_F = \{ \}$
        while $D_N \neq \{ \}$ do
            $D_C = D_N, \quad D_N = \{ \}$
            Let $D_C = \{R_1, R_2, \ldots, R_m\}$
            for each $R_i$ where $2 \le i \le m$ do
                for each $R_j$ where $1 \le j < i$ do
                    $\langle R, v \rangle = merge(R_i, R_j)$
                    if $v = true$ then
                        Remove $R_i$ and $R_j$ from the list $D_C$
                        Add $R$ to the list $D_N$
            for each $R_C \in D_C$ do
                for each $R_F \in D_F$ do
                    $\langle R, v \rangle = merge(R_C, R_F)$
                    if $v = true$ then
                        Remove $R_C$ from the list $D_C$
                        Remove $R_F$ from the list $D_F$
                        Add $R$ to the list $D_N$
            Add the convex array sections in the list $D_C$ to the list $D_F$
        $D = D_F$
    return $D$
```

Figure 5-24. Post-pass after the union operator

$proj(D, i, l, u) \rightarrow D'$
where D and D' are array section descriptors, i is an index variable and l, u are affine expressions.

```
$D' = \{ \}$
**for** each convex array section $R \in D$ **do**
    $D' = Add(D', Proj(R, i, l, u))$
$D' = merge(D')$
**return** $D'$
```

Figure 5-25. Algorithm for the projection operator

5.5.5. Containment Test

The containment test determines if all the indices of one array section descriptor are contained in the other. However, our implementation of the containment test, given in Figure 5-26, is not precise. The operator is conservative and it may return*false* in some cases when one array section descriptor is fully contained in the other. The difficulty of finding containment is illustrated in Figure 5-27, which shows that the single convex array section of the array section descriptor D_1 is contained by two different convex array sections in the array section descriptor D_2. However, this condition occurs infrequently in practice, since many adjacent convex array sections are merged into a single convex array section whenever possible. Therefore, we have not implemented the more expensive test that checks for containment by multiple convex array sections.

$IsContained(D_1, D_2) \rightarrow \{true, false\}$
where D_1 and D_2 are array section descriptors.

```
for each convex array section $R_1 \in D_1$ do
    $found = false$
    for each convex array section $R_2 \in D_2$ do
        if $IsContained(R_1, R_2)$ then
            $found = true$
            break
    if $found = false$ then
        return $false$
return $true$
```

Figure 5-26. Algorithm for the containment test

5.5.6. Equivalence Test

The equivalence test determines if two array section descriptors contain identical parameterized index sets. The implementation of the equivalent test is given in Figure5-28. Since the equivalence test is implemented using the containment test, it is also not precise. There may be equivalent array section descriptors, as in the example given in Figure5-29, that our implementation will conservatively assume to be different.

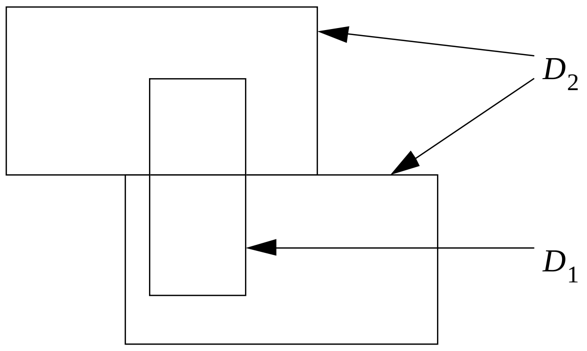


Figure 5-27. Example of the operator $D_1 \subseteq D_2$, where containment is difficult to detect

```
$IsEquivalent(D_1, D_2) \rightarrow \{true, false\}$
where $D_1$ and $D_2$ are array section descriptors.
        **return** $(IsContained(D_1, D_2))\ and\ (IsContained(D_2, D_1))$
```

Figure 5-28. Algorithm for the equivalence test

5.5.7. Subtraction Operator

The subtraction operator creates an array section descriptor with array indices that are present in the first array section descriptor but not in the second. The subtraction operator, as defined by the algorithm in Figure 5-30, is not precise. This is because we use the subtraction operator for convex array sections, which is also not precise, to define the subtraction of array section descriptors. We attempt to subtract the convex array sections multiple

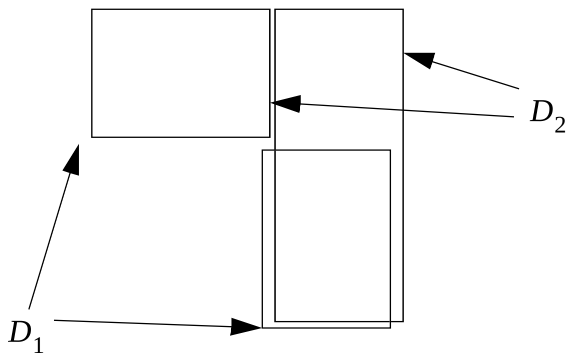


Figure 5-29. Example of equivalent array section descriptors where detection by *IsEquivalent* operator is not possible

times since the result of one subtraction may enable other subtractions. Figure5-31 shows an example where, when subtracting array section descriptors $D_2 = \{R_1, R_2\}$ from D_1, the convex array section R_1 cannot be subtracted from the single convex array section of D_1 until the convex array section R_2 is subtracted from D_1.

5.6. Related Work

Many researchers have used an array index set representation in performing array data-flow analysis [18,64,66,124,137,142]. Accuracy of their analyses is defined by the precision of the summary index set representation. These algorithms use different forms of regular section descriptors as the array index set representation. Each regular section can be used only to precisely represent a limited domain of rectilinear, triangular or diagonal spaces [81]. More complex spaces can be represented using multiple regular sections [142].

The scope of data-flow and data-dependence analysis performed using regular section information is much more restricted than using a representation based on linear inequalities. For example, our data dependence analysis, which uses an array region representation

$Substract(D_1, D_2) \rightarrow D$
where D_1, D_2 and D are array section descriptors.

```
    $D = \{ \}$
    **for** each convex array section $R_1 \in D_1$ **do**
        $iter = true$
        **while** $iter = true$ **do**
            $iter = false$
            **for** each convex array section $R_2 \in D_2$ **do**
                $R = Subtract(R_1, R_2)$
                **if** $R = \phi$ **then**
                    $R_1 = \phi$
                    **break**
                **else if** $R \neq R_1$ **then**
                    $R_1 = R$
                    $iter = true$
                    **break**
        $D = Add(D, R_1)$
    **return** $D$
```

Figure 5-30. Algorithm for the subtraction operator

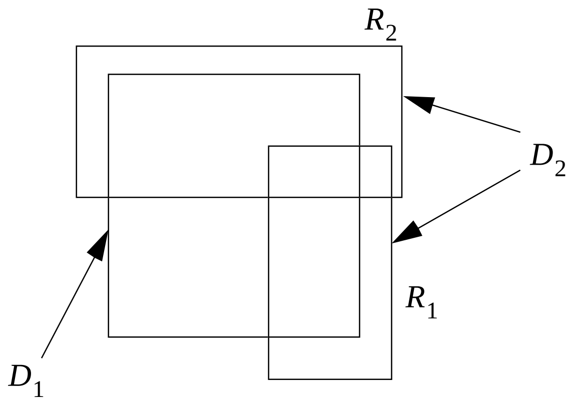


Figure 5-31. Example of a subtraction that needs multiple iterations

based on linear inequalities, is as accurate as the traditional data dependence analysis, which is exact for a pair of array accesses in the domain of loop nests where the loop bounds and array indices are affine functions of the loop indices.

Triolet et al. first proposed using a system of linear inequalities to represent an array index set [137]. This representation was used for data dependences analysis. Their algorithm did not create exact convex regions in many situations, such as sparse access patterns, but provided approximations using a convex hull of all the indices. In the PIPS project, an index set using an integer-lattice was proposed but not implemented due to practical considerations [87]. Our representation for the index sets is most similar to their current representation, which uses a single convex polyhedron as the index set [45,46]. However, there are many access patterns found in practice that cannot be precisely represented by a single convex region. For example, multiple write accesses, described in Figure3-4, can only be precisely represented using a set of convex regions. Thus, even for the array data dependence analysis using the summary information to obtain the same precision as the pair-wise data dependence test [110], a summary based on a single convex region is not sufficient.

5.7. Chapter Summary

In this chapter we introduce an array summary representation based on lists of systems of linear inequalities. Using this representation, we find data-flow information more accurately than any other previous summary representation. We are also able to perform the data-dependence analysis at the same precision as the exact data dependence test [110].

We have defined the set operators used for manipulating array summaries, in this representation. Our intersection, union and projection operators and the empty test are exact. However, the subtraction operator and the containment and equivalence tests we have defined are approximations of the exact result.

6 Array Reshapes Across Procedure Boundaries

The continuing success of FORTRAN as the leading language for programming scientific applications depends heavily upon the ability of FORTRAN programs to outperform programs written in other popular languages. For example, many computationally intensive algorithms coded using FORTRAN can outperform the same algorithms written using C++ by more than a factor of two [78]. Compilers are able to obtain superior performance from FORTRAN programs because many modern language features that hinder compiler optimizations, such as aliasing and dynamic memory allocation, are absent or are severely restricted in the FORTRAN language. A lack of these features makes it possible for compilers to safely perform many aggressive optimizations, such as statement and iteration reorganization, vectorization, and parallelization, on FORTRAN programs.

However, the FORTRAN-77 language standard has three specific features, *parameter reshapes*, *equivalences* and *different common block declarations*, that can suppress many aggressive whole program analyses, needed for finding coarse grain parallelism. It is necessary for an interprocedural compiler to analyze the programs in the presence of these features and determine their effect on the rest of the analysis. Current interprocedural compilers use ad-hoc heuristics and specialized techniques to handle the common cases found in practice. We introduce a systematic approach, based on the linear inequalities framework, to analyze the three classes of reshapes found in FORTRAN programs.

In Section 6.1 we will further describe the three different reshapes found in FORTRAN. Next, we define the array reshape problem in Section6.2 and provide an overview of our solution. Sections 6.3, 6.4 and 6.5 detail the algorithms for solving array reshapes that occur in parameter passing, equivalences and common blocks respectively. We compare our approach to related works in Section 6.6.

6.1. Reshapes in FORTRAN

A reshape occurs when a data structure defined using one shape is also accessed using a different shape within the program. The FORTRAN-77 definition allows three classes of reshapes: parameter reshapes, equivalences, and different common block declarations [150]. Equivalences can affect intraprocedural analysis while the other two affect only interprocedural analysis.

6.1.1. Parameter Reshapes

The FORTRAN-77 definition does not restrict the actual parameters of the caller and the formal parameter of the corresponding callee to be of the same type. This provides the programmer an opportunity to reshape data structures. Figure 6-1 illustrates four examples of reshapes. In Figure 6-1(a), an element of an array in the caller is mapped to a scalar in the callee routine. The real and imaginary parts of a complex variable are mapped to a two-element array in Figure 6-1(b). A simple array reshape is shown in Figure 6-1(c), where a single column of the array Y is mapped to the vector R.

6.1.2. Equivalences

The FORTRAN language, using the equivalence operation, allows the creation of an alias to a scalar or array data structure. The equivalence operation accepts an element of a data structure and an element of the alias structure as input, and aligns the alias structure with the memory layout of the data structure such that the two elements refer to the same memory location. In the example in Figure 6-2, the two-dimensional array B is aliased with the second half of the tenth plain of the three-dimensional array A.

6.1.3. Different Common Block Declarations

Common block structures, used for global variable declaration, provide another opportunity for the programmers to reshape data structures. The common block definition specifies the memory layout of the variables declared in a common block. By not requiring that mul-

```
REAL*8 W(100)
CALL TESTA(W(10))
...

SUBROUTINE TESTA(P)
REAL*8 P
P = ...
```

(a) An array element mapped to a scalar

```
COMPLEX*16 X
CALL TESTB(X)
...

SUBROUTINE TESTB(Q)
REAL*16 Q(2)
....
```

(b) A scalar is mapped to an array

```
INTEGER Y(100,100)
CALL TESTB(Y(10))
...

SUBROUTINE TESTC(R)
INTEGER R(100)
....
```

(c) A slice of an array is mapped to a vector

Figure 6-1. Examples of parameter reshapes

```
REAL*8 A(100,100,100)
REAL*8 B(100,50)
EQUIVALENCE A(10,50,1), B(1,1)
....
```

Figure 6-2. Aliasing using the equivalence operator

tiple definitions of the same common block be identical, the FORTRAN language allows reshaping and overlapping of data structures between different procedures. The example in Figure 6-3 is extracted from the program hydro2d in the SPEC92fp benchmark suite [143]. The common block var1 in the example has two different definitions, one with four two-dimensional arrays of 102×4 elements and the other with a single large vector. Thus, the array element $EN(a, b)$ in procedure INIVAL is the same element accessed by $H1(102b + a + 306)$ in procedure ASW02.

```
PROGRAM ASW02
PARAMETER(MP=102, NP=4)
COMMON /VAR1/ H1(4*MP*NP)
....

SUBROUTINE INIVAL
COMMON /VAR1/ RO(MP,NP), EN(MP,NP), GZ(MP,NP), GR(MP,NP)
....
```

Figure 6-3. Example of a common block reshape from hydro2d

6.2. The Array Reshape Problem

In the interprocedural data-flow analysis algorithm described in Section4.3, we need to propagate the array summary information across procedure boundaries. Propagating an array summary across procedure boundaries requires us to map the summary describing an index set of the formal array to one that defines the corresponding index set for the actual array. Since the FORTRAN-77 language allows the formal and actual array variables to have different dimension sizes, this mapping is not a trivial renaming operation.

Since FORTRAN implements both formal and actual array structures by mapping them to the same linear memory segment, one solution is to perform array data-flow analysis using linearized array accesses [31]. Multi-dimensional array accesses are linearized by converting them to linear offsets of the memory locations. All the linearized arrays have the same shape, thus eliminating any reshape problem. However, the regions in multi-dimensional arrays have to be represented as very complex lattice patterns in a one-dimensional linearized space. Thus, linearizing all the accesses is not a practical solution.

Another solution is to include information describing the relationship between the elements of the formal and the actual arrays in the index set of the formal array. Adding an equality that equates the linearized expressions of the access functions of both shapes to the index set of the formal array is sufficient to make it a valid index set for the actual array. However, the array section created is a complex set of inequalities even when it represents a simple region. Many parameter reshapes that are found in practice map between simple regions. The index set of the actual array, with an equality of linearized access functions, will not directly describe these simple regions. We will demonstrate this using two parameter reshapes in the program turb3d (Figure 6-4) described previously in Section 3.3.3. The elements of the array X that are read by the call are graphically shown in Figure6-5. The relationship between the array X and the array U, after the call to DCFT, is given by the equality $X_1 - 1 = ((U_3 - K)64 + U_2 - 1)64 + U_1 - 1$, where X_1, U_1, U_2, U_3 are dimension variables. Adding this equality to the index sets of the array X will create,

$$\left\{ (U_1, U_2, U_3) \;\middle|\; \exists X_1 \quad \begin{matrix} 1 \le X_1 \le 4224 \\ X_1 - 1 = ((U_3 - K)64 + U_2 - 1)64 + U_1 - 1 \end{matrix} \right\},$$

```
DIMENSION U(66,64,64)
...
DO K=1,64
     CALL DCFT(U(1,1,K),33)
...

SUBROUTINE DCFT(X, INCX)
REAL*8 X(*)
DO I=1,33
     DO II=1,64
          ... = X((I-1)*2+(II-1)*2*INCX+1)
          ... = X((I-1)*2+(II-1)*2*INCX+2)
...
```

Figure 6-4. Example from turb3d with two array reshapes

a valid index set for the array U. However, not directly visible from the index set is the fact that the elements accessed by the array X in DCFT are mapped to a simple plane in the first two dimensions of the array U. We need a sufficiently powerful analysis technique to identify these simple mappings and continue analyzing the caller without these complex equalities which will result in conservative approximations.

Instead of relying on a few special common cases to pattern match and find the simple reshapes, we have developed a general algorithm based on systems of linear inequalities. When the reshape can be described within the affine framework this algorithm is capable of transforming array summaries between different shapes of an array and identifying the simple regions [72,75,76]. We use this algorithm to implement the map operators $\Uparrow$ and $\Downarrow$ defined in Chapter 4.

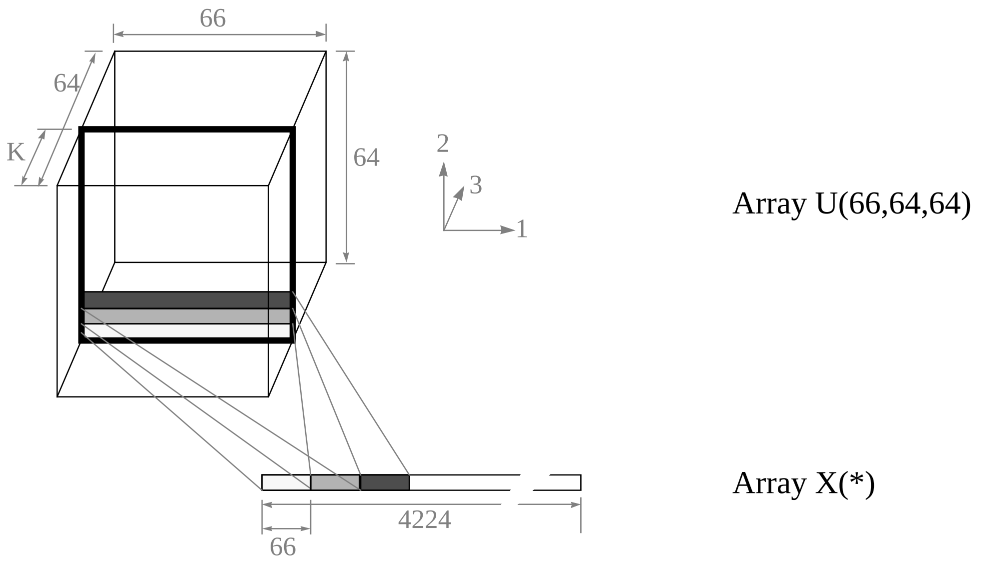


Figure 6-5. The array reshape in turb3d

6.2.1. Algorithm Overview

The array summary reshape algorithm creates a system of inequalities for each reshape problem. The system consists of the convex array region in the original shape of the array, an equality that equates the linearized expressions of the access functions of both shapes and inequalities describing the two array shapes. We then use projection to eliminate the dimension variables in the original array. When there is a simple mapping, we can extract that simple mapping information from the system because it will be given by the integer solution to the system. This key property of integer systems is illustrated using the following simple system. In the following system of inequalities,

$$\{ (i, j, k) \mid 100i = 100j + k, 0 \le k < 100 \}$$

there are many real solutions for i, j and k. But there is only a single integer solution, $i = j$ and $k = 0$, which can be found by integer programming [127]. This property of integer systems allows us to precisely extract many of the simple reshape regions that occur in practice.

By using this algorithm on the reshape in Figure 6-5, we can determine that the result of the reshape is a simple plane of the array U. The original array region, given in Figure 6-6(a), is the convex array section that describes the elements of the array X read by the first call to DCFT. The special system of inequalities of the reshape problem, given in Figure 6-6(b), includes the array section of the original shape, bounds on the dimensions, and the equality of the linearized access functions. By eliminating the dimension variable X_1, the integer solver finds that the only solution for U_1, U_2 and U_3 is a plane in the first two dimensions of the array U. Thus, we are able to find the convex array region of U with the simple region description as shown in the Figure 6-6(c).

6.3. Array Reshapes due to Parameter Passing

We define an array reshape caused by parameter passing as any mapping between an array access as the actual parameter and an array as the corresponding formal parameter. We assume that the elements of both arrays are of the same type.

Definition 6-1: *An array is reshaped in parameter passing when an n-dimensional array A, declared as* $A\left(l_1^A : u_1^A, \ldots, l_n^A : u_n^A\right)$ *in the caller space, is passed as an actual parameter of a procedure call, using the access* $A(f_1, \ldots, f_n)$, *to the m-dimensional array B, declared as* $B\left(l_1^B : u_1^B, \ldots, l_m^B : u_m^B\right)$ *in the callee space, where* $l_1^A, \ldots, l_n^A$, $l_1^B, \ldots, l_m^B$, $u_1^A, \ldots, u_n^A$ *and* $u_1^B, \ldots, u_m^B$ *integers and* $f_1, \ldots, f_n$ *are affine expressions.*

Figure 6-7 shows a code segment representing the array reshape given by the above definition. When the entire array is passed as an actual parameter, the access function becomes the lower bound. Hence, the actual parameter is equivalent to the array access $A\left(l_1^A, \ldots, l_n^A\right)$.

$$\{\ (X_1)\ |\ 1 \le X_1 \le 4224\}$$

(a) Convex array region in the original shape

$$\begin{array}{cc} 1 \le X_1 \le 4224 & 1 \le U_1 \le 66 \\ 1 \le K \le 64 & 1 \le U_2 \le 64 \\ & 1 \le U_2 \le 64 \end{array}$$

$$X_1 - 1 = ((U_3 - K)\, 64 + U_2 - 1)\, 66 + U_1 - 1$$

(b) System of inequalities before projection

$$\left\{ (U_1, U_2, U_3)\ \middle|\ \begin{array}{c} 1 \le U_1 \le 66 \\ 1 \le U_2 \le 64 \\ U_3 = K \end{array} \right\}$$

(c) After projection, convex array region in the new shape

Figure 6-6. Calculating an array summary across an array reshape

By using this definition of an array reshape between an actual and a formal parameter at a procedure call, we can formally describe our algorithm for mapping an index set of the actual array to the corresponding index set of the formal array.

```
DIMENSION A(l1A:u1A,...,lnA:unA)
INTEGER f1,...,fn
...
f1 = ...
...
fn = ...
CALL FOO(...,A(f1,...,fn),...)
...

SUBROUTINE FOO(...,B,...)
DIMENSION B(l1B:u1B,...,lmB:umB)
...
```

Figure 6-7. Code segment representing the reshape in the Definition 6-1.

Theorem 6-1: *Given an array section descriptor D_B of the formal array B at a procedure call according to Definition 6-1, the corresponding array section descriptor at the call site after the array summary reshape is given by*

$$reshape(D_B) = Project\left(Intersect(D_B, \{R\}), \{b_1, \ldots, b_m\} \right) \text{ where}$$

$$R = \left\{ \begin{array}{cc} l_1^A \le a_1 \le u_1^A & l_1^B \le b_1 \le u_1^B \\ \ldots & \ldots \\ l_n^A \le a_n \le u_n^A & l_m^B \le b_m \le u_m^B \\ \multicolumn{2}{c}{\sum_{i=1}^{n} \left((a_i - f_i) \prod_{j=1}^{i-1} \left(u_j^A - l_j^A + 1 \right) \right) = \sum_{i=1}^{m} \left(\left(b_i - l_i^B \right) \prod_{j=1}^{i-1} \left(u_j^B - l_j^B + 1 \right) \right)} \end{array} \right\}$$

and $a_1, \ldots, a_n$, $b_1, \ldots, b_m$ are the dimension variables of the arrays A and B respectively.

In some reshapes found in practice, the lower and upper bounds of many dimensions of the actual and formal arrays are the same. If the lower and upper bounds are the same for inner dimensions and the entire array is passed as the actual parameter, we can reduce the complexity of the projection operation by making the equality of linearized access functions simpler. Theorem 6-2 redefines array summary reshapes due to parameter passing, when the first difference of lower and upper bounds between the actual and the formal is at the *k*-th dimension.

Theorem 6-2: *Given an array section descriptor* D_B *of the formal array B at a procedure call according to Definition 6-1, and* $\forall_{1 \le i \le k-1}\ l_i^A = l_i^B$ and $u_i^A = u_i^B$, *and the array access function used as the actual parameter is* $A\left(l_1^A, \ldots, l_n^A\right)$, *the corresponding array section descriptor at the call site after the array summary reshape is given by*
$reshape(D_B) = Project\left(Intersect(D_B, \{R\}), \{b_1, \ldots, b_m\}\right)$ *where*

$$R = \left\{ \begin{array}{ccc} a_1 = b_1 & u_k^A \le a_k \le u_k^A & u_k^B \le b_k \le u_k^B \\ \ldots & \ldots & \ldots \\ a_{k-1} = b_{k-1} & u_n^A \le a_n \le u_n^A & u_m^B \le b_m \le u_m^B \\ \multicolumn{3}{c}{\sum_{i=k}^{n}\left(\left(a_i - l_i^A\right)\prod_{j=k}^{i-1}\left(u_j^A - l_j^A + 1\right)\right) = \sum_{i=k}^{m}\left(\left(b_i - l_i^B\right)\prod_{j=k}^{i-1}\left(u_j^B - l_j^B + 1\right)\right)} \end{array} \right\},$$

and $a_1, \ldots, a_n$, $b_1, \ldots, b_m$ *are the dimension variables of the arrays A and B respectively.*

6.4. Array Reshapes in Equivalences

An array reshape occurs with an equivalence when both structures in an equivalence statement are arrays.

Definition 6-2: *An array reshape occurs in an equivalence operation when the two accesses given to the equivalence operator are* $A\left(c_1^A, \ldots, c_n^A\right)$ *and* $B\left(c_1^B, \ldots, c_m^B\right)$ *where A is an n-dimensional array declared as* $A\left(l_1^A : u_1^A, \ldots, l_n^A : u_n^A\right)$ *and B is an m-dimensional array declared as* $B\left(l_1^B : u_1^B, \ldots, l_m^B : u_m^B\right)$, *and* $l_1^A, \ldots, l_n^A$, $l_1^B, \ldots, l_m^B$, $u_1^A, \ldots, u_n^A$, $u_1^B, \ldots, u_m^B$, $c_1^A, \ldots, c_n^A$ *and* $c_1^B, \ldots, c_m^B$ *are integers.*

The Figure 6-7 shows a code segment representing the equivalence given by the above definition.When performing array data-flow analysis on a program with an array equivalence, we need to map the array region information of the alias structure to that of the original structure. The algorithm for this mapping is very similar to the one for aliasing due to parameter reshapes. In this algorithm, we assume that the elements of both arrays are of the same type.

```
DIMENSION A(l1^A:u1^A,...,ln^A:un^A)
DIMENSION B(l1^B:u1^B,...,lm^B:um^B)
EQUIVALENCE A(c1^A,...,cn^A), B(c1^B,...,cm^B)
...
```

Figure 6-8. Code segment representing the equivalence in the Definition 6-2.

Theorem 6-3: *Given an array section descriptor D_B of an aliased array of an equivalence operation defined in Definition 6-2, the corresponding array section descriptor of the original array is given by*

$$reshape(D_B) = Project\left(Intersect(D_B, \{R\}), \{b_1, \dots, b_m\} \right) \text{ where}$$

$$R = \left\{ \begin{array}{cc} l_1^A \le a_1 \le u_1^A & l_1^B \le b_1 \le u_1^B \\ \dots & \dots \\ l_n^A \le a_n \le u_n^A & l_m^B \le b_m \le u_m^B \\ \multicolumn{2}{c}{\sum_{i=1}^{n}\left(\left(a_i - c_i^A\right)\prod_{j=1}^{i-1}\left(u_j^A - l_j^A + 1\right)\right) = \sum_{i=1}^{m}\left(\left(b_i - c_i^B\right)\prod_{j=1}^{i-1}\left(u_j^B - l_j^B + 1\right)\right)} \end{array} \right\}$$

and $a_1, \dots, a_n$, $b_1, \dots, b_m$ are the dimension variables of the arrays A and B respectively.

The optimization of the parameter reshape algorithm, given in Theorem 6-2, can also be applied to the equivalence reshape algorithm.

6.5. Array Reshapes in Common Blocks

We need to perform an array summary reshape when mapping a summary of an array declared in a common block of the caller to a callee where the common block has a different definition. We extend our array summary reshape algorithm to handle common block reshapes. We start by defining the shape of a common block.

Definition 6-3: *A shape S_C of a common block C has m arrays that are contiguous in memory where the k-th array, A_k, is an n_k-dimensional array of e_k-byte elements declared as $A_k\left(l_1^k : u_1^k, \ldots, l_{n_k}^k : u_{n_k}^k\right)$.*

The common blocks can have scalar variables and, in this analysis, we treat them as one-dimensional arrays with one element. Next, we define the offset to the starting memory location of the *k*-th array defined in a common block. Since all arrays of a common block are laid out in contiguous memory in the order they are declared, the start offset of an array is calculated by summing the amount of memory allocated to all the previous arrays. The dimension sizes of the arrays are known at compile-time; thus the starting offset is a compile-time constant. In the following discussion, we use the notation given in Definition6-3 without further description.

Definition 6-4: *The starting offset, in bytes, for the k-th array,A_k, of the shape S_C of the common block C is*

$$start(S_C, A_k) = \sum_{i=1}^{k-1}\left(e_i \prod_{j=1}^{n_i}\left(u_j^i - l_j^i + 1\right)\right).$$

By checking the memory locations allocated to each array for any overlap, we can determine if arrays in two common block definitions share elements. Since the offset is a compile-time constant, this can be determined at compile time.

Definition 6-5: *The two arrays A_k and $A'_{k'}$, declared in the respective shapes S_C and S'_C of the common block C, have common elements ($common(A_k, A'_{k'})$) iff $start(S'_C, A'_{k'}) < start(S_C, A_{k+1})$ and $start(S_C, A_k) < start(S'_C, A'_{k'+1})$, where A_k is the k-th array of S_C and $A_{k'}$ is the k'-th array of S'_C.*

If two arrays in different shapes of the common block share common elements, we can find the mapping between these elements by extending the algorithm developed for array summary reshapes. We provide an exact mapping only when the elements of both arrays are of the same type and the elements are aligned with each other. Note that the descriptor are not mapped to a single array, because the array may be overlapped with multiple arrays in the other shape.

Theorem 6-4: *For the two arrays* A_k *and* $A'_{k'}$ *such that* $common\,(A_k, A'_{k'}) = true$, $e_k = e'_{k'}$, *and* $start\,(S_C, A_k) \equiv start\,(S'_C, A'_{k'}) \pmod{e_k}$, *the elements of the array section descriptor* $D_{A'_{k'}}$ *of the array* A_k *that are common to the array* $A'_{k'}$ *are given by* $map\left(D_{A'_{k'}}, A'_{k'}, A_k\right) = Project\left(Intersect\left(D_{A'_{k'}}, \{R\}\right), \{a'_1, a'_2, \ldots, a'_{n'}\}\right)$ *where,*

$$R = \left\{ \begin{array}{ccc} u_1^k \le a_1 \le u_1^k & & u'^{k'}_1 \le a'_1 \le u'^{k'}_1 \\ \ldots & x = x' & \ldots \\ u^k_{n_k} \le a_{n_k} \le u^k_{n_k} & & u'^{k'}_{n'_{k'}} \le a'_{n'_{k'}} \le u'^{k'}_{n'_{k'}} \end{array} \right\},$$

$$\left(x = e_k \sum_{i=1}^{n_k} \left(\left(a_i - l_i^k \right) \prod_{j=1}^{i-1} \left(u_j^k - l_j^k + 1 \right) \right) + start\,(S_C, A_k) \right),$$

$$\left(x' = e'_{k'} \sum_{i=1}^{n'_{k'}} \left(\left(a'_i - l'^{k'}_i \right) \prod_{j=1}^{i-1} \left(u'^{k'}_j - l'^{k'}_j + 1 \right) \right) + start\,(S'_C, A'_{k'}) \right),$$

and the dimension variables of the arrays A_k *and* $A'_{k'}$ *are* $a_1^k, \ldots, a^k_{n_k}$ *and* $a'^{k'}_1, \ldots, a'^{k'}_{n'_{k'}}$.

Since arrays in different common block shapes can overlap, an array section descriptor of an array in one shape will be mapped into multiple array section descriptors in the second shape.

Theorem 6-5: *An array section descriptor* $D_{A'_{k'}}$ *of the array* $A'_{k'}$ *of the common block shape* S'_C *of the common block C with respect to the common block shape*S_C *of the same common block is a set of array descriptors* $\{D_{A_i}, \ldots, D_{A_j}\}$, *where for all k such that*

$i \le k \le j$, A_k *is an array of the common block shape* S_C *and* $common\,(A'_{k'}, A_k) = true$ *and*

$$D_{A_k} = \begin{cases} map\left(D_{A'_{k'}}, A'_{k'}, A_k\right) & \begin{array}{c} e_k = e'_{k'} \quad and \\ start\,(S_C, A_k) \equiv start\,(S'_C, A'_{k'}) \pmod{e_k} \end{array} \\ \\ \{A_k\} & otherwise \end{cases}$$

6.6. Related Work

Many previous interprocedural analyzers did not address the array reshape problem. One way to avoid array reshape analysis is to perform inline substitution and generate equivalence statements to describe the reshapes that occur in parameter passing and in common blocks [67]. This approach only shifted the reshapes of parameters and common blocks to the reshapes of equivalences.

Another proposed scheme to eliminate the reshape problem is to linearize all the array accesses [31]. However, performing array analyses using these linearized accesses is more complex than using multi-dimensional arrays. For example, in array data-flow analysis, many simple regions in multi-dimensional arrays get converted into complex lattice patterns in a one-dimensional linearized space.

Simple array reshape analysis is used in a few interprocedural analyzers [137]. Their scope was limited to a class of reshapes where the formal array declaration is identical to the lower dimensions of the actual array. These simple reshapes are performed by including the upper dimension information of the actual array with the renamed array section of the formal array.

We have designed the first algorithm capable of handling many complex reshape patterns that occur in practice. Using integer projections, we are able to handle many array reshapes that occur in parameter passing, equivalences, and different common block declarations.

Recently, a similar parameter reshape algorithm that uses integer programming was proposed by Creusillet and Irigoin [45]. Their algorithm was an extension of our earlier algorithm [75], which did not eliminate lower dimensions, as presented in Theorem 6-2.

6.7. Chapter Summary

In this chapter we introduce a systematic approach for analyzing array reshapes. We present algorithms to handle array reshapes that occur in parameter passing, equivalences and different common block declarations. The algorithms are based on the linear inequalities framework. We create a special system of inequalities and use integer projection to map an array summary of one shape to the corresponding summary of the other shape. These algorithms are capable of detecting many simple reshape patterns found in practice.

7 Experimental Results in Coarse-Grain Parallelism

In this chapter, we evaluate the impact of coarse grain parallelization analysis. We have implemented the interprocedural array analysis described in the previous four chapters as a part of the Stanford SUIF compiler. We show that the SUIF parallelizer is capable of locating large coarse-grain parallel loops in sequential programs without any user intervention. We also provide an empirical evaluation of the compiler system by using it to parallelize more than 115,000 lines of FORTRAN code from 39 programs in four benchmark suites. We evaluate the effectiveness of using interprocedural analysis, including two advanced array analysis techniques: array privatization and array reduction [76].

We present static counts of the parallelizable loops found using each of these techniques. Static loop counts, though, are not good indicators of whether parallelization is successful. Specifically, parallelizing just one outermost loop can have a profound impact on a program's performance. Dynamic measurements provide much more insight into whether a program may benefit from parallelization. Thus, in addition to static measurements on the benchmark suites, we also present a series of results gathered from executing the programs on a parallel machine. We present overall speedup results, and other measurements of some of the factors that determine the speedup. We also provide results that identify the contributions of the analysis components of our system.

7.1. Experimental Setup

Our compiler system automatically parallelizes sequential applications without relying on any user directives. Parallelized programs generated by our compiler are executed on cache-coherent shared address-space multiprocessors.

7.1.1. The Compiler System

Our experimental setup is based on the Stanford SUIF compiler. The compiler takes a sequential FORTRAN program as input, performs a large suite of analyses to parallelize the code, and outputs the results as a SPMD (Single Program Multiple Data) parallel C version of the program that can be compiled by native C compilers on a variety of architectures. The resulting C program is linked to a parallel run-time system that currently runs on several bus-based shared memory architectures (Silicon Graphics Challenge and Power Challenge, and Digital 8400 multiprocessors [57]) and scalable shared-memory architectures (Stanford DASH [105] and Kendall Square KSR-1 [94]).

We have developed an interprocedural parallelizer with advanced array analyses and optimizations, that is capable of detecting coarse-grain parallelism [75,76,71]. The parallelizer is integrated as a part of the SUIF compiler system [144]. Other advanced optimizations such as loop transformations [146], data and computation co-location [13], data transformations (Chapter 8), synchronization elimination [140], compiler-directed page coloring [30], and compiler-inserted prefetching [116] have also been implemented in the SUIF compiler system. Detection of coarse-grain parallelism, in combination with these other optimizations, can achieve significant performance improvements for sequential scientific applications on multiprocessors. The SUIF compiler system has demonstrated this by obtaining the highest known SPEC92fp and SPEC95fp ratios to date [8,73].

However, in this chapter we focus only on the ability of the compiler to detect coarse-grain parallelism. Thus, to obtain parallel executions, we have adopted a very simple parallel code generation strategy that does not include the locality optimizations. The compiler parallelizes only the outermost loop that the analysis has proven to be parallelizable. Our compiler suppresses parallel execution if the overhead involved is expected to overwhelm the benefits. The run-time system estimates the amount of computation in each parallelizable loop using the knowledge of the iteration count at run time, and runs the loop sequentially if it is considered too fine-grained to have any parallelism benefit. The iterations of a parallel loop are evenly divided between the processors at the time the parallel loop is spawned.

7.1.2. Multiprocessors

We evaluate the effectiveness of coarse grain parallelism by executing the parallelized SPMD loop nests using two different bus-based shared-memory multiprocessors. A summary of the Silicon Graphics Challenge and the Digital AlphaServer 8400 multiprocessors is given in Figure 7-1.

The Silicon Graphics Challenge multiprocessor used in the experiments is a bus-based shared-memory multiprocessor containing 8 MIPS R4400 microprocessors, a single-issue superpipelined processor. The floating point unit of the R4400 is not fully pipelined, i.e., it cannot issue a new floating point instruction to the same functional unit every clock cycle. The R4400 has 16 Kbytes of on-chip instruction cache and 16 Kbytes of on-chip data cache. The data interface to the off-chip cache is 128 bits wide and runs at a half or a third of the on-chip clock rate. The multiprocessor interconnect used in the Challenge is called PowerPath-2. PowerPath-2 is a wide, split transaction bus capable of a sustained transfer rate of 1.2 Gigabytes per second. The bus implements a write invalidate cache coherency protocol and has independent 256-bit data bus and 40-bit address bus. The block size used in the Challenge is 128 bytes. The independent data and address buses provide support for split transactions and the PowerPath-2 can have up to eight outstanding read transactions [82].

The Digital AlphaServer 8400 used in the experiments is a bus-based shared-memory multiprocessor containing 8 Digital 21164 Alpha processors. The Digital 21164 Alpha is a quad-issue superscalar microprocessor with two 64-bit integer and two 64-bit floating point pipelines [49]. There are two levels of caches on-chip: 8 KB instruction/ 8 KB data level 1 cache, and 96 KB of combined level 2 cache. The memory system allows multiple outstanding off-chip memory accesses. Each processor has 4 MB of 10ns external cache. The architecture provides 32 integer and 32 floating-point registers. The 256-bit data bus, which operates at 75MHz, supports 265ns memory read latencies and 2.1 GB per second of data bandwidth. Banked memory modules are attached to the bus [57].

7.2. Examples of Coarse-Grain Parallelism

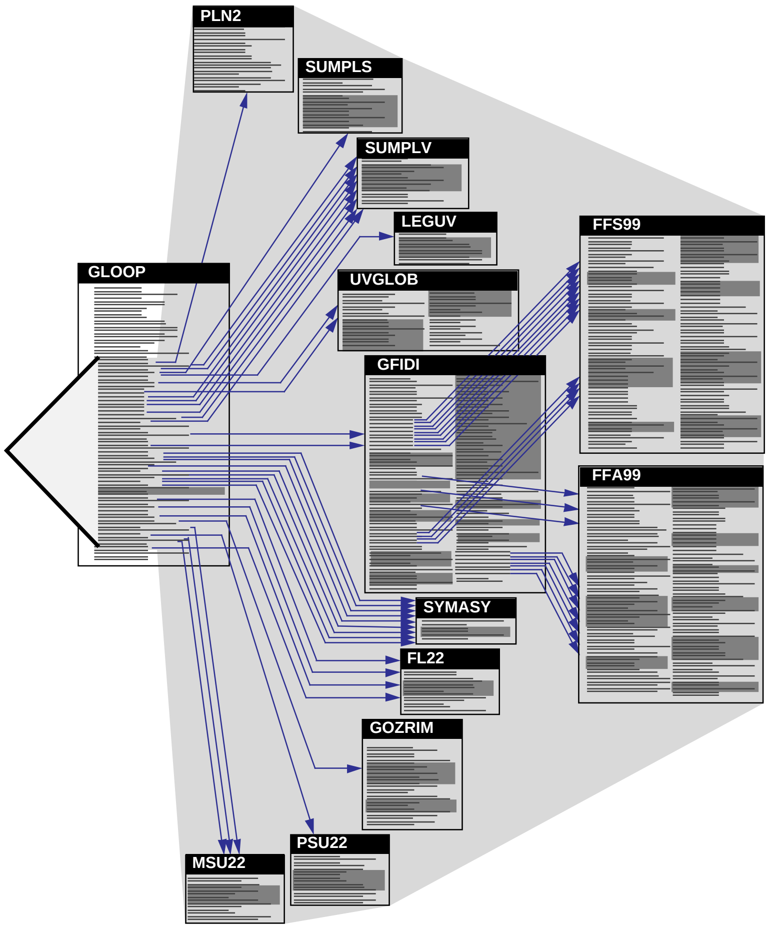
Not only do some of the SUIF-parallelized loops execute for a long time, they can also be very large. The largest loop SUIF parallelizes is from spec77 of the Perfect benchmark

Machine	**Silicon Graphics Challenge**	**Digital AlphaServer 8400**
Number of processors	8	8
Main memory	768 MB	4 GB
System bus bandwidth	1.2 GB/sec	2.1 GB/sec
Operating system	IRIX 5.3	OSF1 V3.2
Processor	MIPS R4400	Digital Alpha 21164
Clock speed	200 MHz	300 MHz
On-chip cache	16 KB Instruction + 16 KB Data	8 KB Instruction + 8 KB Data 96 KB combined second level
External cache	4 MB	4 MB
Uniprocessor SPECfp92	131	513

Figure 7-1. Characteristics of the two multiprocessor systems used for the experiments

suite [97], consisting of 1002 lines of code from the original loop and its invoked procedures. An outline of the loop is shown in Figure7-2. The boxes represent procedures and the lines represent procedure invocations. The outer parallel loop, marked using light gray shading, contains 60 subroutine calls to 13 different procedures. Within this loop, the compiler found 48 interprocedural privatizable arrays, 5 interprocedural reduction arrays and 27 other arrays accessed independently.

Another example of a large coarse-grain parallel loop discovered by the SUIF compiler is in the SPEC95fp program turb3d. The four main computation loops in theturb3d compute a series of three-dimensional FFTs. While these loops are parallelizable, they all have a complex control structure containing large amounts of code, as shown in Figure7-3. Each parallel loop, as indicated in the diagram, consists of over 500 lines of code spanning eight or nine procedures, with up to 42 procedure calls. It is necessary to parallelize these



Procedure
Procedure call
A parallel loop nest found by first generation parallelizers
Single coarse-grain parallel region

Figure 7-2. Parallelizable regions from a code segment in spec77

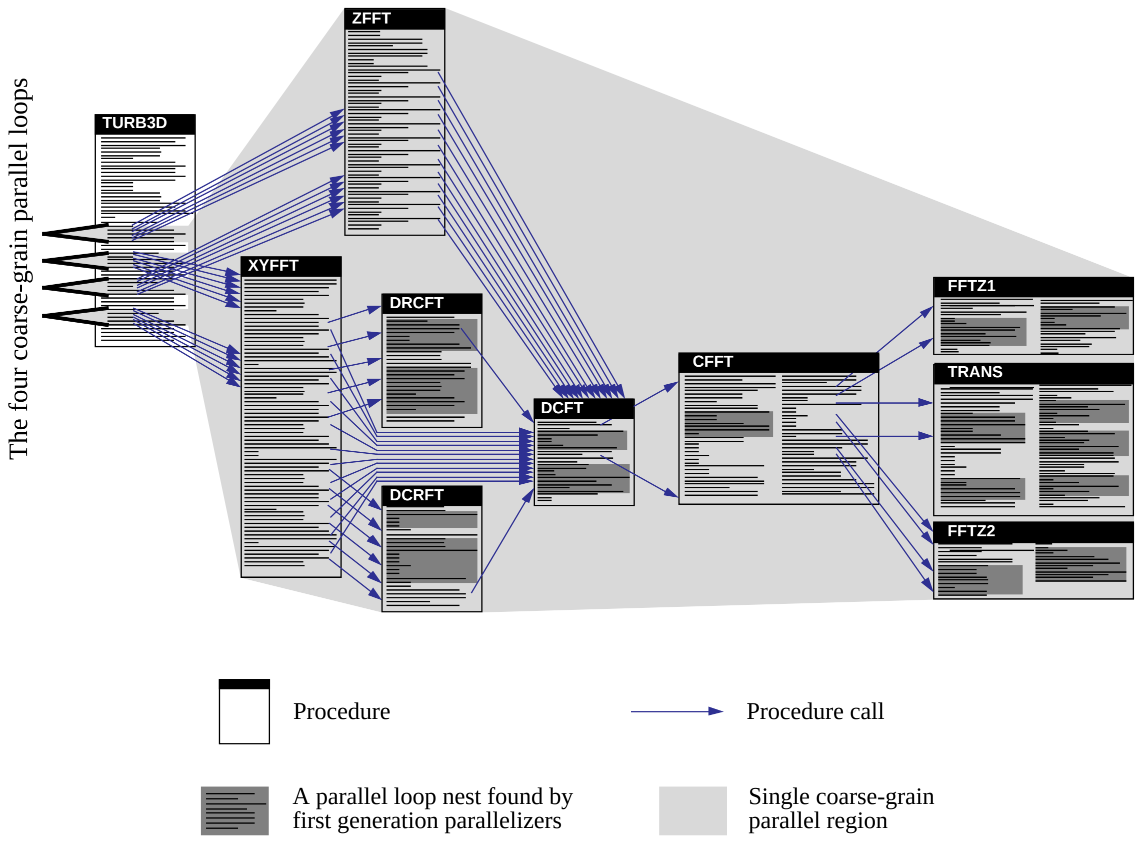

The four coarse-grain parallel loops
TURB3D
ZFFT
XYFFT
DRCFT
DCRFT
DCFT
CFFT
FFTZ1
TRANS
FFTZ2
Procedure
Procedure call
A parallel loop nest found by first generation parallelizers
Single coarse-grain parallel region

Figure 7-3. Parallelizable regions from a code segment in turb3d

outer loops to get any significant speedup. The key to discovering the parallelism is interprocedural array analysis. The compiler is able to determine that iterations of the outer loops operate on independent planes of the arrays across the procedure calls. The analysis is further complicated by the array reshapes found in the program, of which an example is given in Figure 6-4. Once parallelized, turb3d speeds up by over 5.8 times on a 8-processor Digital AlphaServer 8400, as shown in Figure 7-4.

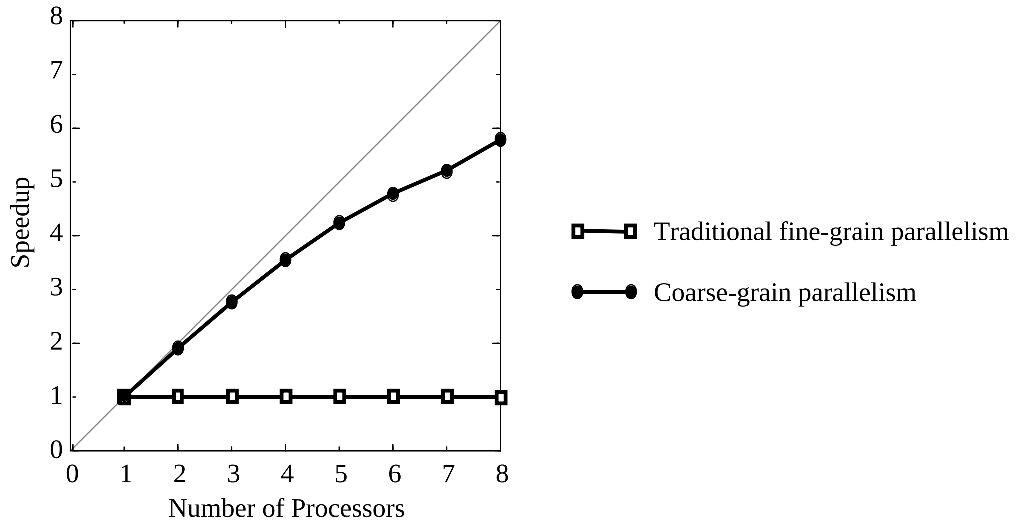


Figure 7-4. Parallel speedup for turb3d on a 8 processor AlphaServer

7.3. Benchmark Programs

To evaluate our parallelization analysis, we measured its success at parallelizing four standard benchmark suites described in Figure 7-5: the Fortran programs from the SPEC95fp and SPEC92fp benchmark suites, the sample Nas benchmarks, and the Perfect Club benchmark suite. We have made a very small number of modifications to the original programs, mainly to fix bugs. These are explicitly stated in the benchmark descriptions.

7.3.1. SPEC95fp Benchmark Suite

SPEC95fp is a set of 10 floating-point programs created by an industry-wide consortium and is currently the industry standard in benchmarking uniprocessor architectures and compilers. In our analysis, we omit fpppp because it contains very little loop-level parallelism and has many type errors in the original Fortran source.

Program	**Length**	**Description**		**Execution time** (seconds)	
				Challenge	AlphaServer
SPEC95fp					
tomcatv	190 lines	mesh generation			314.4
swim	429 lines	shallow water model			282.1
su2cor	2332 lines	quantum physics			202.9
hydro2d	4292 lines	Navier-Stokes			350.1
mgrid	484 lines	multigrid solver			367.3
applu	3868 lines	parabolic/elliptic PDEs			393.4
turb3d	2100 lines	isotropic, homogeneous turbulence			347.7
apsi	7361 lines	mesoscale hydrodynamic model			193.3
wave5	7764 lines	2-D particle simulation			217.4
SPEC92fp					
doduc	5334 lines	Monte Carlo simulation		20.0	4.8
mdljdp2	4316 lines	equations of motion		45.5	19.4
wave5	7628 lines	2-D particle simulation		42.9	12.6
tomcatv	195 lines	mesh generation		19.8	9.2
ora	373 lines	optical ray tracing		89.6	21.5
mdljsp2	3885 lines	equations of motion, single precision		40.5	19.5
swm256	487 lines	shallow water model		129.0	42.6
su2cor	2514 lines	quantum physics		156.1	20.1
hydro2d	4461 lines	Navier-Stokes		110.0	31.6
nasa7	1105 lines	NASA Ames Fortran kernels		143.7	59.0
Nas					
appbt	4457 lines	block tridiagonal PDEs	12^3x5^2 grid	10.0	2.3
			64^3x5^2 grid		3,039.3
applu	3285 lines	parabolic/elliptic PDEs	12^3x5^2 grid	4.6	1.2
			64^3x5^2 grid		2,509.2
appsp	3516 lines	scalar pentadiagonal PDEs	12^3x5^2 grid	7.7	2.2
			64^3x5^2 grid		4,409.0
buk	305 lines	integer bucket sort	65,536 elements	0.6	0.3
			8,388,608 elements		45.7
cgm	855 lines	sparse conjugate gradient	1,400 elements	5.4	2.0
			14,000 elements		93.2
embar	135 lines	random number generator	256 iterations	4.6	1.4
			65,536 iterations		367.4
fftpde	773 lines	3-D FFT PDE	64^3 grid	26.3	6.2
			256^3 grid		385.0
mgrid	676 lines	multigrid solver	32^3 grid	0.6	0.2
			256^3 grid		127.8
Perfect					
adm	6105 lines	pseudospectral air pollution model		20.2	6.4
arc2d	3965 lines	2-D fluid flow solver		185.0	46.4
bdna	3980 lines	molecular dynamics of DNA		63.7	12.4
dyfesm	7608 lines	structural dynamics		18.3	3.8
flo52	1986 lines	transonic inviscid flow		24.1	7.2
mdg	1238 lines	moleclar dynamics of water		194.5	62.1
mg3d	2812 lines	depth migration		410.9	250.7
ocean	4343 lines	2-D ocean simulation		71.8	23.6
qcd	2327 lines	quantum chromodynamics		9.6	3.1
spec77	3889 lines	spectral analysis weather simulation		124.6	20.7
track	3735 lines	missile tracking		6.2	1.8
trfd	485 lines	2-electron integral transform		21.1	5.5

Figure 7-5. Benchmark descriptions, data-set sizes and execution times

7.3.2. SPEC92fp Benchmark Suite

SPEC92fp is a set of 14 floating-point programs from the 1992 version of the SPEC benchmark suite. The programs tomcatv, swm256, su2cor, hydro2d, wave5 and fpppp are the same as SPEC95fp, but with smaller data sets. Because the interprocedural analysis is available only for FORTRAN, we omit alvinn and ear, the two C programs, and spice, a program of mixed Fortran and C code. We also omit fpppp for the same reasons given above.

7.3.3. Nas Parallel Benchmark Suite

Nas is a suite of eight programs used for benchmarking parallel computers. NASA provides sample sequential programs plus application information, with the intention that they can be rewritten to suit different machines. We use all the NASA sample programs except for embar. We substitute for embar a version from Applied Parallel Research(APR) that separates the first call to a function, which initializes static data, from the other calls. We present results for both small and large data set sizes.

7.3.4. Perfect Club Benchmark Suite

Perfect is a set of sequential codes used to benchmark parallelizing compilers. We present results on 12 of 13 programs here. Spice contains pervasive type conflicts and parameter mismatches in the original FORTRAN source that violate the FORTRAN-77 standard. This program is considered to have very little loop-level parallelism. We corrected a few type declarations and parameters passed in arc2d, bdna, dyfesm, mgrid, mdg and spec77.

7.4. Applicability of Advanced Analyses

In this section we present static and dynamic measurements to assess the impact of the array analysis components. We define a *baseline* system that serves as a basis of comparison throughout this section. The baseline refers to our system without any of the advanced array analyses. It performs intraprocedural data dependence, and lacks the capability to privatize arrays or recognize reductions. Note that the baseline system is much more powerful than many existing parallelizing compilers as it contains all the interprocedural scalar analysis

[76]. Our full system, in addition to the analyses in the baseline system, performs array reduction and privatization analysis and carries out all the analyses interprocedurally. We also separately measure the impact of the three components: interprocedural analysis, array reductions, and array privatization.

7.4.1. Static Measurements

The table in Figure 7-6 counts the number of parallel loops found by the SUIF compiler using different combinations of techniques. The first column of the table is the total number of loops in each program. The last column indicates the counts of all parallelizable loops, including those nested within other parallel loops which would consequently not be executed in parallel under our parallelization strategy. Columns 2 through 9 indicate the combinations of techniques needed to parallelize each of these loops. The second column gives the number of loops that are parallelizable in the baseline system. The next three columns measure the applicability of the intraprocedural versions of advanced array analyses. We separately measure the effect of including reduction recognition, privatization, and both reduction recognition and privatization. The next set of four columns includes interprocedural data dependence analysis. Similarly, the eighth and ninth columns measure the effect of adding interprocedural privatization, with and without reduction recognition.

We see from this table that the advanced array analyses are applicable to a majority of the programs in the benchmark suite, and several programs can take advantage of all the interprocedural array analyses. Although the techniques do not apply uniformly to all the programs, the frequency with which they are applicable for this relatively small set of programs demonstrates that the techniques are general and useful. We observe that many of the parallelizable loops do not require any new array techniques. However, the coarse-grained loops, parallelized with advanced array analyses, often contain a significant potion of the overall computation of the program and, as shown below, can make a substantial difference in overall performance.

	# of loops	**Parallel Loops**								
		Intraprocedural				Interprocedural				Total
Array Reduction										
Array Privatization										
SPEC95fp										
tomcatv	16	10								10
swim	24	22								22
su2cor	117	89								89
hydro2d	163	155								155
mgrid	46	35								35
applu	168	127	10	6		6				149
turb3d	70	55		3		4				62
apsi	298	169				2				171
wave5	362	307								307
SPEC92fp										
doduc	280	230				7				237
mdljdp2	33	10	2	1			2			15
wave5	364	198								198
tomcatv	18	10								10
ora	8	5				3				8
mdljsp2	32	10	2	1			2			15
swm256	24	24								24
su2cor	128	65	3			1				69
hydro2d	159	147								147
nasa7	133	59	1	6						66
Nas										
appbt	192	139	3	18		6			3	169
applu	168	117	4	6		6			3	136
appsp	198	142	3	12		6			3	166
buk	10	4								4
cgm	31	17	2							19
embar	8	3	1						1	5
fftpde	50	25								25
mgrid	56	38								38
Perfect										
adm	267	172				2		2		176
arc2d	227	190								190
bdna	217	111	28					1		140
dyfesm	203	122	5	2			1	5		135
flo52	186	148		1		7				156
mdg	52	35		1				2		38
mg3d	155	104	2							106
ocean	135	102	1	6						109
qcd	157	92	7							99
spec77	378	281	13	2		17			1	314
track	91	51	3			1				55
trfd	38	15	5	1						21
TOTAL	5262	2635	95	66	0	68	5	10	11	2890

Figure 7-6. Static Measurements: Number of parallel loops found by each technique

7.4.2. Dynamic Measurements

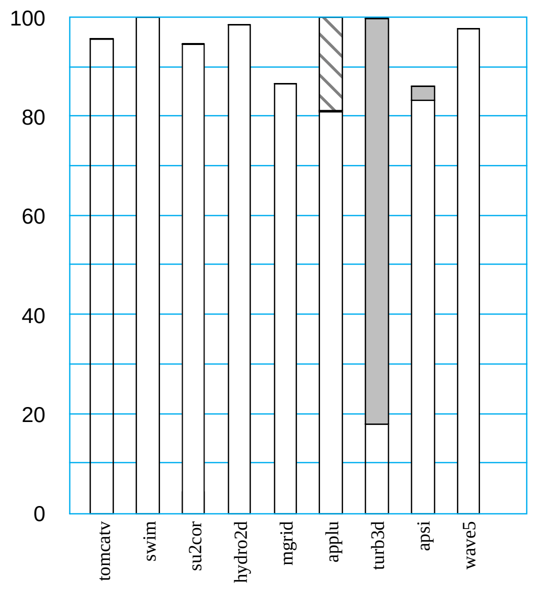
We also measure the dynamic impact of each of the advanced array analyses. We present the results for Nas benchmark with both small and large data sets. The three benchmarks with small execution times—SPEC92fp, Nas with small dataset and Perfect—are executed on the Silicon Graphics Challenge multiprocessors, and the results are given in Figures 7-8, 7-10 and 7-11 respectively. The other two benchmarks, SPEC95fp and Nas with the large data set, are tested on the Digital AlphaServer, and the results are given in Figures 7-7 and 7-9.

While parallel speedups measure the overall effectiveness of a parallel system, they are also highly machine dependent. Not only do speedups depend on the number of processors, they are sensitive to many aspects of the architecture, such as the cost of synchronization, the interconnect bandwidth, and the memory subsystem. Furthermore, speedups measure the effectiveness of the entire compiler system and not just the parallelization analysis. Thus, to capture more precisely how well the parallelization analysis performs, we measure the *parallelism coverage* and the *granularity of parallelism*, as explained below.

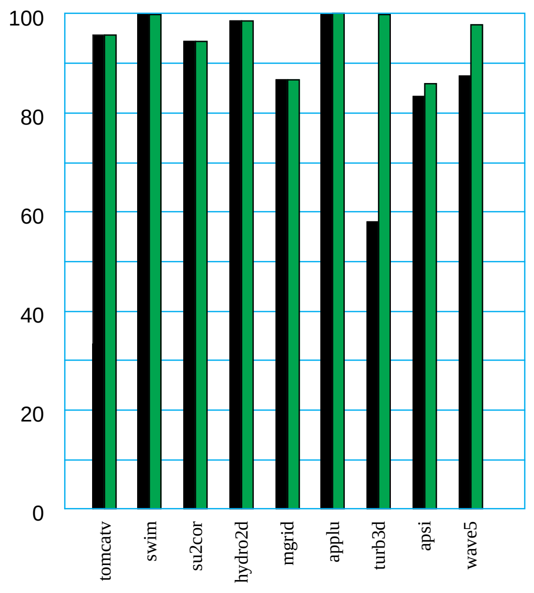
7.4.2.1. Parallelism coverage

We term the overall percentage of time spent in parallelized regions as the *parallelism coverage*. The coverage measurements are taken by running the programs on a single processor of the multiprocessor. As the coverage results are reported in relative terms, they are less sensitive to differences between processors. Parallel coverage is an important metric for measuring the effectiveness of parallelization analysis. By Amdahl's law, programs with low coverage will not achieve good parallel speedup. For example, a program with a parallel coverage of 80% can at most speedup by 2.5 on 4 processors. High coverage is indicative that the compiler analysis is locating significant amounts of parallelism in the computation.

In the figures, we present the contribution of each analysis component to the parallel coverage of the SUIF compiler. These coverage measurements were taken by recording the specific array analyses that apply to each parallelized loop, and instrumenting the sequen-



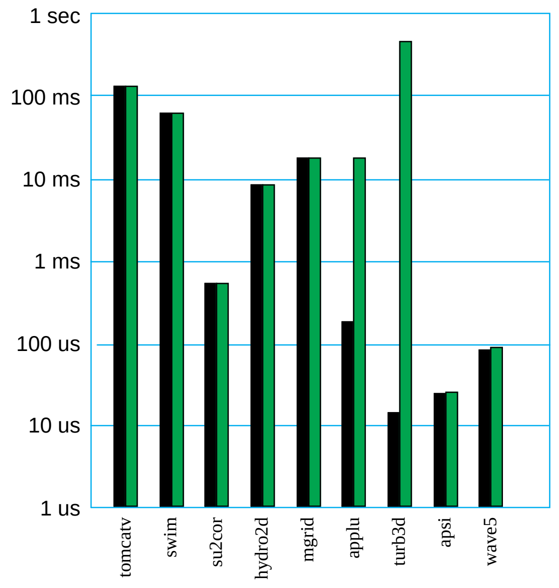
(A) Applicable % of Computation



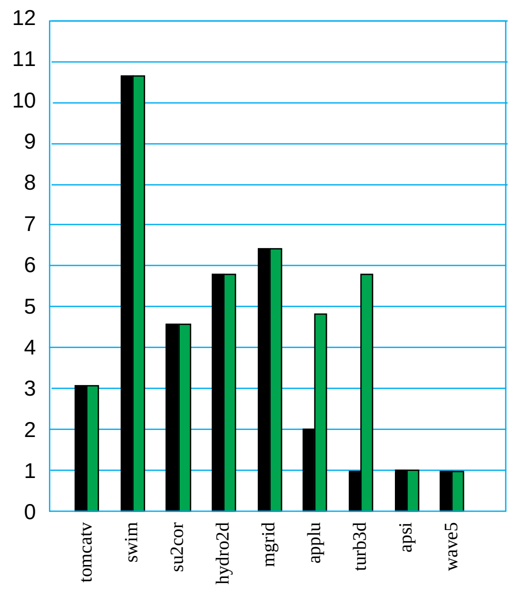
(B) Parallelism Coverage (%)

Intra-	Inter-	Techniques
procedural		
□	▒	Data Dependence Analysis
▨	▨	+ Array Reduction
▧	▧	+ Array Privatization
▩	■	+ Array Reduction + Array Privatization

■	Baseline:	Interprocedural Intraprocedural	Scalar Analysis Data Dependence Analysis
🟩	SUIF:	Interprocedural	Scalar Analysis Data Dependence Analysis Array Privatization Array Reduction

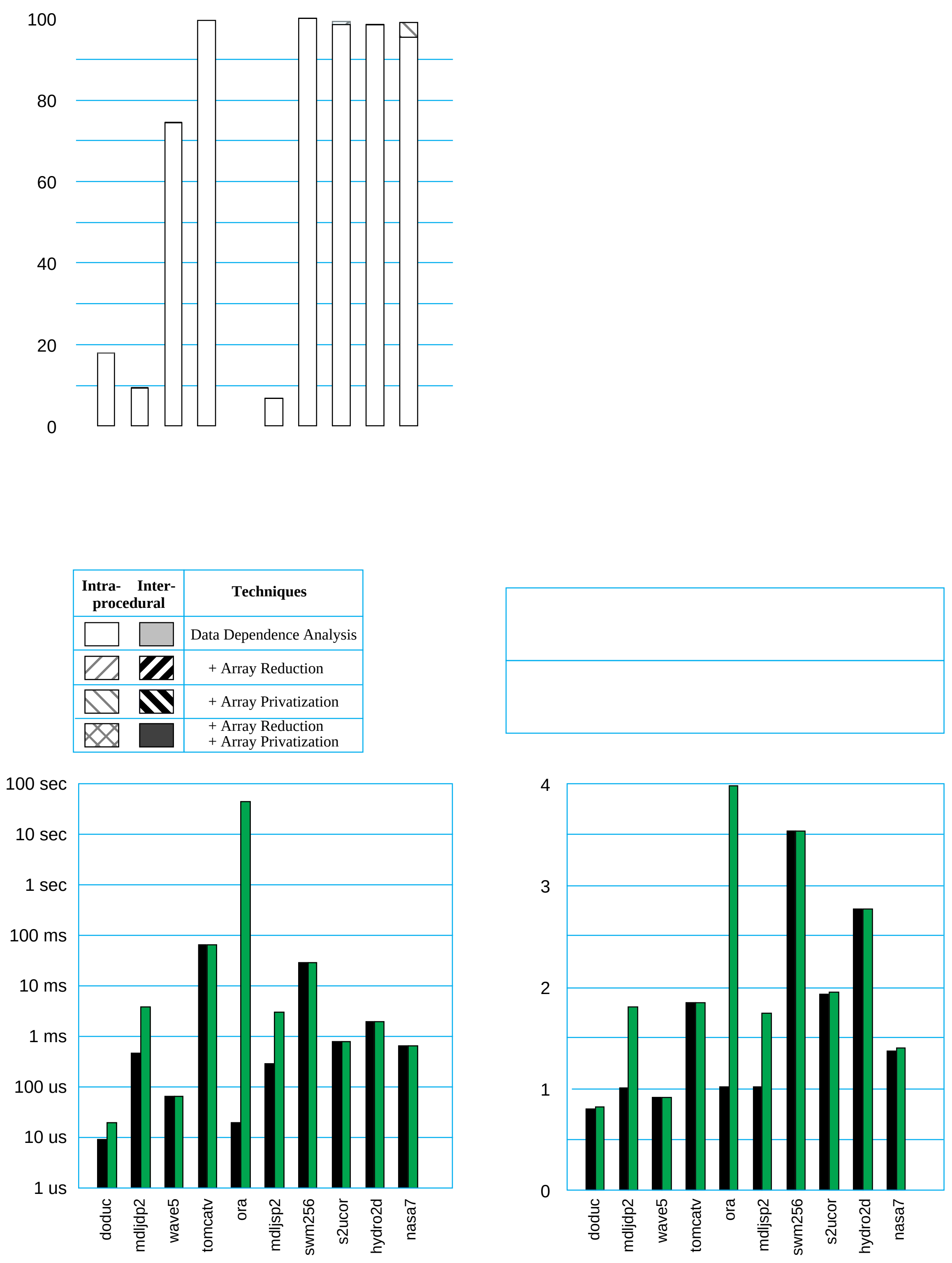


(C) Granularity of Parallelism



(D) Speedup on 8 Processors

Figure 7-7. Dynamic Measurements on the AlphaServer for SPEC95fp



Intra- procedural	Inter- procedural	Techniques
□	▩	Data Dependence Analysis
▨	▨	+ Array Reduction
▧	▧	+ Array Privatization
▩	■	+ Array Reduction + Array Privatization

Figure 7-8. Dynamic Measurements on the Challenge for SPEC92fp