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 $\Im \subseteq \mathcal{Z}^n$.

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*

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

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*

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

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.

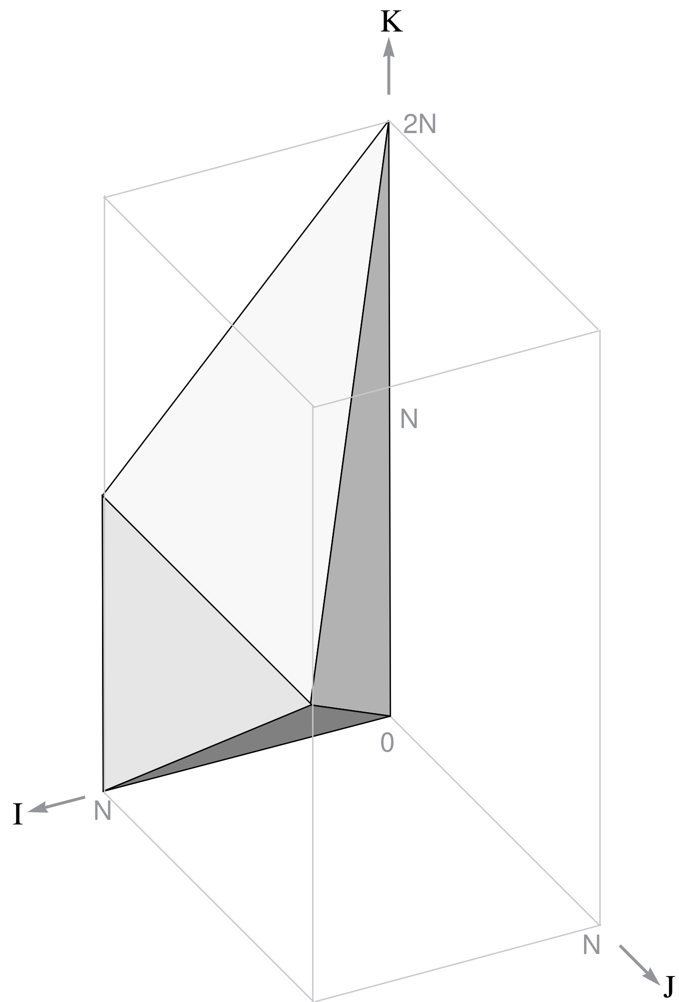


Figure 2-3. Convex polyhedron representing the iteration space

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 Figure 2-5. 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 Figure 2-1 is given by the scanning order (I, J, K). A loop nest with the same scanning order is generated by the projec-

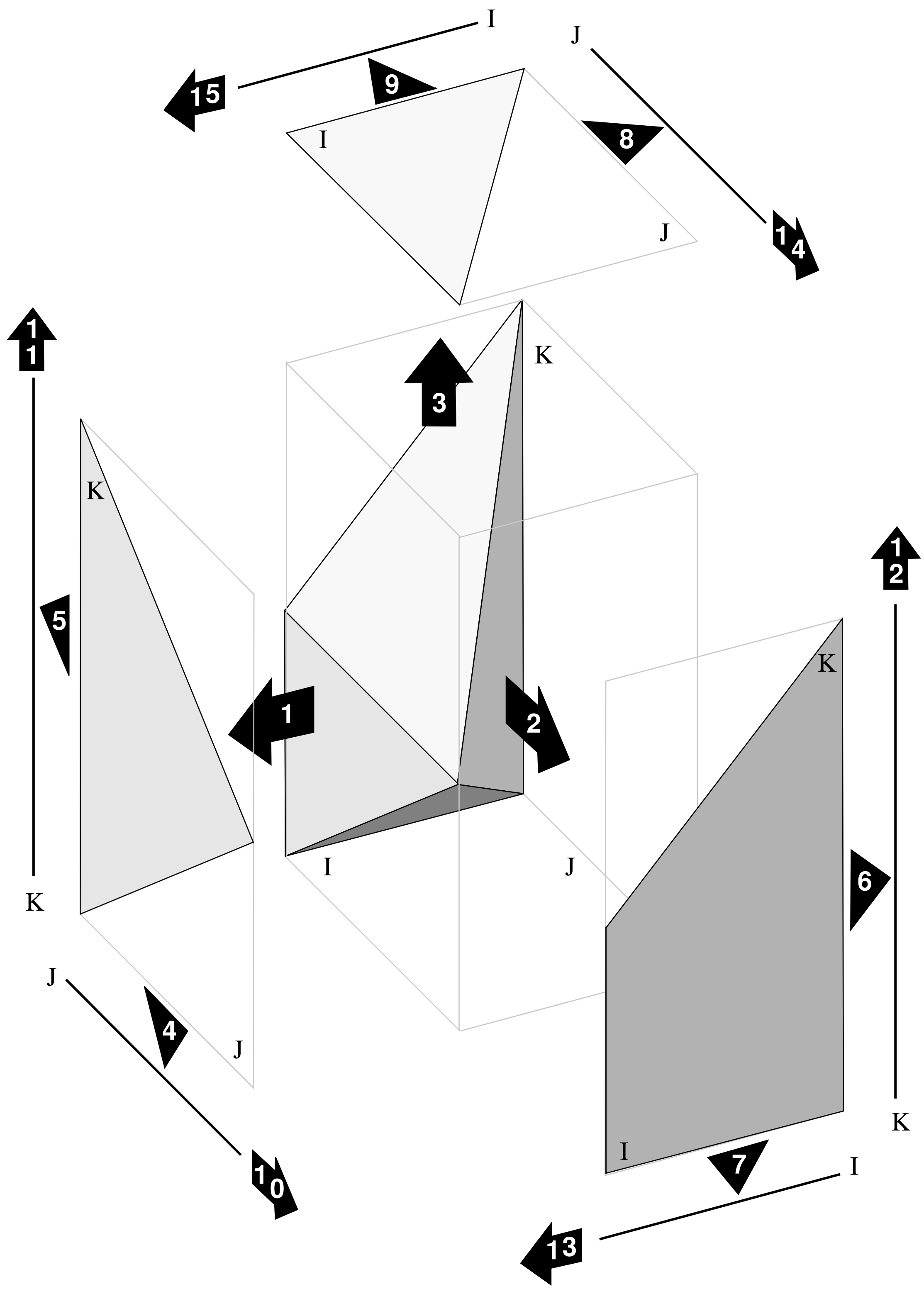


Figure 2-4. Projecting for all the possible scanning orders

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.

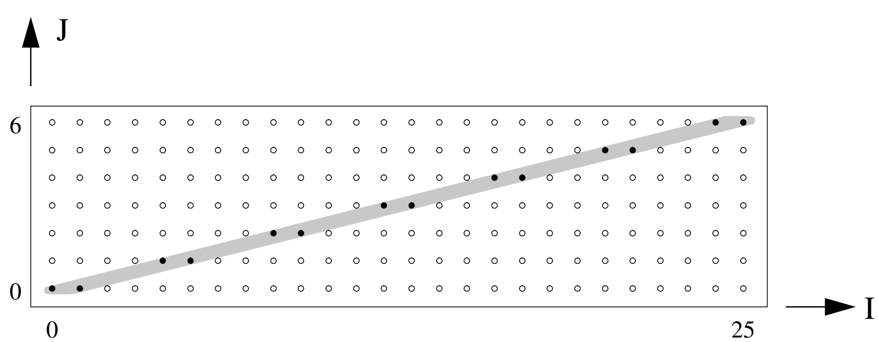
While the algorithm described above is correct, the generated code can be inefficient. Although the iteration space is dense, the presence of tight bounds may create gaps in the iterations. We demonstrate this using the example loop, in Figure 2-6(a), with the iteration space given in Figure 2-6(b).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

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

(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:

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

$$\textbf{DO}\ v_k = v_k',\ min(v_k' + \beta - \gamma, h)$$

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:

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

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 $S \cap \{\neg I\} \neq \varnothing$ **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. Figure 2-10 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 $\{a_o + a_1 v_1 + \ldots + a_k k_k + b_1 i_1 + \ldots + b_n i_n \geq 0\} \in S$
> 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 $\{-a_o - a_1 v_1 - \ldots - a_k k_k - b_1 i_1 - \ldots - b_n i_n \geq 0\} \in S$ **then**
> >
> > > $c = c - 2n$
> >
> > **if** $|b_n| \neq 1$ **then**
> >
> > > $c = c + n$
> >
> > $W\left(\{a_o + a_1 v_1 + \ldots + a_k k_k + b_1 i_1 + \ldots + b_n i_n \geq 0\} \right) = c$
>
> 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*

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

where $v_1, \ldots, v_n$ *are integer variables,* $u_1, \ldots, u_m > 0$ *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*

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

where $v_1, \ldots, v_n$ *are integer variables,* $u_1, \ldots, u_m > 0$ *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* $\forall y \quad (0 \leq y \leq m)$ *,* $pa_y^n = qb_y^n$ *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-

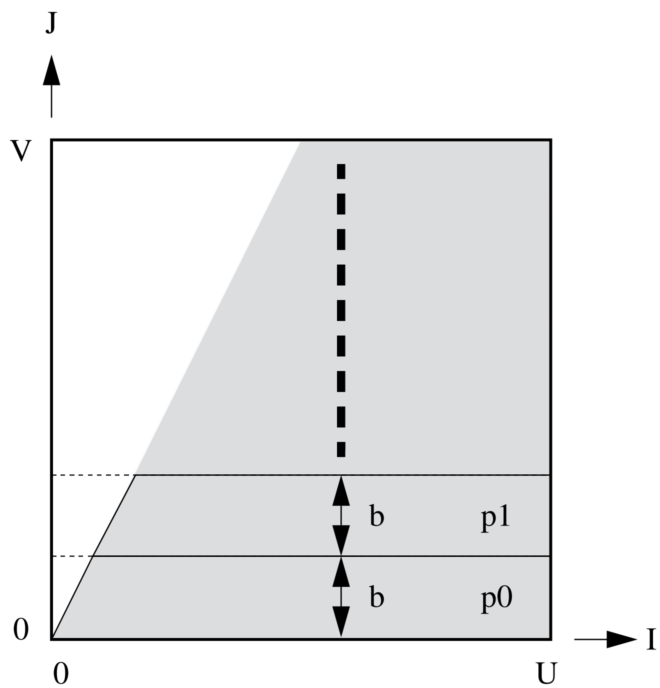


Figure 2-13. Iteration space

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 $\mathcal{A}$ and the processor space $\mathcal{P}$.

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

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*

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

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*

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

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 for many 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

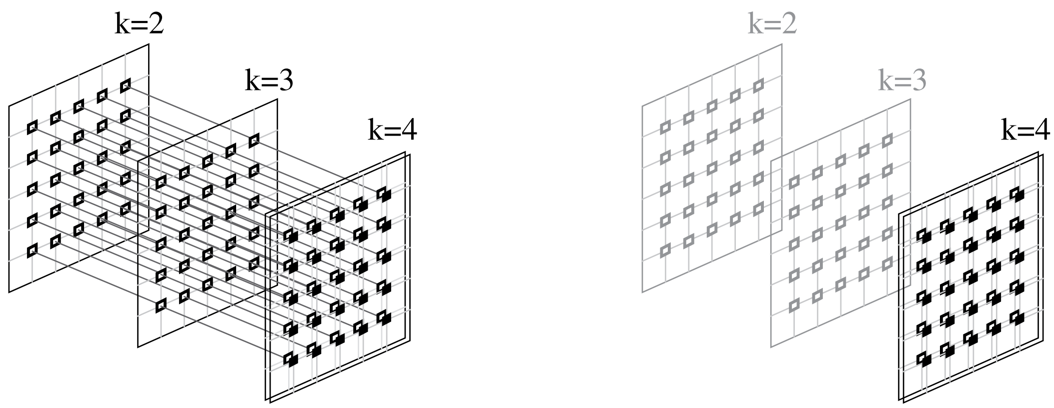
```
DO K = 2, NZ-1
     DO M = 1, 5
          DO N = 1, 5
               TM(1:5,M) = ...
     DO M = 1, 5
          DO N = 1, 5
               ... = TM(N,M)
```

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.



(a) Location-based dependences (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 Section 4.6, 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 Figure 4-1 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-

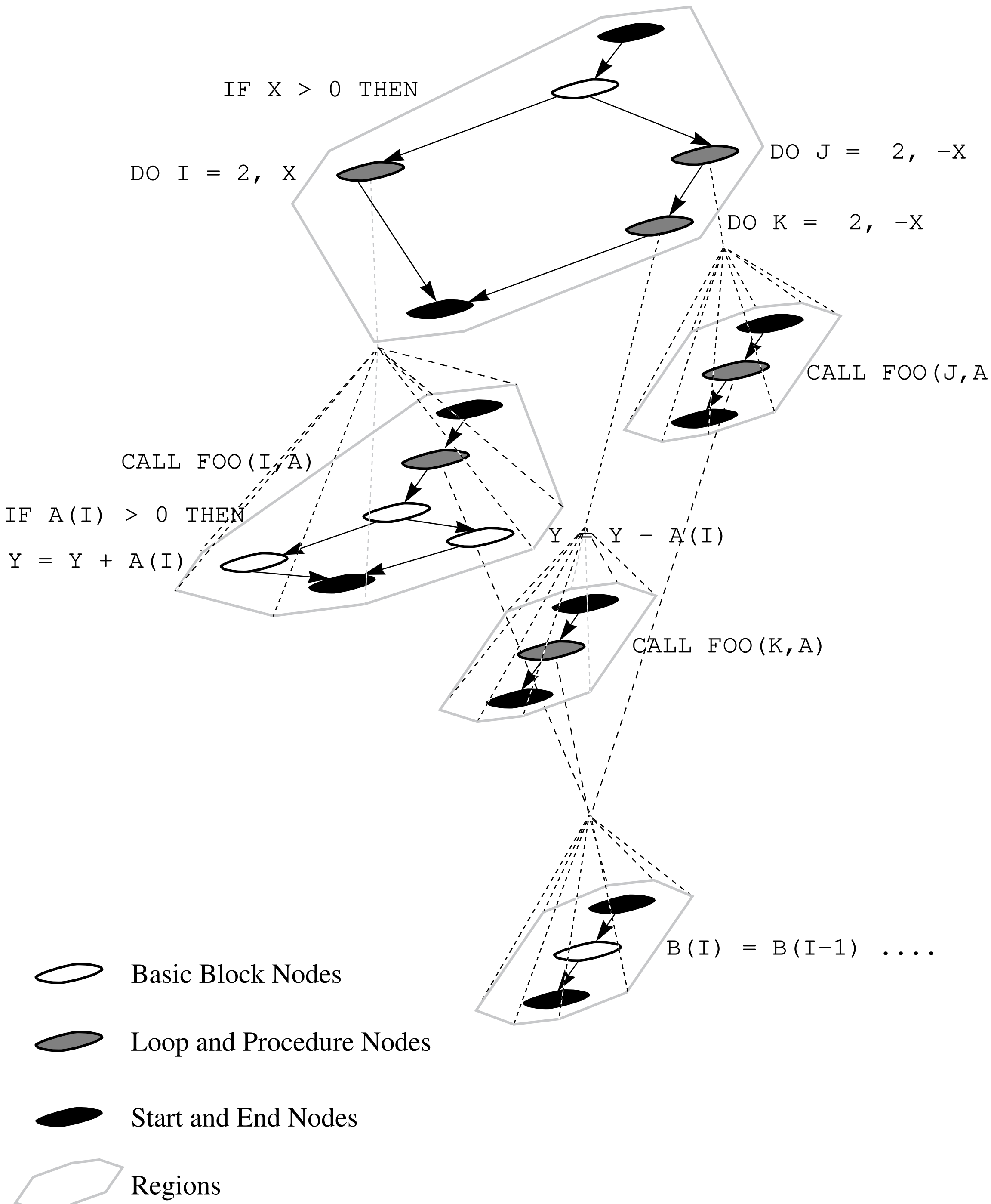


Figure 4-2. Regions graph of the example program in Figure 4-1.

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. Figure 4-3 shows the bottom-up, forward-flow propagation for the regions graph example in Figure 4-2. 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 Figure 4-4. We start by calculating the local values for each node.We define the function Loc that provides the local value for

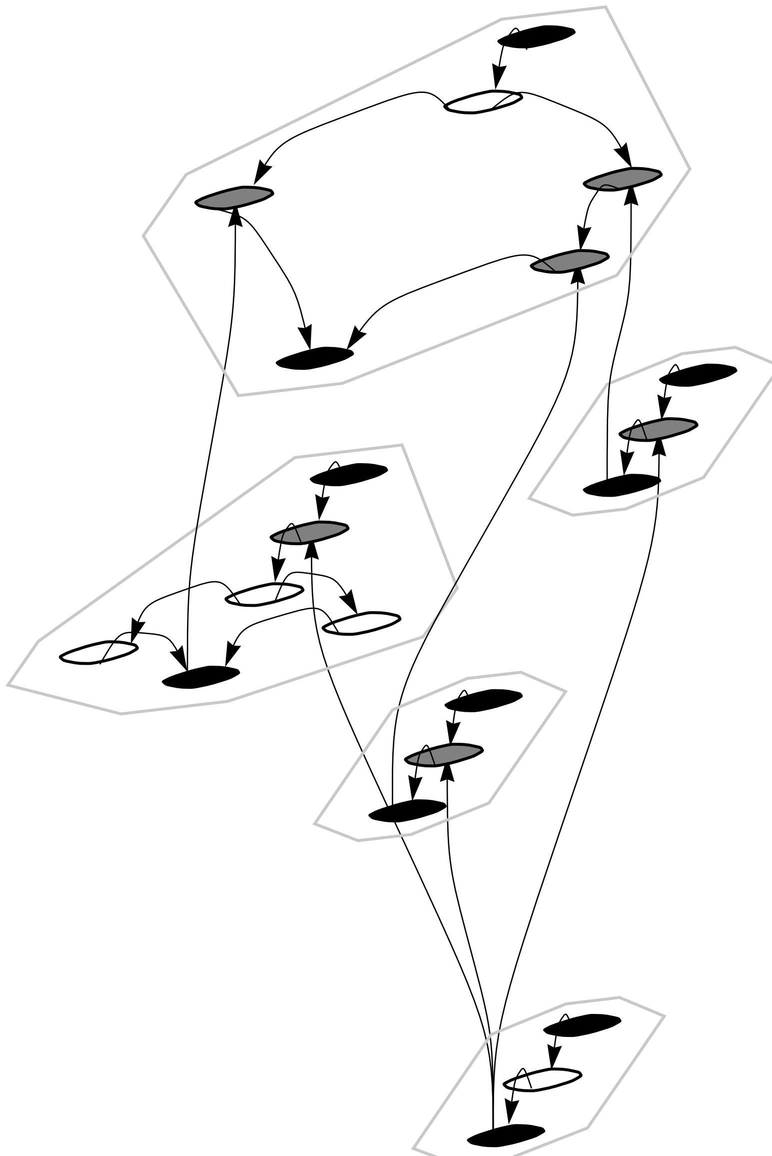


Figure 4-3. Bottom-up, forward-flow traversal

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**

$l_n = \perp$

if forward-flow problem **then**

for each node $n \in N$ **do**

$$V_n = T\left(\bigwedge_{(n', n) \in E} V_{n'}, l_n \right)$$

$V_R = V_e$

else if backward-flow problem **then**

for each node $n \in N$ **do**

$$V_n = T\left(\bigwedge_{(n, n') \in E} V_{n'}, l_n \right)$$

$V_R = V_s$

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,

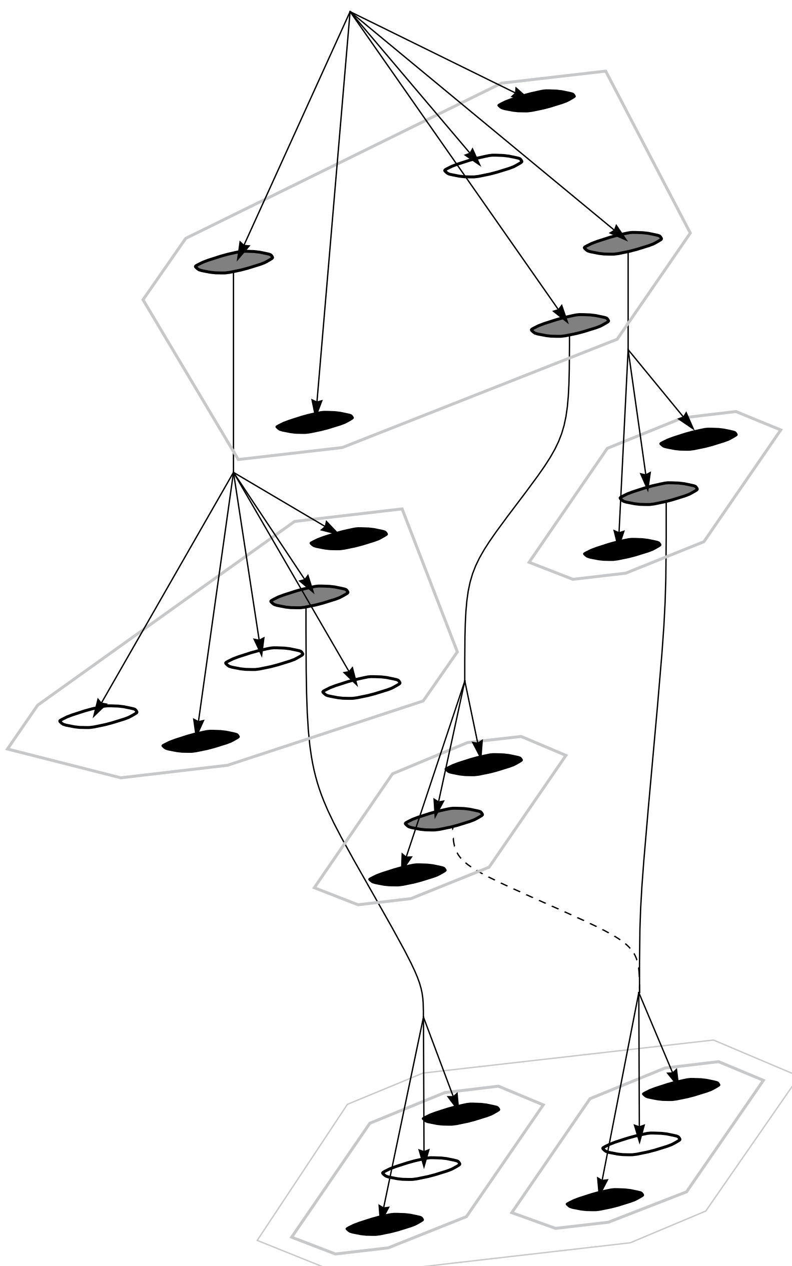


Figure 4-5. Top-down, flow-insensitive pass needing selective procedure cloning

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 Figure 4-6. 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 $\Downarrow_n$ 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* $\mathcal{A}$*. Each array index of the array is an n-tuple* $(a_1, \ldots, a_n) \in Z^n$ *such that* $(a_1, \ldots, a_n) \in \mathcal{A}$ *if and only if* $\forall_{i = 1, \ldots, n}\ l_i \le a_i \le u_i$ *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 $\mathcal{A}$.

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* $r(i_1, \ldots, i_x) \subseteq \mathcal{A}$.

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 $\langle \mathcal{A}, \mathcal{A}, \emptyset, \emptyset \rangle$.

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 $\langle \emptyset, \emptyset, \mathcal{A}, \emptyset \rangle$.

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

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 by the entire index set $\mathcal{A}$. 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

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

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* $W(i_1, \ldots, i_{n-1}, k) \cap R(i_1, \ldots, i_{n-1}, l) \ne \varnothing$.

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* $R(i_1, \ldots, i_{n-1}, k) \cap W(i_1, \ldots, i_{n-1}, l) \ne \varnothing$.

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.*
$W(i_1, \ldots, i_{n-1}, k) \cap W(i_1, \ldots, i_{n-1}, l) \ne \varnothing$.

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* $W(i_1, \ldots, i_{n-1}, k) \cap E(i_1, \ldots, i_{n-1}, l) \ne \varnothing$.

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* $\forall_{k = 1, \ldots, u} \quad W(i_1, \ldots, i_{n-1}, k) \subseteq W(i_1, \ldots, i_{n-1}, u)$.

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*

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

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 $\mathcal{A}$ 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*

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

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.*

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

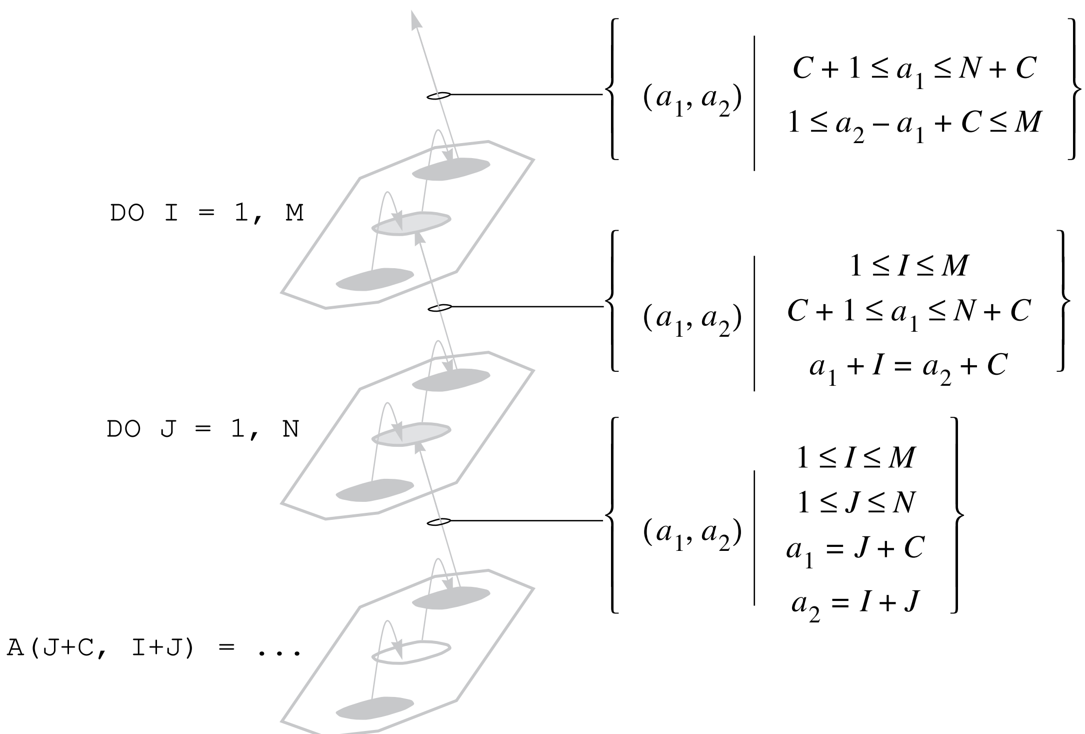


Figure 5-2. Summarizing the array access patterns

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 Section 4.3.3.1.

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 Definition 5-1, 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 Figure 5-4 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.

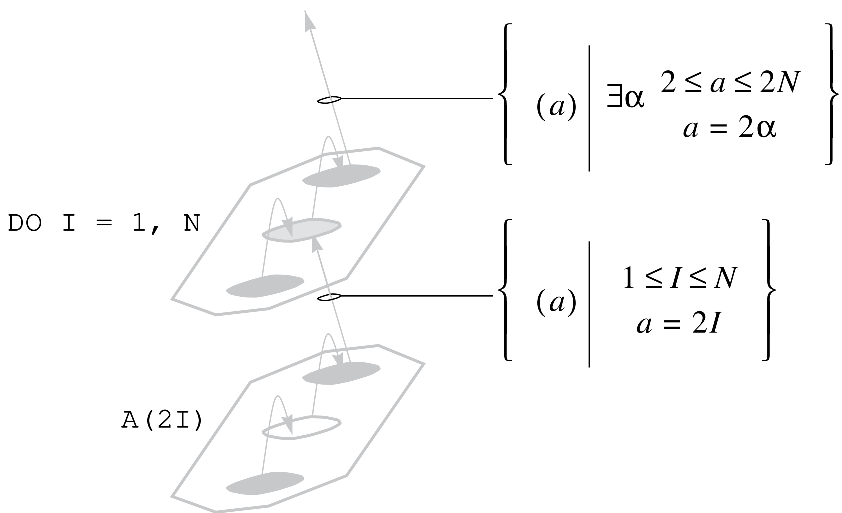


Figure 5-4. An array summary with an auxiliary variable

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 Figure 5-5, 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 Figure 5-6. 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*.

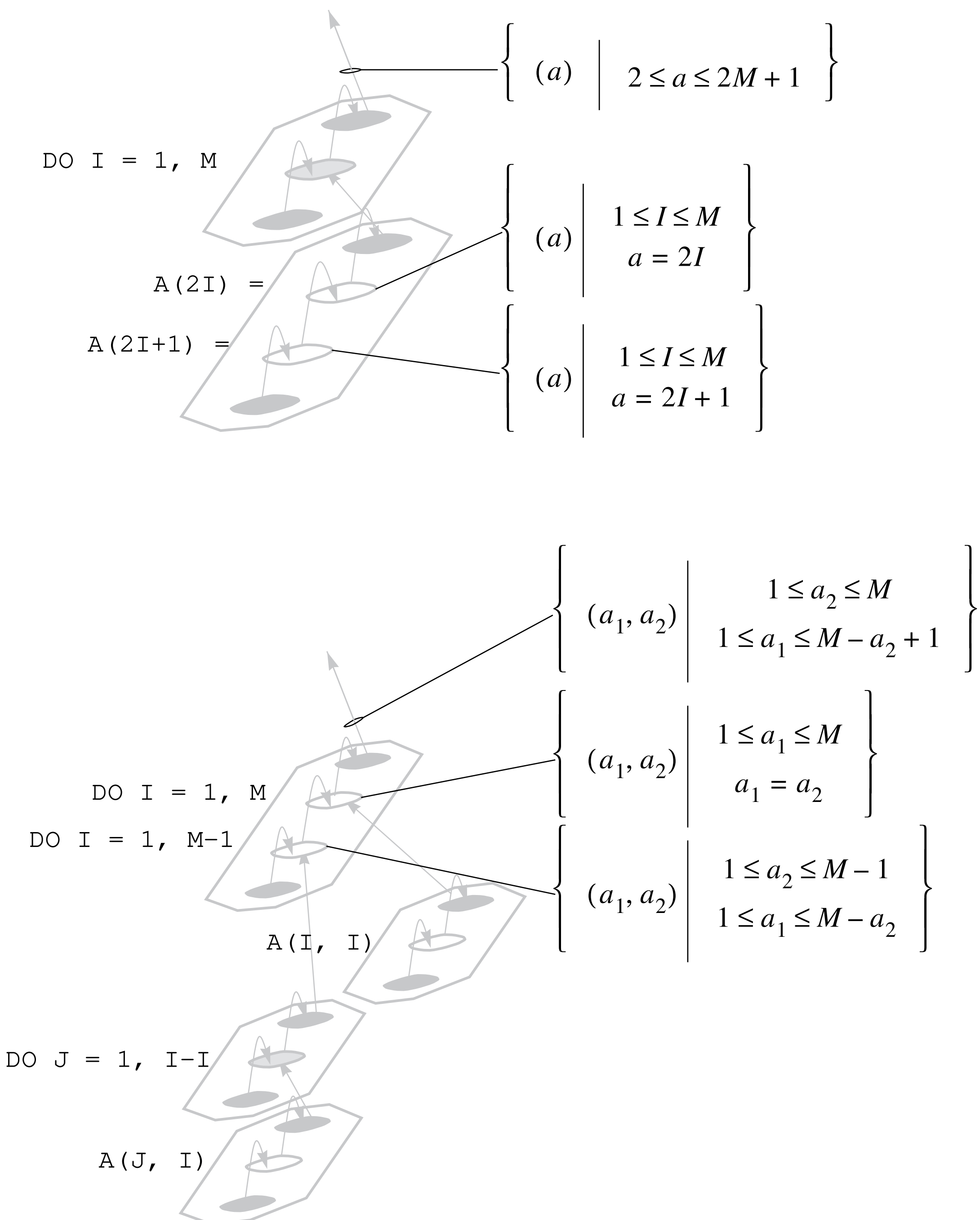


Figure 5-7. Examples of convex array sections that can be merged after union operator.

$MergeSimple(R_1, R_2) \rightarrow \langle R, \{true, false\} \rangle$
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 $c_1^1 = c_2^1 \ldots c_1^k = c_2^k$ and $n_1 = n_2$ **then**

$l = min(l_1, l_2)$

$u = max(u_1, u_2)$

else if $c_1^1 = c_2^1 \ldots c_1^k = c_2^k$ 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 $c_1^1 = c_2^1 \ldots c_1^k = c_2^k$ 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 $\langle \emptyset, false\rangle$

$$R = \left\{ \begin{array}{l} -l - c_1^1 i_1 - \ldots - c_1^k i_k + n_1 \alpha \geq 0 \\ u + c_1^1 i_1 + \ldots + c_1^k i_k - n_1 \alpha \geq 0 \end{array} \right\} \text{ where } \alpha \text{ is a new auxiliary variable}$$

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 α_1 used by the inequalities of R_1 **do**

$R_1^{\alpha} = \{I | I \in R_1 and$ variable α_1 is in $I\}$

for each auxiliary variable α_2 used by the inequalities of R_2 **do**

$R_2^{\alpha} = \{I | I \in R_2 and$ variable α_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 α_1 is not in $I\}$

$R_2 = \{I | I \in R_2 and$ variable α_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 Figure 5-12, 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 Figure 5-13. Thus, we define

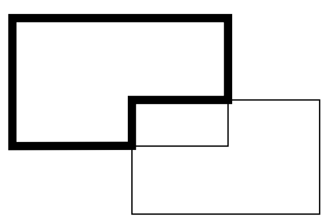


Figure 5-13. An example of a subtraction of two convex sections resulting in a single non-convex section

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 Figure 5-14. 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.

$Substract(R_1, R_2) \to R$
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** $\varnothing$
>
> **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 α 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 Figure 5-19 removes these redundant inequalities and auxiliary variables.

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

for all auxiliary variables α in R **do**

  for all auxiliary variables β 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

---

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

for all auxiliary variables α 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 Figure 5-20. 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.

$RemoveSimpleRedundant(R) \rightarrow R'$
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 Figure 5-23. 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 Figure 5-28. 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 Figure 5-29, that our implementation will conservatively assume to be different.

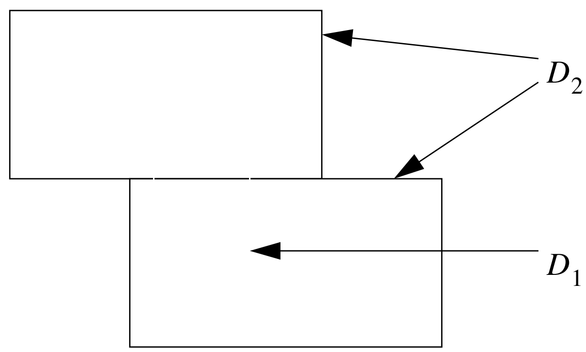


Figure 5-27. Example of the operator $D_1 \subseteq D_2$, where containment is difficult to detect

$IsEquivalent(D_1, D_2) \rightarrow \{true, false\}$
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

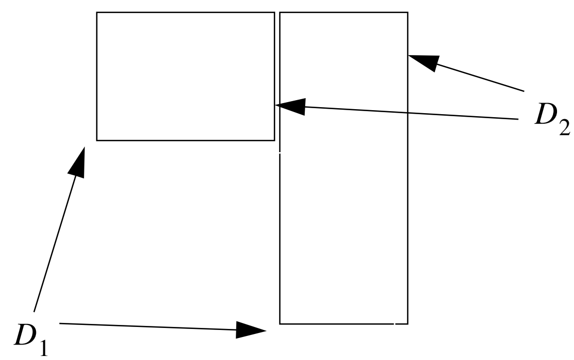


Figure 5-29. Example of equivalent array section descriptors where detection by *IsEquivalent* operator is not possible

times since the result of one subtraction may enable other subtractions. Figure 5-31 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

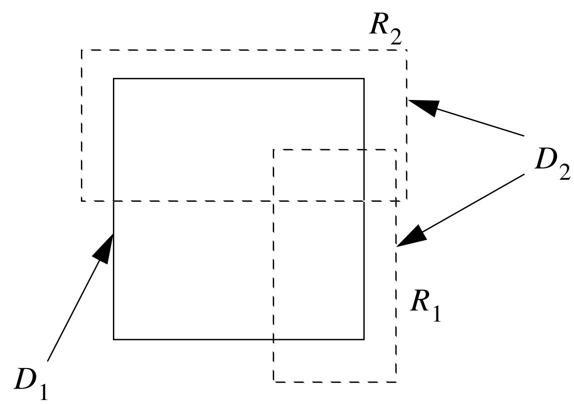


Figure 5-31. Example of a subtraction that needs multiple iterations

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 Figure 3-4, 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 Section 6.2 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 Section 4.3, 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 Figure 6-5. 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,

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

```
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.

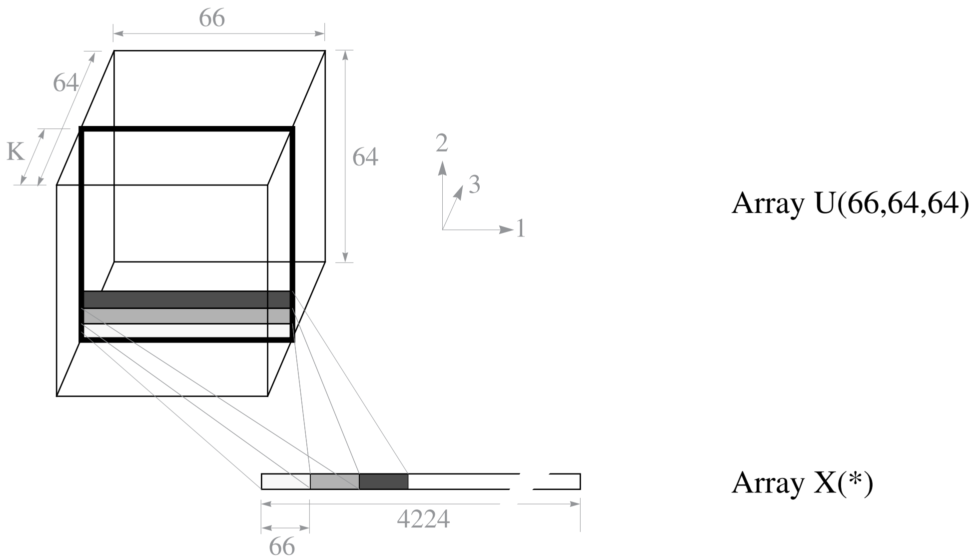


Figure 6-5. The array reshape in turb3d

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,

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

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 $A\left(l_1^A, \ldots, l_n^A\right)$.

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

(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(l1^A:u1^A,...,ln^A:un^A)
INTEGER f1,...,fn
...
f1 = ...
...
fn = ...
CALL FOO(...,A(f1,...,fn),...)
...

SUBROUTINE FOO(...,B,...)
DIMENSION B(l1^B:u1^B,...,lm^B:um^B)
...
```

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* $\underset{1 \le i \le k-1}{\forall} l_i^A = l_i^B$ 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.

$$\texttt{DIMENSION A(}l_1^A\texttt{:}u_1^A\texttt{,...,}l_n^A\texttt{:}u_n^A\texttt{)}$$
$$\texttt{DIMENSION B(}l_1^B\texttt{:}u_1^B\texttt{,...,}l_m^B\texttt{:}u_m^B\texttt{)}$$
$$\texttt{EQUIVALENCE A(}c_1^A\texttt{,...,}c_n^A\texttt{), B(}c_1^B\texttt{,...,}c_m^B\texttt{)}$$
$$\texttt{...}$$

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*

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

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

and $a_1, \ldots, a_n$, $b_1, \ldots, b_m$ 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 Definition 6-3 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 $a_1^k, \ldots, a_{n_k}^k$ 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*

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

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 Figure 7-2. 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 the turb3d 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 Figure 7-3. 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

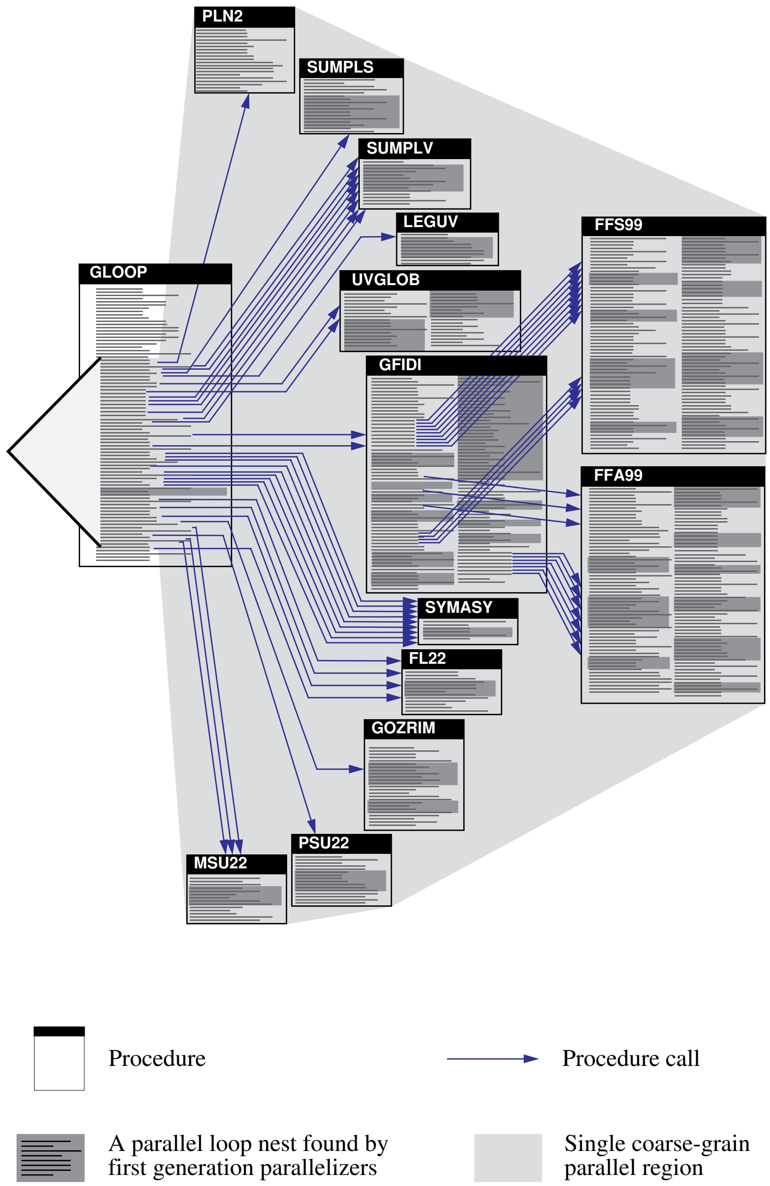


Figure 7-2. Parallelizable regions from a code segment in spec77

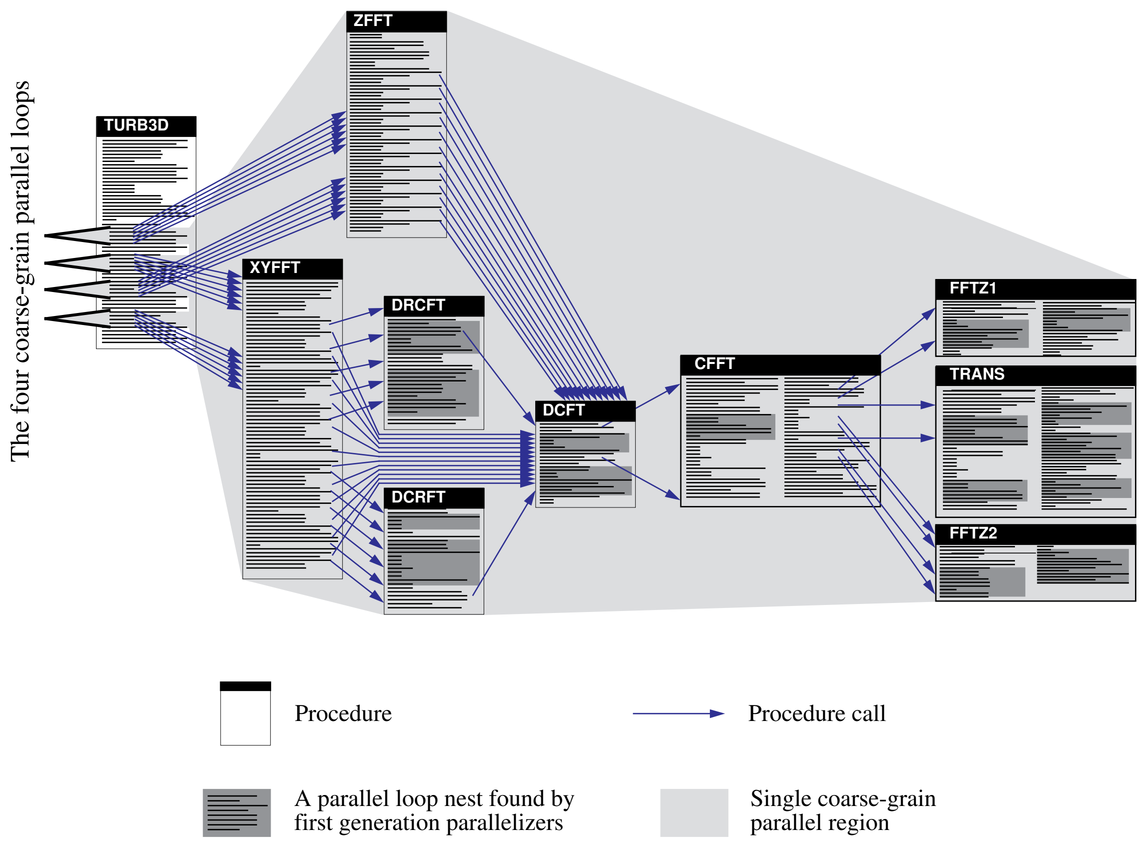


Figure 7-3. Parallelizable regions from a code segment in turb3d

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.

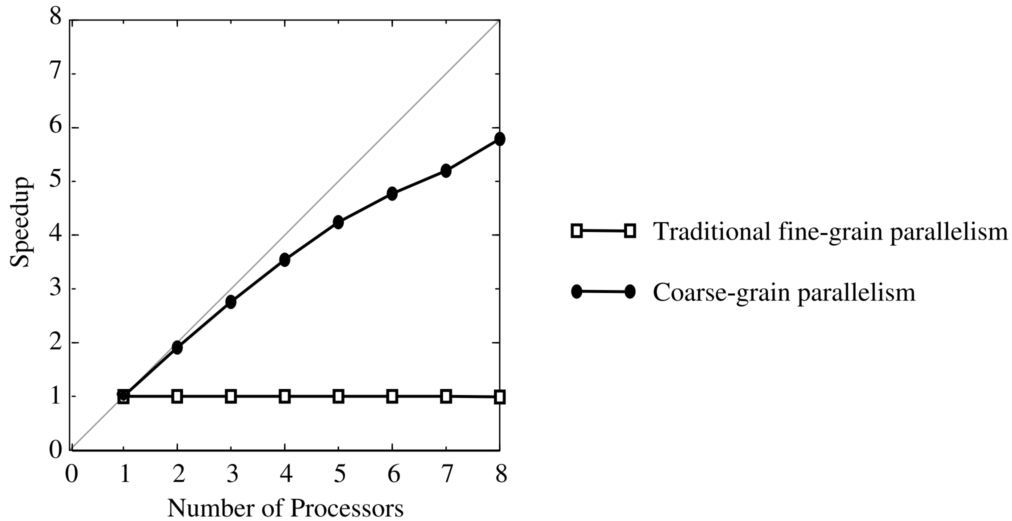


Figure 7-4. Parallel speedup for turb3d on a 8 processor AlphaServer

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-

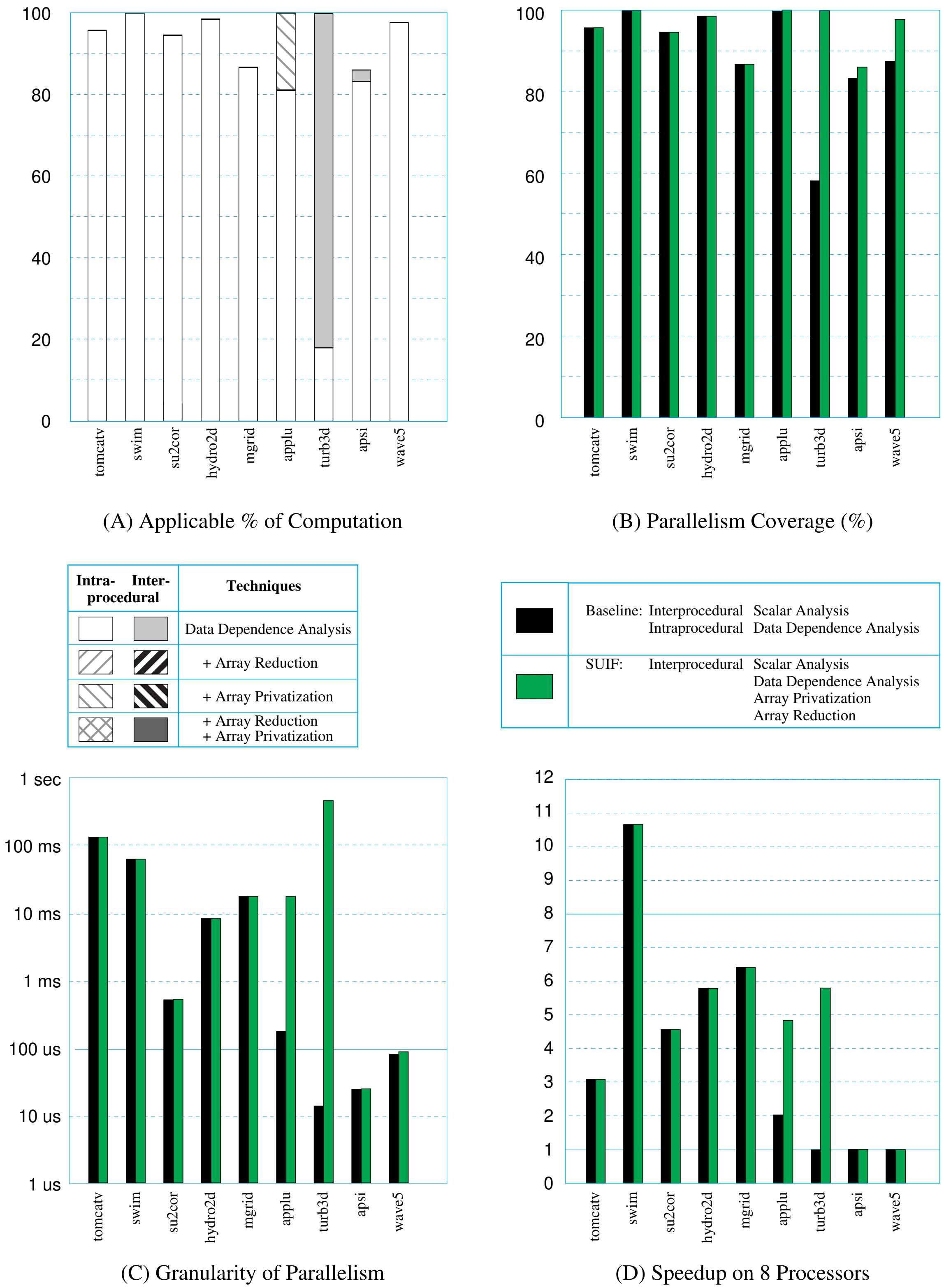
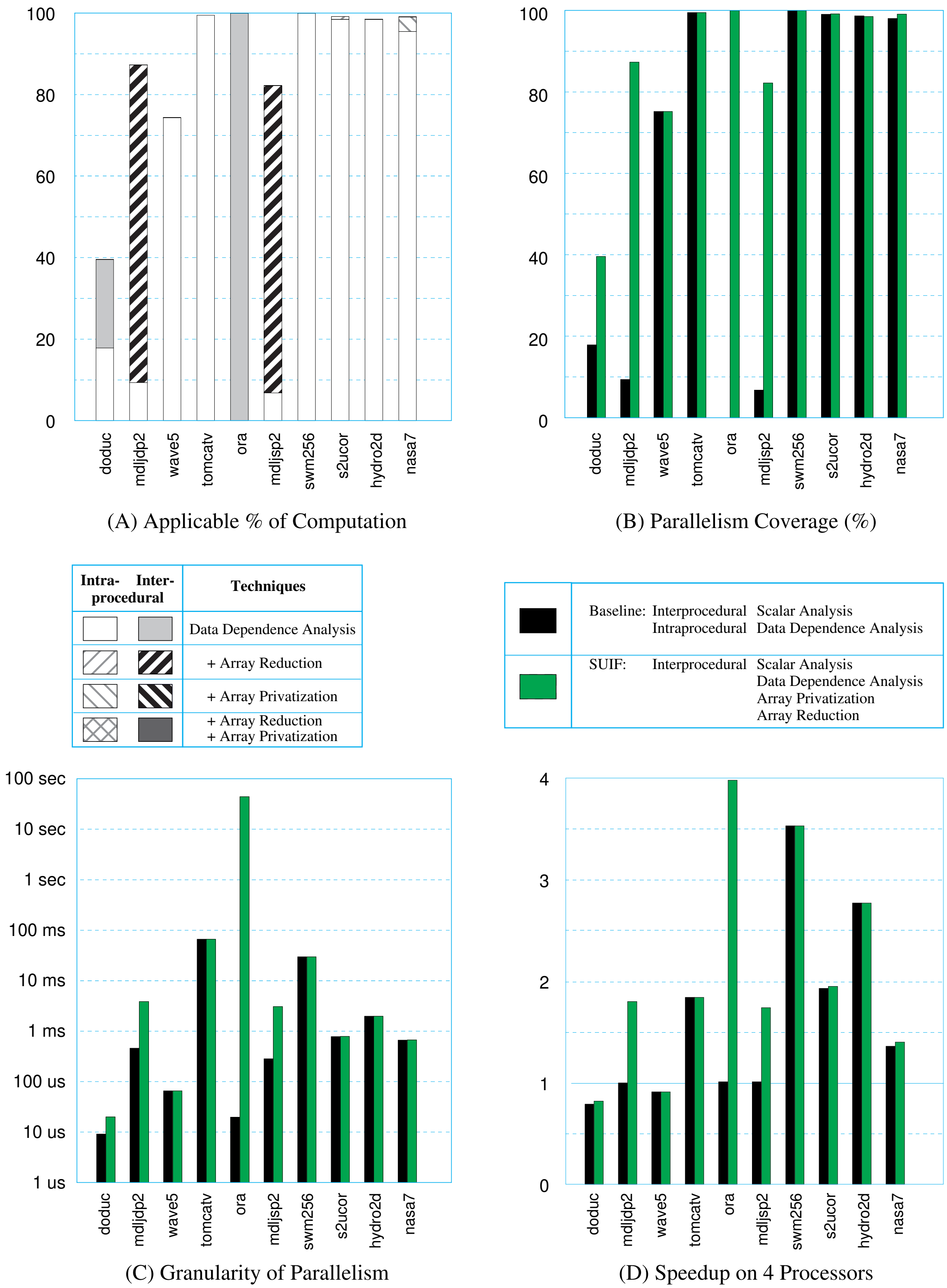


Figure 7-7. Dynamic Measurements on the AlphaServer for SPEC95fp



(1) SPEC92FP

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

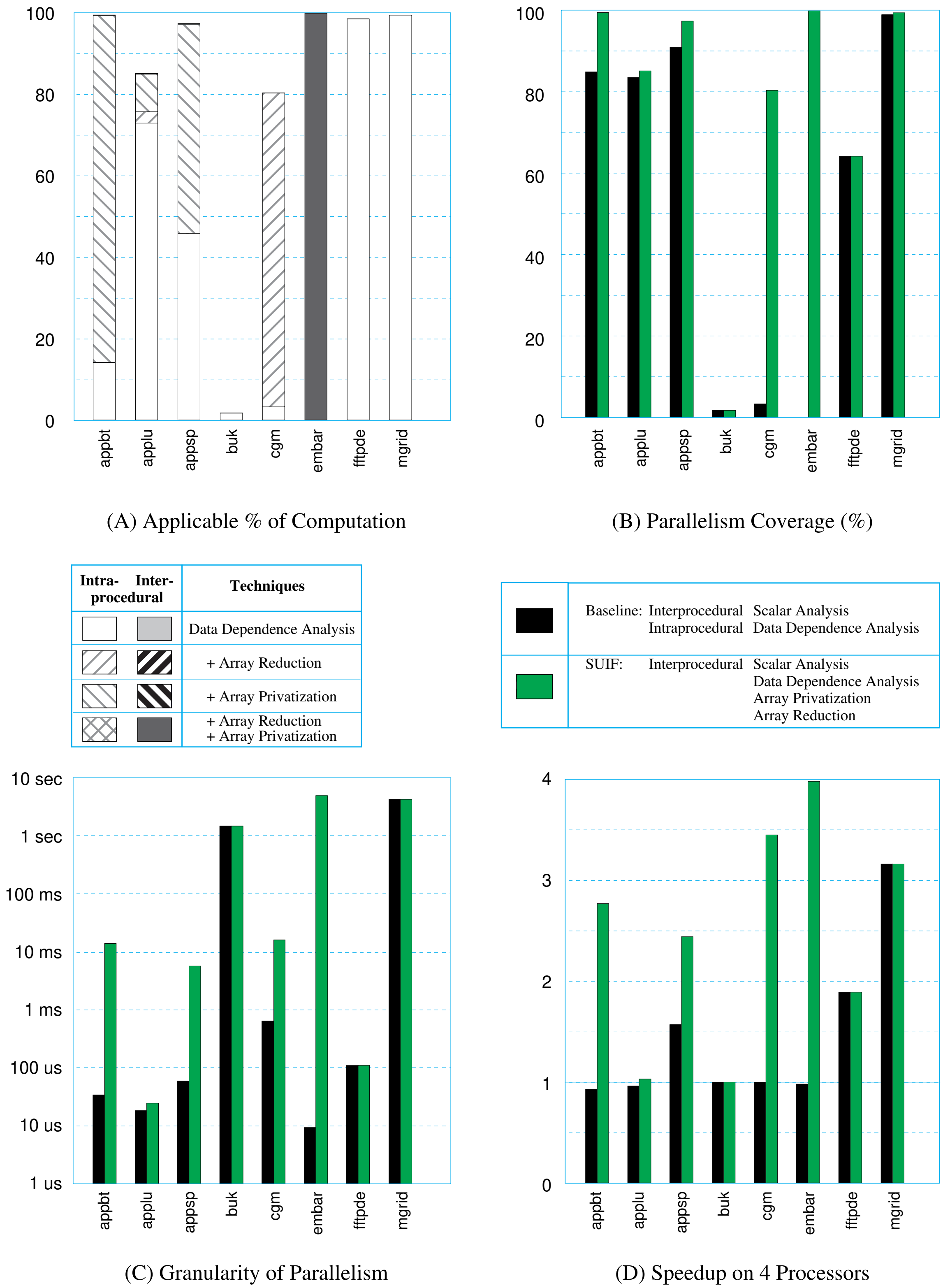
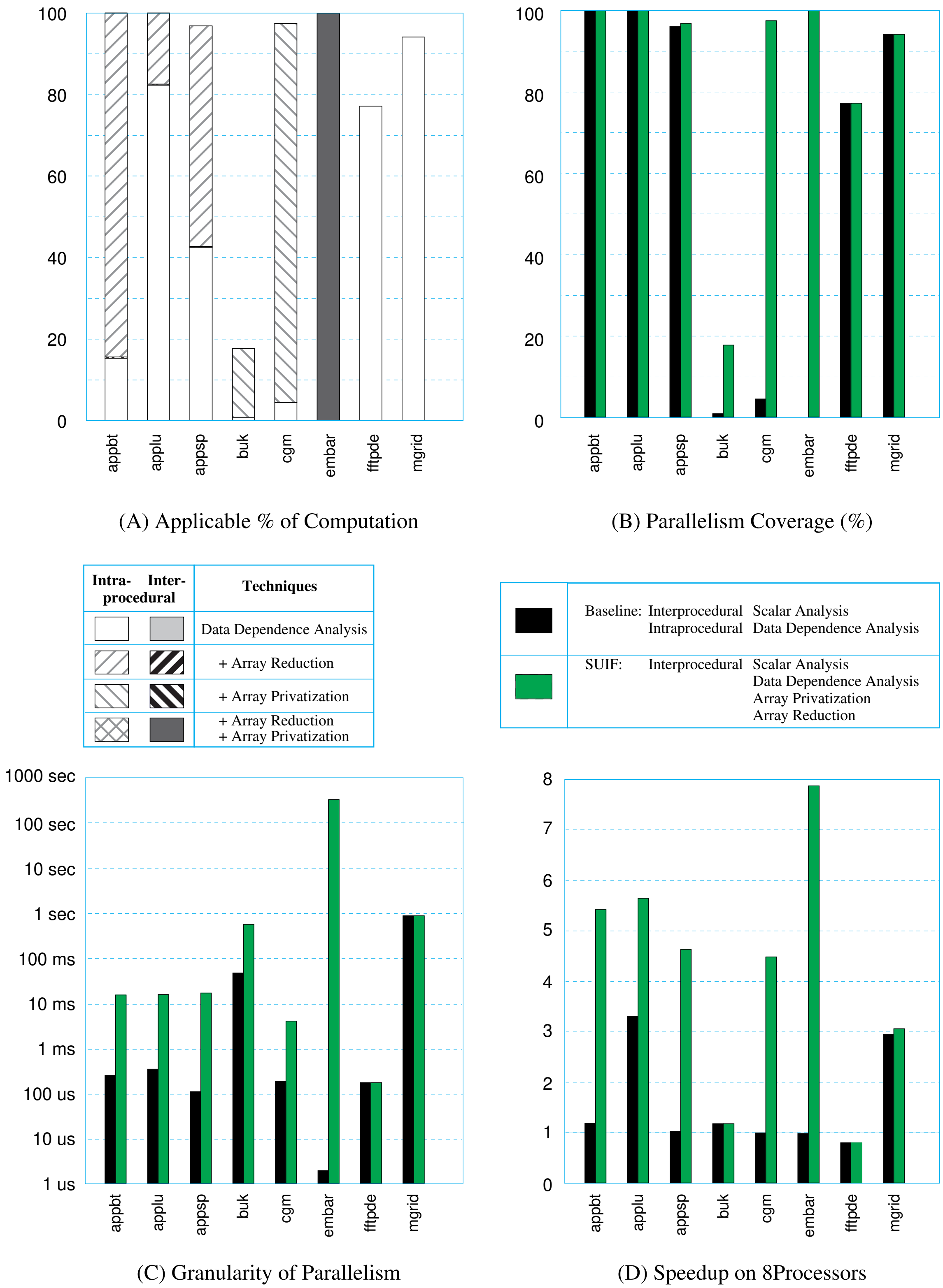


Figure 7-9. Dynamic Measurements on the Challenge for Nas using the small data set



(A) Applicable % of Computation

(B) Parallelism Coverage (%)

(C) Granularity of Parallelism

(D) Speedup on 8Processors

Figure 7-10. Dynamic Measurements on the AlphaServer for Nas using the large data set

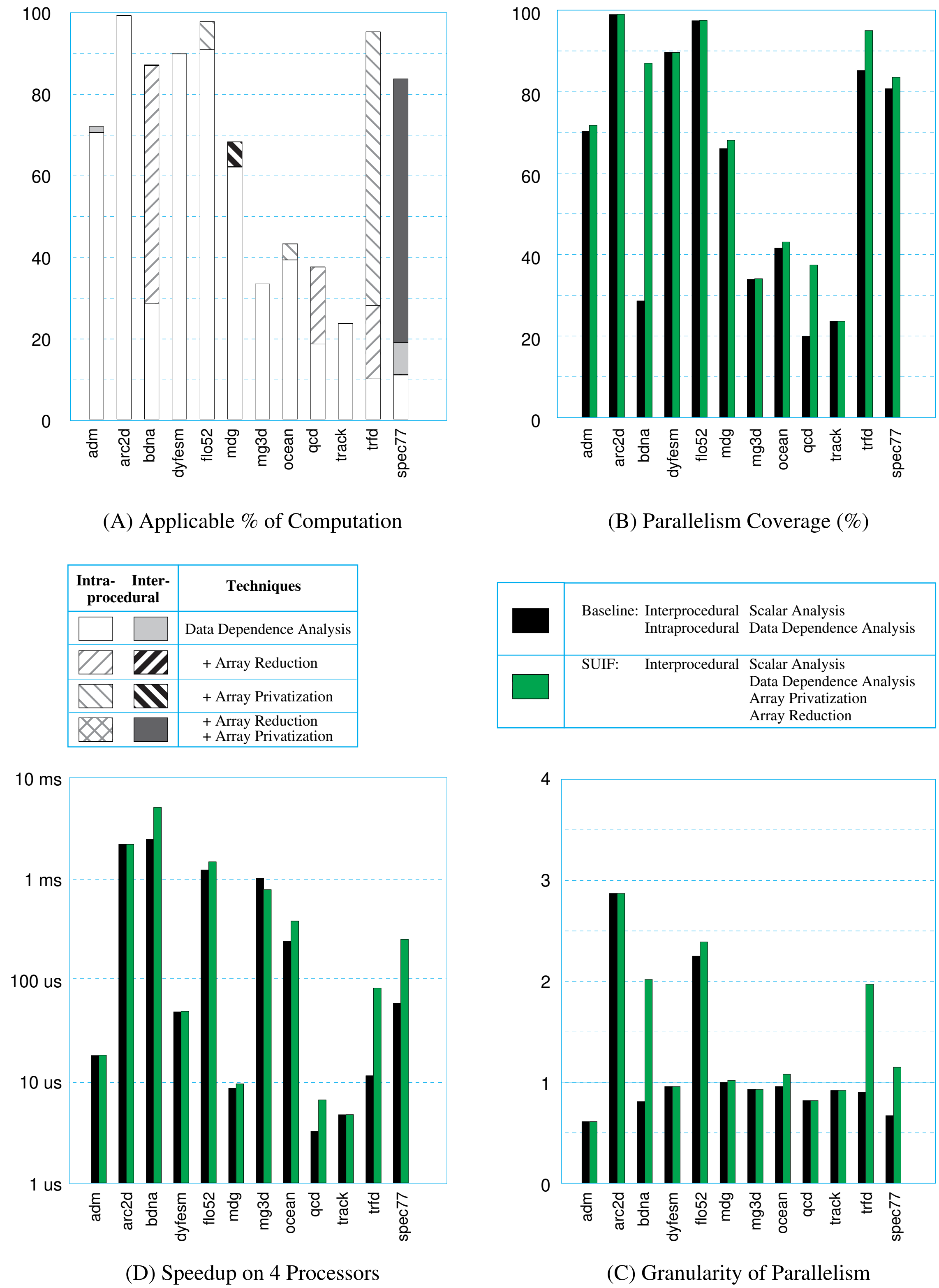


Figure 7-11. Dynamic Measurements on the Challenge for Perfect

tial code to determine the execution time of each of the loops. We also show a comparison of the parallelism coverage achieved by the SUIF and the baseline compiler.

Overall, we observe rather good coverage (above 80%) for all 9 programs in SPEC95fp, 8 of the 10 programs in SPEC92fp, 7 of the 8 Nas programs and 6 of the 12 Perfect benchmarks. A third of the programs spend more than 50% of their execution time in loops that require advanced array analysis techniques.

7.4.2.2. Granularity of parallelism

A program with high coverage is not guaranteed to achieve parallel speedup because of a number of factors. The granularity of parallelism extracted is a particularly important factor, as frequent synchronizations can slow down a fine-grain parallel computation. To quantify this property, we define a program's granularity as the average execution time spent in its parallel regions when the program is executed on a single processor. In the figures, we show a comparison between the granularity achieved by the SUIF and the baseline compiler.

7.4.2.3. Program speedup

The final result we present is a set of speedup measurements. Speedups are calculated as ratios between the execution time of the original sequential program and the parallel execution time. The parallel speedup results are also shown in the figures. Note that the speedups were obtained without any futher optimizations that are enabled by parallelization. The speedups of many of these programs can be improved by performing optimizations to improve data locality and reduce communication and synchronization [8,73].

7.4.3. Discussion

7.4.3.1. SPEC95fp benchmarks

Array privatization has an impact on applu and interprocedural analysis is required for turb3d. The overall coverage of the programs is above 80%. However, apsi and wave5 do not yield any parallel speedup due to the small granularity of the parallel regions. The program swim shows superlinear speedup because its working set fits into the multiprocessor's aggregate cache. The performance of tomcatv and swim can be further improved by memory optimizations (Chapter 8).

7.4.3.2. SPEC92fp benchmarks

Figure 7-8(B) shows that while the impact of array privatization is minimal, array reduction analysis dramatically increases the parallel speedup of mdljsp2 and mdljdp2. Although mdljsp2, mdljdp2 and nasa7 required array privatization, only nasa7 had any visible benefit from this optimization. However, the compiler achieves good overall results in parallelizing SPEC92fp. Coverage is above 80% for 8 of the 10 programs, and we achieve speedups on all eight.

The results also show that high coverage is necessary but not sufficient for high speedups. Programs with a fine granularity of parallelism, even those with high coverage such as su2cor, tomcatv and nasa7, tend to have lower speedups. Another important factor that affects speedups is data locality. Two of these programs, tomcatv and nasa7, have poor memory behavior. We will show that the performance of these programs can be improved significantly via data and loop transformations to improve cache locality (Chapter 8), and by using techniques to minimize synchronization [140].

7.4.3.3. Nas benchmarks

We have gathered results for the Nas benchmark on two different multiprocessors using two different datasets as given in Figures 7-9 and 7-10. The results on the different multiprocessors are quite similar. The advanced array analyses in SUIF are important to the successful parallelization of the Nas benchmarks. Comparing SUIF with the baseline system, we observe that the array analyses have two important effects. Array privatization enables the compiler to locate significantly more parallelism in two of the programs, cgm and embar. Array reductions increase the granularity of parallelism in appbt, applu and appsp by parallelizing an outer loop instead of inner loops nested inside it. Observe that, in appbt with the large dataset, finding an outer loop even when the coverage is at 100% has a significant impact on performance.

The improvements in coverage and granularity in Nas translate to good speedup results. Six of the eight programs yield speedups. In both experiments, buk yields no speedup due to low coverage, which is not surprising as it implements a bucket sort algorithm. With the small dataset, applu is too fine-grained to yield any speedup. Overall, the advanced array analyses are important for Nas; more than half of the benchmark suite would not speed up without these techniques.

7.4.3.4. Perfect benchmarks

As displayed in Figure 7-11(B)-(D), the array privatization analysis significantly improves the parallelism coverage of spec77and trfd while some improvements occur in flo52, mdg and ocean. Granularity is increased for spec77 and trfd, and speedup is achieved in the case of trfd. Although little parallel speedup is observed on spec77, the improvement over the baseline system confirms the validity of our preference for outer loop parallelism. As a whole, SUIF doubles the number of programs that achieve a speedup from 2 to 4. The loops requiring array privatization in adm, bdna and dyfesm had no impact on parallel execution.

The overall parallelization of Perfect was not as successful as for the other two benchmark suites. As Figure 7-11 indicates, there are two basic problems. Half of the programs have coverage below 80%. Furthermore, the parallelism found is rather fine-grained, with most of the parallelizable loops taking less than 100 µs on a uniprocessor. In fact, had the run-time system not suppressed the parallelization of fine-grained loops in Perfect, the results would have been much worse. Thus, not only is the coverage low, the system can only exploit a fraction of the parallelism extracted.

We now examine the difficulties in parallelizing Perfect to determine the feasibility of automatic parallelization and to identify possible future research directions. We found that some of these programs are simply not parallelizable as implemented. Some of these programs contain a lot of input and output (e.g. mg3d and spec77); their speedup depends on the success of parallelizing I/O. Further, “dusty deck” features of these programs, such as the use of *equivalence* constructs in ocean, obscure information from analysis. In contrast, most of the SPEC95fp, SPEC92fp and Nas programs are cleanly implemented, and are thus more amenable to automatic parallelization.

Many of these programs, particularly ocean, adm, and mdg, have key computational loops that are safe to parallelize, but they are beyond the scope of the techniques implemented in SUIF. ocean and adm contain non-linear array subscripts involving multiplicative induction variables that are beyond the scope of the higher-order induction variable recognition algorithm in the SUIF compiler. There will always be extensions to an automatic parallelization system that can improve its effectiveness for some programs. Nonetheless, there is a fundamental limitation to static parallelization. Some programs cannot be parallelized with only compile-time information. For example, the main loop in adm is parallelizable only if the problem size, which is unknown at compile time, is even. A promising solution is to have the program check if the loop is parallelizable at run time, using dynamic information. Interprocedural analysis and optimization can play an important part in such an approach by improving the efficiency of the run-time tests. Analysis can derive highly optimized run-time tests and hoist them to less frequently executed portions of the program, possibly even across procedure boundaries. The interprocedural analysis in our system provides an excellent starting point for work in this area.

The advanced analysis can also form the basis for a useful interactive parallelization system. Even when the analyses are not strong enough to determine that a loop is parallelizable, the results can be used to isolate the problematic areas and focus the users' attention on them. For example, our compiler finds in the program qcd a 617-line interprocedural loop that would be parallelizable if not for a small procedure. Examination of that procedure reveals that it is a random number generator, which a user can potentially modify to run in parallel. By requesting very little help from the user, the compiler can parallelize the loop and perform all the tedious privatization and reduction transformations automatically.

7.5. Related Work

Previous evaluations of interprocedural parallelization systems have provided static measurements of the number of additional loops parallelized as a result of interprocedural analysis [81,84,108,138]. We have compared our results with a recent empirical study, which examines the Spec89 and Perfect benchmark suites [84]. When considering only those loops containing calls for the set of 16 programs used in that study, the SUIF system is able to parallelize more than five times as many loops [76]. The key difference between the two systems is that SUIF contains full interprocedural array analysis, including array privatization and reduction recognition.

The Polaris compiler system is also a fully implemented parallelizer using advanced analyses [24,142]. However, Polaris performs no interprocedural analysis, instead relying on full inlining of the programs to obtain interprocedural information. It is difficult to make direct comparison between the two systems. For example, optimizations such as unused procedure elimination, which eliminates some loops, and selective procedure inlining, which creates copies of some loops, make the parallel loop counts different. The latest results from the Polaris compiler can be found in [22].

7.6. Chapter Summary

We have a fully implemented interprocedural parallelizer with advanced array analyses and we have evaluated its effectiveness by parallelizing more than 115,000 lines of FORTRAN code from 39 programs in four benchmark suites. Out of 5,262 loops found in these programs we were able to parallelize more than 55%, of which 255 required advanced analyses. However, since static loop counts alone do not provide a good measurement of coarse grain parallelism, we measured the dynamic behavior of these programs.

Figure 7-12 summarizes the impact of the improvements from the advanced array analyses on coverage, granularity and speedup in the three benchmark suites. The first row contains the number of programs reported from each benchmark suite. The second row shows how many programs have their coverage increased to above 80% after adding the advanced array analyses. The third row gives the number of programs that have increased granularity (but similar coverage) as a result of the advanced array analyses. The fourth row shows how these significant improvements affect overall performance. Overall, 75% of the programs obtained parallel coverage over 80% and half the programs were able to achieve higher than 50% of the ideal speedup.

	SPEC95fp	SPEC92fp	Nas	Perfect
Total number of programs	9	10	8	12
Programs with improved coverage (> 80%)	1	3	3	1
Programs with increased granularity	1	0	2	2
Programs with improved speedup (> 50% of the perfect speedup)	2	1	5	2

Figure 7-12. Summary of the experimental results

8 Improving Memory Performance with Data Transformations

It is ideal to provide the programmer with uniform access to an unlimited amount of the fastest memory available. However, it is neither technologically feasible nor economically viable to build such machines, computer designers thus attempt to create the illusion of having uniform and fast access to memory by exploiting *locality of reference* commonly found in programs. Modern computers are designed with a hierarchical memory subsystem to take advantage of the locality of reference in programs, and deliver fast access times to a large amount of data stored in the memory system. Small amounts of fast memory or a cache are located closer to the processor to shield the programs from the high latency of the larger memory in the lower levels of the hierarchy. While these caches can provide a tremendous performance boost for the programs with locality, they are very sensitive to the memory access patterns of the programs. Many simple and common access patterns found in practice can trigger unexpected problems in the performance of the caches. Two such problems that occur in multiprocessor caches are *false sharing misses* and *cache conflict misses*. Large cache lines cause false sharing in multiprocessors. When different processors modify different data that happen to be co-located on the same cache line, the cache line bounces back and forth between the two caches. Cache conflict misses occur when accessing an array with a stride such that some accesses will be mapped to the same cache location. Although the cache may be empty, the data has to be fetched from memory every time the access is repeated.

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. This is true for bus-based shared address space machines [50,51], and even more so for scalable shared address space machines [26] such as the Stanford DASH [105] and FLASH multiprocessors [101], MIT ALEWIFE [1], Kendall Square's KSR-1 [94], the Convex Exemplar [115], and the Silicon Graphics Origin. The memory on remote processors in these architectures constitutes yet another level in the memory hierarchy. The differences in access times among cache, local, and remote memory can be very large. For example, on the DASH multiprocessor, the ratio of access times between the first-level cache, second-level cache, local memory, and remote memory is roughly 1:10:30:100. Therefore, it is important to minimize the number of accesses to all the slower levels of the memory hierarchy.

This chapter is organized as follows. In Section 8.1, we define the problem of false sharing misses and conflict misses that occur in multiprocessor caches. We propose a compiler algorithm to eliminate them in Section 8.2, by performing data transformations that will make all array elements assigned to each processor contiguous in memory. The array access functions created by the data transformations are inefficient. Therefore, in Section 8.3, we propose a set of optimizations to simplify access functions. We have implemented this algorithm and Section 8.4 evaluates its impact. We compare our approach to related works in Section 8.5. Finally, we summarize in Section 8.6.

8.1. Problem Statement

In this chapter, we focus on false sharing misses and conflict misses. We introduce a compiler algorithm to eliminate these two classes of problems by transforming data arrays in the programs [7,12].

8.1.1. False Sharing Misses

In a modern computer, data is transferred in fixed-size units known as cache lines, which are typically 4 to 128 bytes long [82]. A computation is said to have *spatial locality* if it uses multiple words in a cache line before the line is displaced from the cache. While spatial locality is common to both uni- and multiprocessors, false sharing is unique to multiprocessors. False sharing occurs when different processors use different data that happen to be co-located on the same cache line. Even if a processor re-uses a data item, the item may no longer be in the cache due to an intervening access by another processor to a different word in the same cache line.

Assuming the FORTRAN convention that arrays are allocated in column-major order, our example contains is a significant amount of false sharing, as shown in Figure 8-1. If the number of rows accessed by each processor is smaller than the number of words in a cache line, every cache line is shared by at least two processors. Each time one of these lines is accessed, unwanted data is brought into the cache. Furthermore, when one processor writes part of the cache line, that line is invalidated in the other processor's cache. This particular combination of computation mapping and data layout will result in extremely poor cache performance.

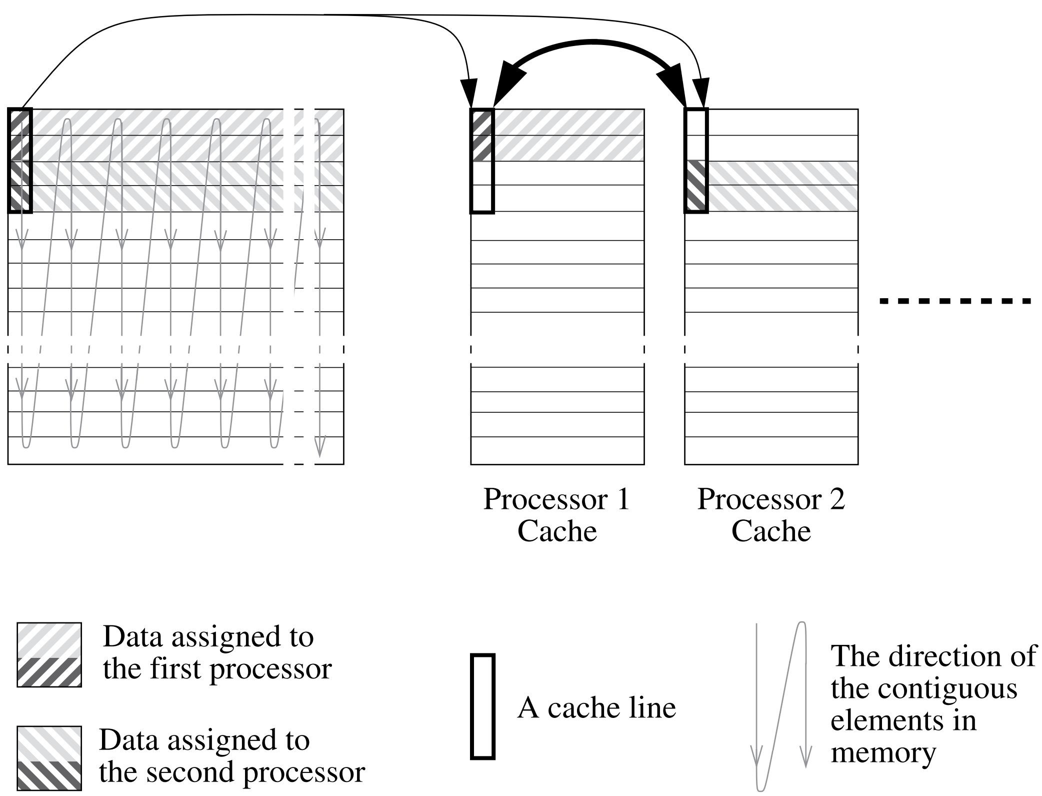


Figure 8-1. False Sharing

8.1.2. Cache Conflict Misses

Another problematic characteristic of data caches is that they typically have a small set-associativity; that is, each memory location can be cached only in a small number of cache locations. Conflict misses occur whenever different memory locations contend for the same cache location. Since each processor only operates on a subset of the data, the addresses accessed by each processor may be distributed throughout the shared address space.

Consider what happens to the example in Figure 8-2. If the arrays are of size 1024×1024 and the target machine has a direct-mapped cache of size 64KB. Assuming that REALs are

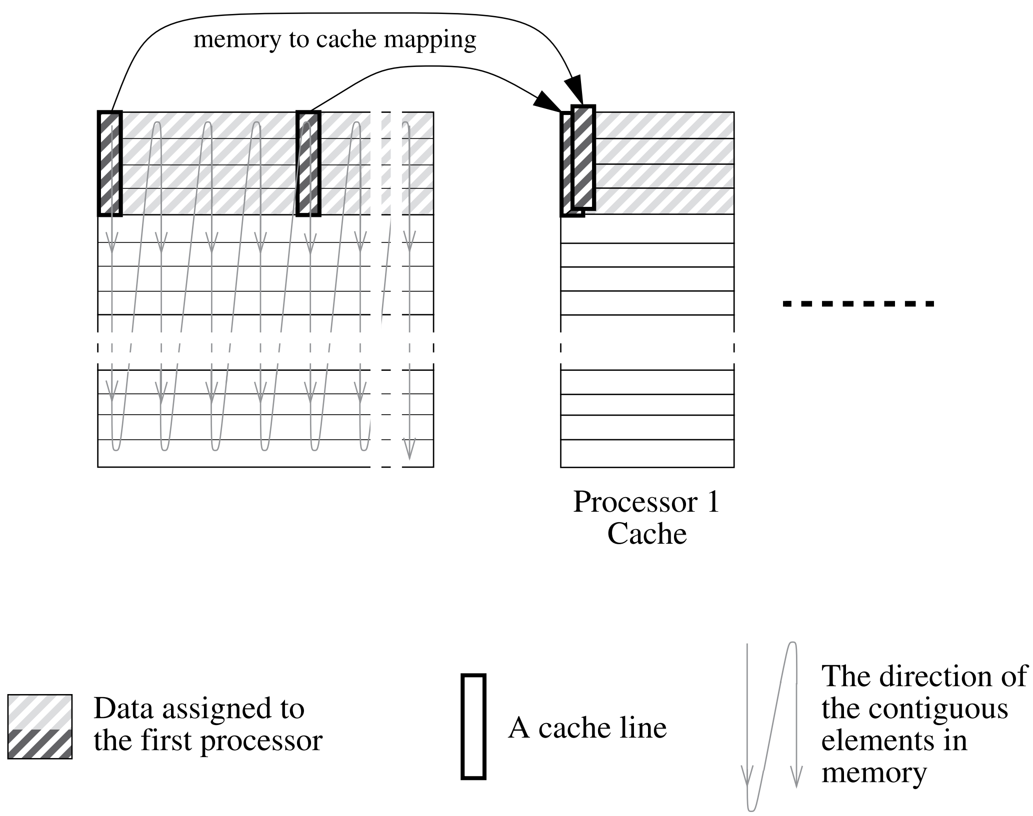


Figure 8-2. Cache Conflicts

4B long, the elements in every 16-th column will map to the same cache location and cause conflict misses. This problem exists even if the caches are set associative.

8.2. Data Transformations

As shown in the previous section, the cache performance on a multiprocessor depends on the pattern of the data layout in memory. Instead of simply obeying the data layout convention used by the input language (e.g. column-major in FORTRAN and row-major in C), we can improve the cache performance by customizing the data layout for a specific program. We observe that multiprocessor cache performance problems can be minimized by making the data accessed by each processor contiguous in the shared address space, an example of which is shown in Figure 8-3. Such a layout enhances spatial locality, minimizes false sharing and also minimizes conflict misses.

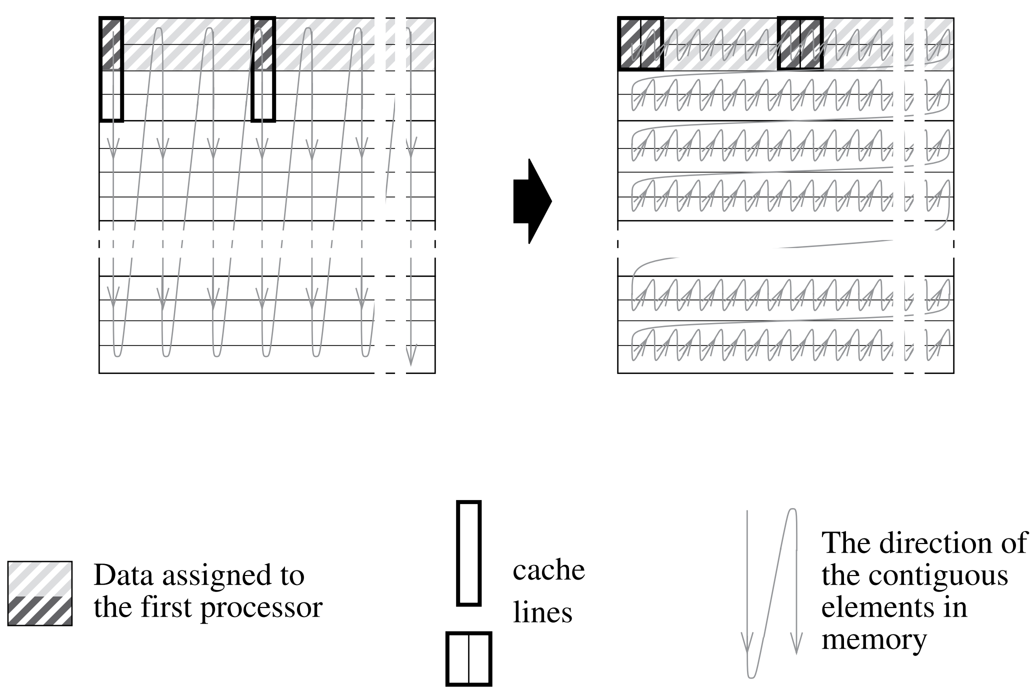


Figure 8-3. Making data accessed by each processor contiguous in memory

The importance of optimizing memory subsystem performance for multiprocessors has also been confirmed by several studies of hand optimizations on real applications. Singh et al. explored performance issues on scalable shared address space architectures; they improved cache behavior by transforming two-dimensional arrays into four-dimensional arrays so that each processor's local data are contiguous in memory [132]. Torrellas et al. [135] and Eggers et al. [50,51] also showed that improving spatial locality and reducing false sharing resulted in significant speedups for a set of programs on shared-memory machines.

8.2.1. Data Transformation Model

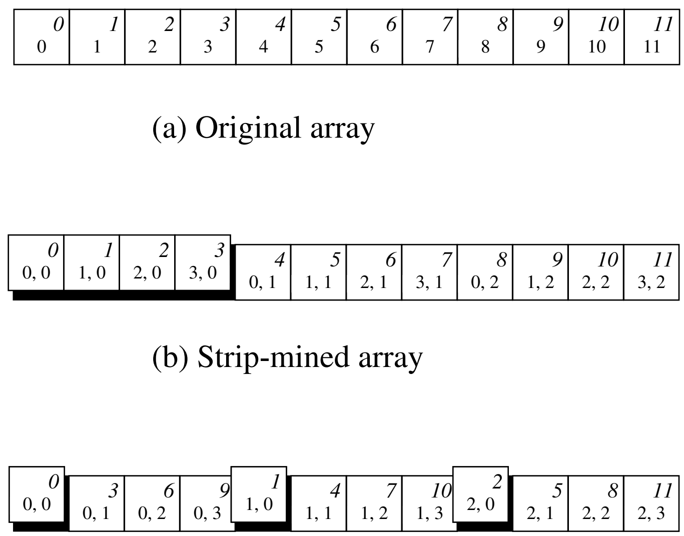
To facilitate the design of our data layout algorithm, we have developed a data transformation model that is analogous to the well-known loop transformation theory [20,148]. We represent an n-dimensional array as an n-dimensional polytope whose boundaries are given by the array bounds, and the interior integer points represent all the elements in the array. As with sequential loops, the ordering of the axes is significant. In the rest of this chapter, we assume the FORTRAN convention of column-major ordering by default. For clarity the array dimensions are 0-based, which means that for an n-dimensional array with array bounds $d_1 \times d_2 \times \ldots \times d_n$, the linearized address for array element $(i_1, \ldots, i_n)$ is

$$((\ldots((i_n \times d_{n-1} + i_{n-1}) \times d_{n-2} + i_{n-2}) \times \ldots + i_3) \times d_2 + i_2) \times d_1 + i_1 \cdot$$

Next, we introduce two primitives, *strip-mining* and *permutation*, that are used in combination to perform the data transformations.

8.2.1.1. Strip-mining primitive

Strip-mining an array dimension re-organizes the original data in that dimension as a two-dimensional structure. For example, strip-mining a one-dimensional, d-element array with strip size b turns the array into a $b \times \left\lceil \frac{d}{b} \right\rceil$ array. Figure 8-4(a) shows the data in the original array, and Figure 8-4(b) shows the new indices in the strip-mined array. The first column of this strip-mined array is highlighted in the figure. The number in the upper right corner of each square shows the linear address of the data item in the new array. The i-th element in the original array now has coordinates $\left(i \bmod b, \left\lfloor \frac{i}{b} \right\rfloor\right)$ in the strip-mined array. Given



(a) Original array

(b) Strip-mined array

(c) The final array after permutation

Figure 8-4. The indices of array accesses at each stage of transformation. The number in the upper right corner shows the linearized address of the data.

that block sizes are positive, with the assumption that arrays are 0-based, we can replace the floor operators in array access functions with integer division assuming truncation. The address of the element in the linear memory space is $\frac{i}{b} \times b + i \bmod b = i$. Strip-mining, on its own, does not change the layout of the data in memory. It must be combined with permutation transformations to have an effect.

8.2.1.2. Permutation primitive

A permutation transform T maps an n-dimensional array space to another n-dimensional space; that is, if $\vec{i}$ is the original array index vector, the transformed array indices $\vec{j}$ is $\vec{j} = T\vec{i}$. The array bounds must also be transformed similarly.

For example, an array transpose maps (i_1, i_2) to (i_2, i_1). Using matrix notation this becomes

$$\begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} i_1 \\ i_2 \end{bmatrix} = \begin{bmatrix} i_2 \\ i_1 \end{bmatrix}$$

The result of transposing the array in Figure 8-4(b) is shown in Figure 8-4(c). Figure 8-4(c) shows the data in the original layout. Each item is labeled with its new indices in the transposed array in the center. The new linearized address is in the upper right corner. As highlighted in the diagram, this example shows how a combination of strip-mining and permutation can make every fourth data element in a linear array contiguous to each other. In a cyclically distributed array, this could be used to make each processor's share of data contiguous.

In theory, we can generalize permutations to other unimodular transforms. For example, rotating a two-dimensional array by 45 degrees makes data along a diagonal contiguous, which may be useful if a loop accesses the diagonal in consecutive iterations. There are two plausible ways of laying the data out in memory. The first is to embed the resulting parallelogram in the smallest enclosing rectilinear space, and the second is to simply place the diagonals consecutively, one after the other. However, the former consumes an excessive amount of storage, and the latter generates complex address calculations. Furthermore, we do not expect unimodular transforms other than permutations to be important in practice. Thus, we have not implemented general unimodular data transformations.

8.2.2. Legality

Unlike loop transformations, which must satisfy data dependences, any combinations of strip-mining and permutation is a valid data transformation. On the other hand, loop transforms have the advantage that they affect only one specific loop nest; performing an array data transform requires that all accesses to the array in the entire program use the new layout. Current programming languages such as C and FORTRAN have features that can make these transformations difficult. The compiler cannot restructure an array unless it can guarantee that all possible accesses of the same data can be updated accordingly. For example, in FORTRAN, the storage for a common block array in one procedure can be re-used to form a completely different set of data structures in another procedure. In C, pointer arithmetic and type casting can prevent data transformations.

8.2.3. Algorithm Overview

The data transformation algorithm uses data decompositions that are either provided by the programmer using a language such as HPF (High Performance FORTRAN) [83] or automatically generated by a compiler algorithm [2,13,16,21,68,107,129]. Figure 8-5 is an example of a data decomposition specification using the HPF language. In the example, the array A is mapped to a two-dimensional processor grid by two-dimensional blocks using the template T. The HPF decomposition format allows the distribution of each dimension of an array to be independently specified. Each dimension of an array can be either allocated to the same processor (denoted by *), or distributed in a *block, cyclic,* or *block-cyclic* manner. Our algorithm supports data decompositions provided in the HPF decomposition format.

```
DIMENSION A(N, N)
!HPF$ PROCESSORS P(2, 2)
!HPF$ TEMPLATE T(N, N)
!HPF$ DISTRIBUTE T(BLOCK, BLOCK) ONTO P
!HPF$ ALIGN A(I, J) WITH T(I, J)
....
```

Figure 8-5. Example array declaration in HPF

Given the HPF data decompositions, many equivalent memory layouts make each processor's data contiguous in the shared address space. Our current implementation simply retains the original data layout as much as possible. That is, all the data accessed by the same processor maintain the original relative ordering. We expect this compilation phase to be followed by another algorithm that analyzes the computation executed by each processor and improves the cache performance by reordering data and operations on each processor [33,61,147].

Next, we present three examples of data transformations applied to a two-dimensional array with $(\mathrm{BLOCK}, \mathrm{BLOCK})$, $(\mathrm{CYCLIC}, *)$ and $(\mathrm{CYCLIC}(b), *)$ decompositions. These examples illustrate how we apply the permutation and strip-mining primitives to transform the data such that array elements assigned to each processor are contiguous in memory.

8.2.3.1. Example of a two-dimensional block distribution

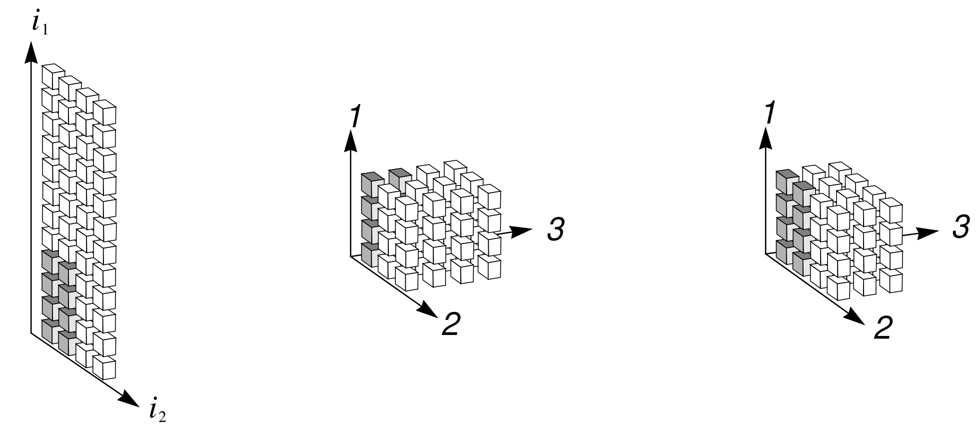
This example illustrates the transformation process of a $(\mathrm{BLOCK}, \mathrm{BLOCK})$ distributed two-dimensional array. The steps of the transformation process are given in Figure 8-6. Figure 8-6(a) is a graphical representation of the memory layout of the array where the elements assigned to the first processor are highlighted. The access functions and the dimension sizes are given by Figures 8-6(b) and 8-6(c) respectively. We assume that the dimensions of the array are $d_1 \times d_2$, and $P_1 \times P_2$ are the number of processors in the processor grid. In the first step of the transformation, the inner dimension is strip-mined with a strip size of $\left\lceil \frac{d_1}{P_1} \right\rceil$. The identifier of the processor owning the data is specified by the second of the strip-mined dimensions. Then, the three dimensions are permuted such that the processor dimension is made outermost.

We further demonstrate this transformation by using a 12×4 array, given in Figure 8-7(a), where the elements are distributed to six processors on a 3×2 grid. The offset in the memory is given in the upper right corner of each element and the elements assigned to the fist processor are highlighted. In the transformed array, in Figure 8-7(b), the elements of the first processor are now assigned to contiguous locations in memory.

8.2.3.2. Example of a cyclic distribution

The next example is the transformation of a two-dimensional array distributed in a $(\mathrm{CYCLIC}, *)$ manner. The steps of the transformation process are given in Figure 8-6.

(a) Memory Layout



(b) Access Functions

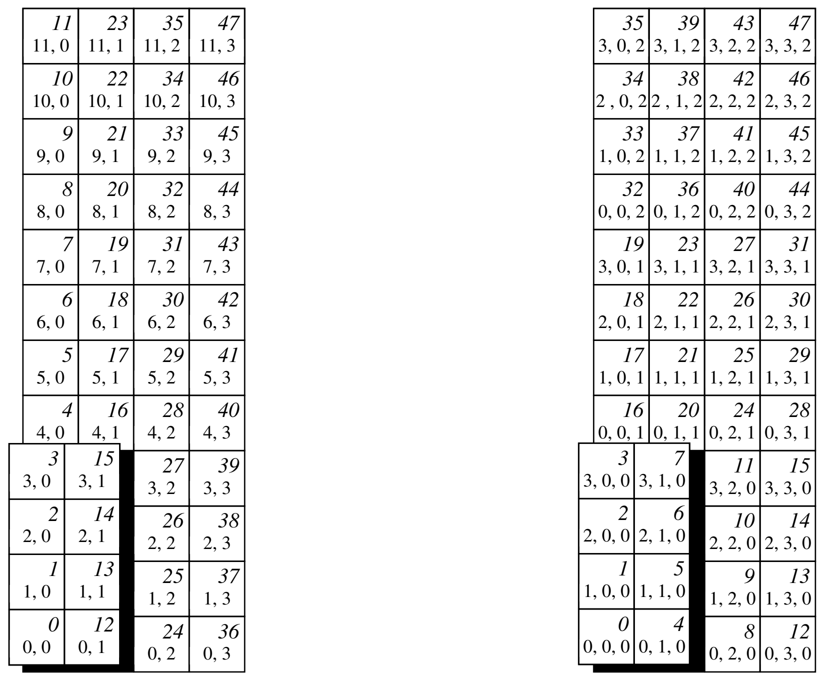
$$\begin{bmatrix} i_1 \\ i_2 \end{bmatrix} \qquad \begin{bmatrix} i_1 \bmod \left\lceil \dfrac{d_1}{P_1} \right\rceil \\ i_1 / \left\lceil \dfrac{d_1}{P_1} \right\rceil \\ i_2 \end{bmatrix} \qquad \begin{bmatrix} i_1 \bmod \left\lceil \dfrac{d_1}{P_1} \right\rceil \\ i_2 \\ i_1 / \left\lceil \dfrac{d_1}{P_1} \right\rceil \end{bmatrix}$$

(c) Array Bounds

$$\begin{pmatrix} d_1 \\ d_2 \end{pmatrix} \qquad \begin{pmatrix} \left\lceil \dfrac{d_1}{P_1} \right\rceil \\ P_1 \\ d_2 \end{pmatrix} \qquad \begin{pmatrix} \left\lceil \dfrac{d_1}{P_1} \right\rceil \\ d_2 \\ P_1 \end{pmatrix}$$

Figure 8-6. Transformation process of an array with (BLOCK, BLOCK) distribution

The inner dimension is first strip-mined with a strip size of P, where P is the number of processors. The identifier of the processor owning the data is specified by the first of the strip-mined dimensions. Next, the three dimensions are permuted such that the processor dimension is made outermost.



(a) before transformation (b) after transformation

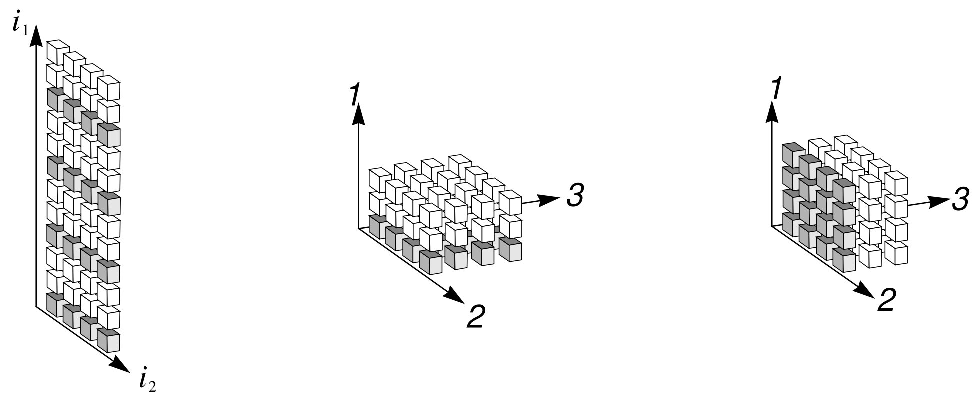
Figure 8-7. A (BLOCK, BLOCK) distributed array before and after transformations

We expand on the example in Figure 8-9(a), where a 12×4 array is distributed on three processors. The transformed array, in Figure 8-9(b), has elements of the first processor assigned to contiguous locations in memory.

8.2.3.3. Example of a block-cyclic distribution

We illustrate a more complex transformation where a two-dimensional array with a $(\text{CYCLIC}\,(b), *)$ decomposition is made contiguous in memory. The transformation can be represented as two strip-mining operations and a permutation, as shown in Figure 8-10. In Figure 8-10(a), the four-dimensional data arrays in the last two steps are shown as a flat

(a) Memory Layout



(b) Access Functions

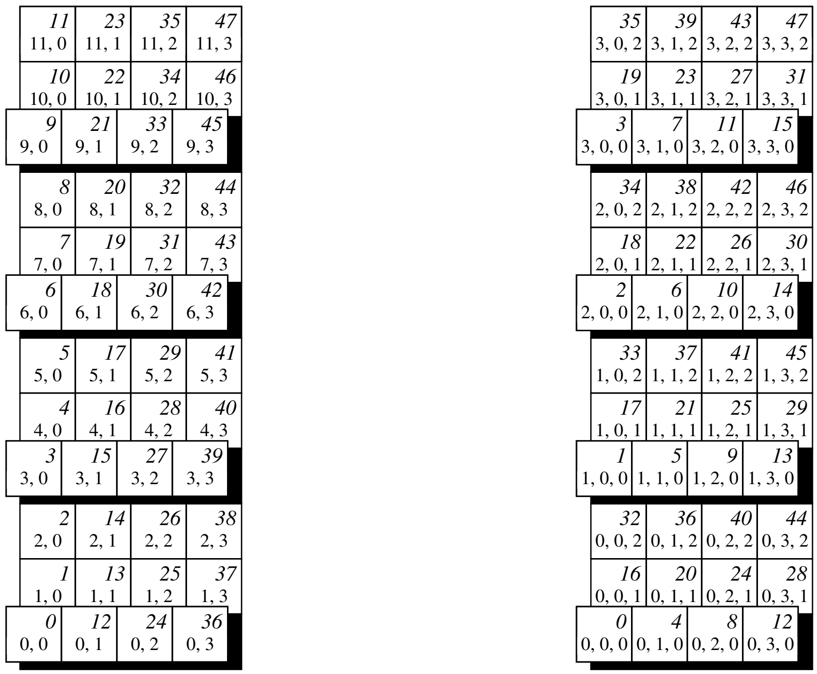
$$\begin{bmatrix} i_1 \\ i_2 \end{bmatrix} \qquad \begin{bmatrix} i_1 \bmod P \\ i_1 / P \\ i_2 \end{bmatrix} \qquad \begin{bmatrix} i_1 / P \\ i_2 \\ i_1 \bmod P \end{bmatrix}$$

(c) Array Bounds

$$\begin{pmatrix} d_1 \\ d_2 \end{pmatrix} \qquad \begin{pmatrix} P \\ \left\lceil \frac{d_1}{P} \right\rceil \\ d_2 \end{pmatrix} \qquad \begin{pmatrix} \left\lceil \frac{d_1}{P} \right\rceil \\ d_2 \\ P \end{pmatrix}$$

Figure 8-8. Transformation process of an array with (CYCLIC, *) distribution

structure where the inner two dimensions are represented by a tile and the outer two dimensions are the placement of these tiles. The inner dimension is first strip-mined with strips of size of b. Then, the second of the strip-mined dimensions is strip-mined again with a strip size of P, where P is the number of processors. The identifier of the processor owning the data is specified by the first strip-mined dimension of the latter strip-mining step. Thus, the four dimensions are permuted such that this processor dimension is made outermost.



(a) before transformation (b) after transformation

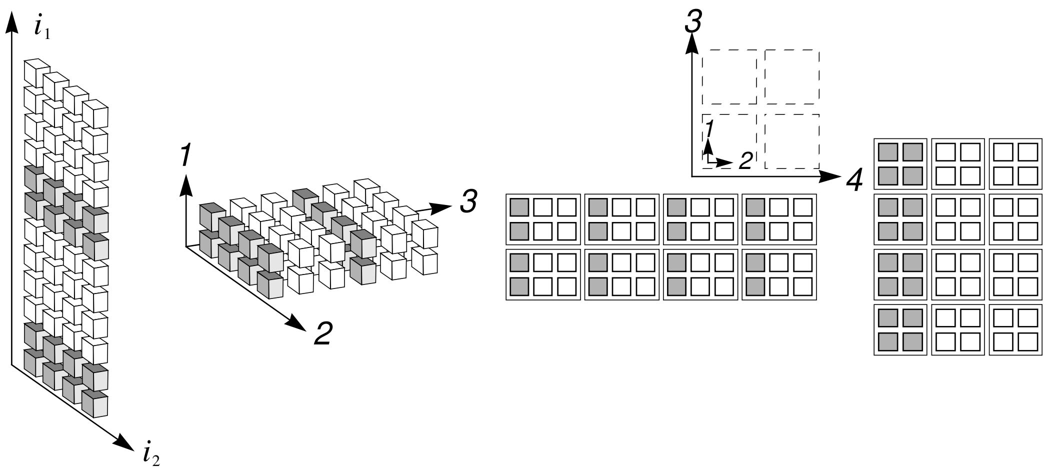
Figure 8-9. A (CYCLIC, *) distributed array before and after transformations

The 12×4 array, given in Figure 8-11(a), is distributed on three processors using the $(\mathrm{CYCLIC}(2), *)$ decomposition. After the transformation, as shown in Figure 8-11(b), the elements of the first processor are contiguous in memory.

8.2.4. Data Transformation Algorithm

We have developed a data transformation algorithm that will change the data layout in memory such that array elements assigned to each processor are made contiguous in memory. The algorithm uses the HPF decomposition information of the array to create the new array dimensions and array access functions. Figure 8-12 gives the algorithm for trans-

(a) Memory Layout



(b) Access Functions

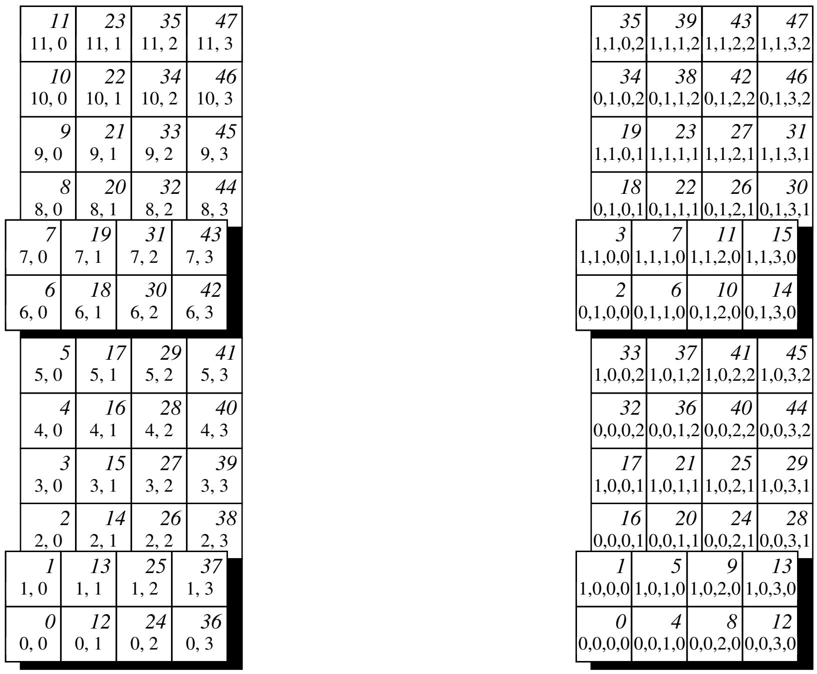
$$\begin{bmatrix} i_1 \\ i_2 \end{bmatrix} \quad \begin{bmatrix} i_1 \bmod b \\ i_1 / b \\ i_2 \end{bmatrix} \quad \begin{bmatrix} i_1 \bmod b \\ (i_1/b) \bmod P \\ i_1/(Pb) \\ i_2 \end{bmatrix} \quad \begin{bmatrix} i_1 \bmod b \\ i_1/(Pb) \\ i_2 \\ (i_1/b) \bmod P \end{bmatrix}$$

(c) Array Bounds

$$\begin{pmatrix} d_1 \\ d_2 \end{pmatrix} \quad \begin{pmatrix} b \\ \left\lceil \frac{d_1}{b} \right\rceil \\ d_2 \end{pmatrix} \quad \begin{pmatrix} b \\ P \\ \left\lceil \frac{d_1}{Pb} \right\rceil \\ d_2 \end{pmatrix} \quad \begin{pmatrix} b \\ \left\lceil \frac{d_1}{Pb} \right\rceil \\ d_2 \\ P \end{pmatrix}$$

Figure 8-10. Transformation process of an array with a (CYCLIC(b), *) distribution

forming array dimensions. In this presentation, the innermost dimension of the array is given by index 1, while higher indices indicate outer dimensions. Furthermore, p indicates the number of processor dimensions inserted into the array, and P_p is the number of processors in the p-th processor dimension. The new array dimensions are calculated by



(a) before transformation

(b) after transformation

Figure 8-11. A (CYCLIC(2), *) distributed array before and after transformations

applying transformations on each dimension with a block, cyclic, or block-cyclic decomposition. Each block or cyclic dimension is sub-divided into two dimensions, and the processor dimension is moved outermost. Each dimension with a block-cyclic decomposition is subdivided into three dimensions. Again, the processor dimension is moved outermost. The transformation applied to each of the dimensions corresponds to a combination of strip-mining and permutation transformations, as described in the algorithm overview section. The algorithm to transform accesses to the distributed array, given in Figure 8-13, is similar to the previous algorithm. The new array access function is calculated by applying

$NewDimsize((x_1, \ldots, x_n), (d_1, \ldots, d_n)) \rightarrow (d'_1, \ldots)$

where $\underset{1 \le k \le n}{\forall} x_k \in \{*, \text{BLOCK}, \text{CYCLIC}, \text{CYCLIC}(b)\}$, b is the block size, and $(d_1, \ldots, d_n)$ are array dimension sizes.

$D = (d_1, \ldots, d_n)$

$p = 1$

for $k = n$ **down to** 1 **do**

Let $D = (\ldots, d_{k-1}, d_k, d'_{k+1}, \ldots, d'_m)$ where $d'_{k+1}, \ldots, d'_m$ are updated dimension sizes

if (x_k = BLOCK and $k < n$)

or x_k = CYCLIC **then**

$$D = \left(\ldots, d_{k-1}, \left\lceil \frac{d_k}{P_p} \right\rceil, d'_{k+1}, \ldots, d'_m, P_p\right)$$

$p = p + 1$

if x_k = CYCLIC (b) **then**

$$D = \left(\ldots, d_{k-1}, b, \left\lceil \frac{d_1}{P_p b} \right\rceil, d'_{k+1}, \ldots, d'_m, P_p\right)$$

$p = p + 1$

return D

Figure 8-12. Algorithm for calculating new array dimensions

a combination of strip-mining and permutation transformations to each dimension with a block, cyclic, or block-cyclic decomposition.

We have made one minor local optimization to our algorithm. If the highest dimension of the array is distributed as BLOCK, no permutation is necessary since the processor dimension is already in the outermost position; thus no strip-mining is necessary either since, as discussed above, strip-mining on its own does not change the data layout.

$NewIndex((x_1, \ldots, x_n), (i_1, \ldots, i_n), (d_1, \ldots, d_n)) \rightarrow (i'_1, \ldots)$
where $\underset{1 \le k \le n}{\forall} x_k \in \{*, \text{BLOCK}, \text{CYCLIC}, \text{CYCLIC}(b)\}$, b is the block size $(i_1, \ldots, i_n)$ are array index functions and $(d_1, \ldots, d_n)$ are array dimension sizes.

$$I = (i_1, \ldots, i_n)$$

$$p = 1$$

for $k = n$ **down to** 1 **do**

Let $I = (\ldots, i_{k-1}, i_k, i'_{k+1}, \ldots, i'_m)$ where $i'_{k+1}, \ldots, i'_m$ are updated array index functions

if $x_k = \text{BLOCK}$ **and** $k < n$ **then**

$$I = \left(\ldots, i_{k-1}, i_k \bmod \left\lceil \frac{d_k}{P_p} \right\rceil, i'_{k+1}, \ldots, i'_m, \frac{i_k}{\left\lceil \frac{d_k}{P_p} \right\rceil}\right)$$

$$p = p + 1$$

if $x_k = \text{CYCLIC}$ **then**

$$I = \left(\ldots, i_{k-1}, \frac{i_k}{P_p}, i'_{k+1}, \ldots, i'_m, i_k \bmod P_p\right)$$

$$p = p + 1$$

if $x_k = \text{CYCLIC}(b)$ **then**

$$I = \left(\ldots, i_{k-1}, i_k \bmod b, \frac{i_k}{P_p b}, i'_{k+1}, \ldots, i'_m, \left(\frac{i_k}{b}\right) \bmod P_p\right)$$

$$p = p + 1$$

return I

Figure 8-13. Algorithm for calculating new array indices

Our current implementation is restricted to handling only the decompositions that map a single array dimension to one processor dimension. Handling general affine decompositions is a straightforward extension to our algorithm. However, they rarely occur in practice, and the corresponding data transformations would result in complex array access functions.

8.2.5. Code Generation

We illustrate code generation after data transformations using the example program segment in Figure 8-14. The data transformation of the array A is the same as the example in Section 8.2.3.2. After applying the transformations, the code segment is given in Figure 8-15.

```
DIMENSION A(N, N)
!HPF$ PROCESSORS P(nproc)
!HPF$ TEMPLATE T(N, N)
!HPF$ DISTRIBUTE T(CYCLIC, *) ONTO P
!HPF$ ALIGN A(I, J) WITH T(I, J)

      DO J = 2, 99
            DO I = lb, ub
                  A(I, J) = ...
            ENDDO
      ENDDO
```

Figure 8-14. Example program segment

The exact dimensions of a transformed array often depend on the number of processors, which may not be known at compile time. For example, if P is the number of processors and d is the size of the dimension, the strip sizes used in CYCLIC and BLOCK distribu-

```
DIMENSION A(0:(N+nproc-1)/nproc, N, 0:nproc)

      DO J = 2, 99
            DO I = lb, ub
                  A((I-1)/nproc+1, J, mod(I-1,nproc)+1) = ...
            ENDDO
      ENDDO
```

Figure 8-15. Program segment after data transformation

tions are P and $\left\lceil \frac{d}{P} \right\rceil$, respectively. As discussed above, strip-mining a d-element array dimension with strip size b produces a subarray of size $b \times \left\lceil \frac{d}{b} \right\rceil$. This total size can be greater than d, but is always less than $d + b - 1$. We can still allocate the array statically provided that we can bound the value of the block size. If b_{max} is the largest possible block size, we simply need to add $b_{max} - 1$ elements to the original dimension.

8.3. Modulo and Division Optimization

Producing the correct array index functions for transformed arrays is straightforward. However, the modified index functions now contain modulo and division operations; if these operations are performed on every array access, the overhead will be much greater than any performance gained by improved cache behavior. In this section, we introduce a set of optimizations that eliminates most of the modulo and division instructions that are introduced by the data transformation algorithm. In performing these optimizations, we exploit fundamental properties of these operations, as well as the specialized knowledge the compiler has about these address calculations. We also use simple extensions to standard compiler techniques such as loop invariant removal and induction variable recognition to move some of the division and modulo operators out of inner loops [3]. The optimizations, described below, have proved to be important and effective in practice.

8.3.1. Modulo and division simplification

We exploit fundamental properties of modulo and division operations [63] to simplify expressions with these operations. Figure 8-16 is the list of simplifications performed on expressions with modulo and division operations by the compiler. In the list, f, f_1, f_2, g are expressions and c, c_1, c_2, d are integers. The modulo operation is denoted by %.

.

$$(f_1 g + f_2)\ \%\ g \quad \Rightarrow \quad f_2\ \%\ g$$

$$(f_1 g + f_2) / g \quad \Rightarrow \quad f_1 + f_2 / g$$

$$(c_1 f_1 + c_2 f_2)\ \%\ d \quad \Rightarrow \quad ((c_1\ \%\ d) f_1 + (c_2\ \%\ d) f_2)\ \%\ d$$

$$(c_1 f_1 + c_2 f_2) / d \quad \Rightarrow \quad ((c_1\ \%\ d) f_1 + (c_2\ \%\ d) f_2) / d + (c_1 / d) f_1 + (c_2 / d) f_2$$

$$(c f_1 g + f_2)\ \%\ dg \quad \Rightarrow \quad ((c\ \%\ d) f_1 g + f_2)\ \%\ dg$$

$$(c f_1 g + f_2) / (dg) \quad \Rightarrow \quad ((c\ \%\ d) f_1 g + f_2) / (dg) + (c / d) f_1$$

If $0 \le f < g$ and $g > 0$

$$f\ \%\ g \quad \Rightarrow \quad f$$

$$f / g \quad \Rightarrow \quad 0$$

If $-g \le f < 0$ and $g > 0$

$$f\ \%\ g \quad \Rightarrow \quad g + f$$

$$f / g \quad \Rightarrow \quad -1$$

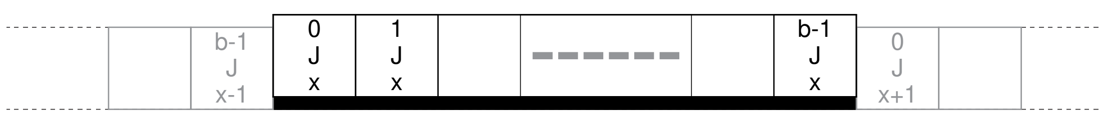
Figure 8-16. List of algebraic simplifications performed by the compiler on expression with modulo and division operations

8.3.2. Optimizing when data within the strip is accessed

When eliminating modulo and division instructions, we take advantage of the fact that a processor often addresses only elements within a single strip-mined partition of the array. An example of such an SPMD loop is given in Figure 8-17(a). Figure 8-17(b) shows the access functions of the array and highlights the elements accessed in executing the loop by the processor *x*. By formulating a problem within the framework of linear inequalities, the

```
DO I = b*myid+1, min(b*myid+b,100)
      A(mod(I-1,b), J, (I-1)/b) = ...
ENDDO
```

(a) Original loop



(b) Elements accessed by executing the loop in the processor *x*

```
idiv = myid
imod = 0
DO I = b*myid+1, min(b*myid+b,100)
      A(imod, J, idiv) = ...
      imod = imod + 1
ENDDO
```

(c) Loop after optimization

Figure 8-17. Optimize when the loop is accessing only a single strip of the array compiler can determine that in the range of $b \times myid + 1$ to $min\ (b \times myid + b, 100)$ the expression $(i - 1) / b$ is constant, and $mod\ (i - 1, b)$ is a linear expression. Thus, we can eliminate the modulo and division operations and generate the more efficient code given in Figure 8-17(c).

8.3.3. Optimizing when data in single strip is accessed after cyclic distribution

We can also simplify the access functions when a processor accesses array elements in a single strip using a loop with a non-unit step size. An example SPMD loop is given in Figure 8-18(a) and the elements accessed by a processor are illustrated in Figure 8-18(b). Within the iterations of the loop nest executed by each processor, the function $mod\ (i - 1, b)$ is constant and $(i - 1) / b$ is incremented by one. Thus, we are able to eliminate the modulo and division operations and generate code given in Figure 8-18(c).

8.3.4. Optimizing when data in a strip and its neighbors are accessed

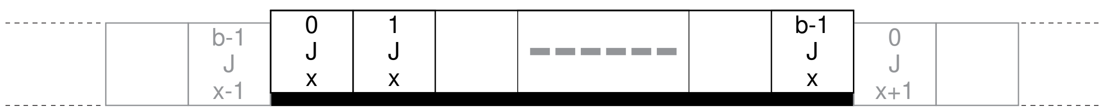
It is more difficult to eliminate modulo and division operations when the data accessed in a loop crosses the boundaries of strip-mined partitions. We optimize the cases where only the first or last few iterations cross such a boundary, as in the example loop in Figure 8-19(a). Figure 8-19(b) shows the array elements in two neighboring strips accessed in a single processor by the read access. Within most of the iterations of the loop nest, the function $(i - 1) / b$ is constant, and the function $mod\ (i - 1, b)$ is continuous. We simply peel off those iterations and apply the optimization given in Section 8.3.2. The program segment after the optimizations appears in Figure 8-19(c).

8.3.5. Optimizing when access is by a sequential loop

An array is distributed across processors mainly due accesses from parallel loops. In the previous optimizations, we eliminate the modulo and division instructions in the accesses of parallel loops. However, there can be other sequential loop nests that access the same transposed data arrays. The modulo and division operations in such access functions can adversely affect the execution of these loop nests. One such example loop is given in Figure 8-20(a), where the access pattern is shown in Figure 8-20(b). For these loops, we

```
DO I = lb to ub step P
      A((I-1)/P, J, mod(I-1,P)) = ...
ENDDO
```

(a) Original loop



(b) Elements accessed by executing the loop in the processor *x*

```
idiv = (lb-1)/P
imod = mod(lb-1,P)
DO I = lb to ub step P
      A(idiv, J, imod) = ...
      idiv = idiv + 1
ENDDO
```

(c) Loop after optimization

Figure 8-18. Optimize when a loop with a step size access a single strip of the array

enable other optimizations by strip-mining the loop. Figure 8-20(c) illustrates the results of the strip-mining and the subsequent optimizations of the example.

```
DO I = max(2,b*myid+1), min(b*myid+b,100)
      A(mod(I-1,b), J, (I-1)/b) = A(mod(I-2,b), J, (I-2)/b)
ENDDO
```

(a) Original loop

(b) Elements accessed by executing the loop in the processor *x*

```
idiv = myid
imod = 0
IF(max(2,b*myid+1) <= min(b*myid+1,100)) THEN
      A(imod, J, idiv) = A(b, J, idiv-1)
ENDDO
imod = imod + 1
DO I = b*myid+2, min(b*myid+b,100)
      A(imod, J, idiv) = A(imod-1, J, idiv)
      imod = imod + 1
ENDDO
```

(c) Loop after optimization

Figure 8-19. Optimize when the loop is accessing two neighboring strip of the array.

```
DO I = 1, 100
      A(mod(I-1,b), J, (I-1)/b) = ...
ENDDO
```

(a) Original loop

(b) Elements accessed by the execution of the loop

```
DO t = 0, ceiling(100/b)
      imod = 0
      DO I = b*t+1, min(b*t+b,100)
            A(imod, J, t) = ...
            imod = imod + 1
      ENDDO
ENDDO
```

(c) Loop after optimization

Figure 8-20. Optimize when the loop is accessing multiple strips

8.3.6. Extended strength reduction optimization

Finally, when all the other optimizations have failed to eliminate modulo and division instructions from the inner loop, we use a technique akin to strength reduction. This optimization is applicable when the modulo operation is an affine expressions of the loop index. Any division operations sharing the same operands can also be optimized along with the modulo operation. In each iteration through the loop, we increment the modulo operand. Only when the result is found to exceed the modulus must we perform the modulo and the corresponding division operations. Consider the example in Figure 8-21(a). Combining the optimization described with the additional information in this example that the modulus is a multiple of the stride, we obtain the more efficient code shown in Figure 8-21(b).

```
DO J = a, b
      A(mod(4*J+c,64), (4*J+c)/64) = ...
ENDDO
```

(a) Original loop

```
jmodst = mod(c,4)
jmod = mod(4*a+c,64)
jdiv = (4*a+c)/64
DO J = a, b
      A(jmod, jdiv) = ..
      jmod = jmod + 4
      IF(jmod >= 64) THEN
            jmod = jmodst
            jdiv = jdiv + 1
      ENDIF
ENDDO
```

(b) Loop after optimization

Figure 8-21. Optimize using strength reduction

8.4. Evaluation

We have implemented the data transformation algorithm and modulo and division optimizations on the SUIF compiler infrastructure [6,133,144]. We evaluate the usefulness of our techniques by applying them to a set of benchmark programs and executing them on a cache-coherent NUMA architecture.

8.4.1. Experimental Setup

The inputs to the SUIF compiler are either sequential FORTRAN or C programs. The output is a parallelized C program that contains calls to a portable run-time library. The parallelized program is compiled on the parallel machine using the native C compiler.

Our target machine is the Stanford DASH multiprocessor. DASH has a cache-coherent NUMA architecture. The machine we used for our experiments, described in Figure 8-22, consists of 32 processors, organized into 8 clusters of 4 processors each. Each processor is a 33MHz MIPS R3000, that has a 64KB first-level cache and a 256KB second-level cache. Both the first- and second-level caches are direct-mapped and have 16B lines. Each cluster has 28MB of main memory. A directory-based protocol is used to maintain cache coherence across clusters. It takes a processor 1 cycle to retrieve data from its first-level cache, about 10 cycles from its second-level cache, 30 cycles from its local memory and 100-130 cycles from a remote memory. The DASH operating system allocates memory to clusters at the page level. The page size is 4KB and pages are allocated to the first cluster that touches the page. We compiled the C programs produced by SUIF using gcc version 2.5.8 at optimization level -O3.

To focus on the memory hierarchy issues, our benchmark suite includes only those programs that exhibit a significant amount of parallelism. Several of these programs were identified as having memory performance problems in a simulation study [136]. We compiled each program under each of the methods described below. The compiler steps are given in Figure 8-23. We plot the speed up of the parallelized code on the DASH machine. All speedups are calculated over the best sequential version of each program.

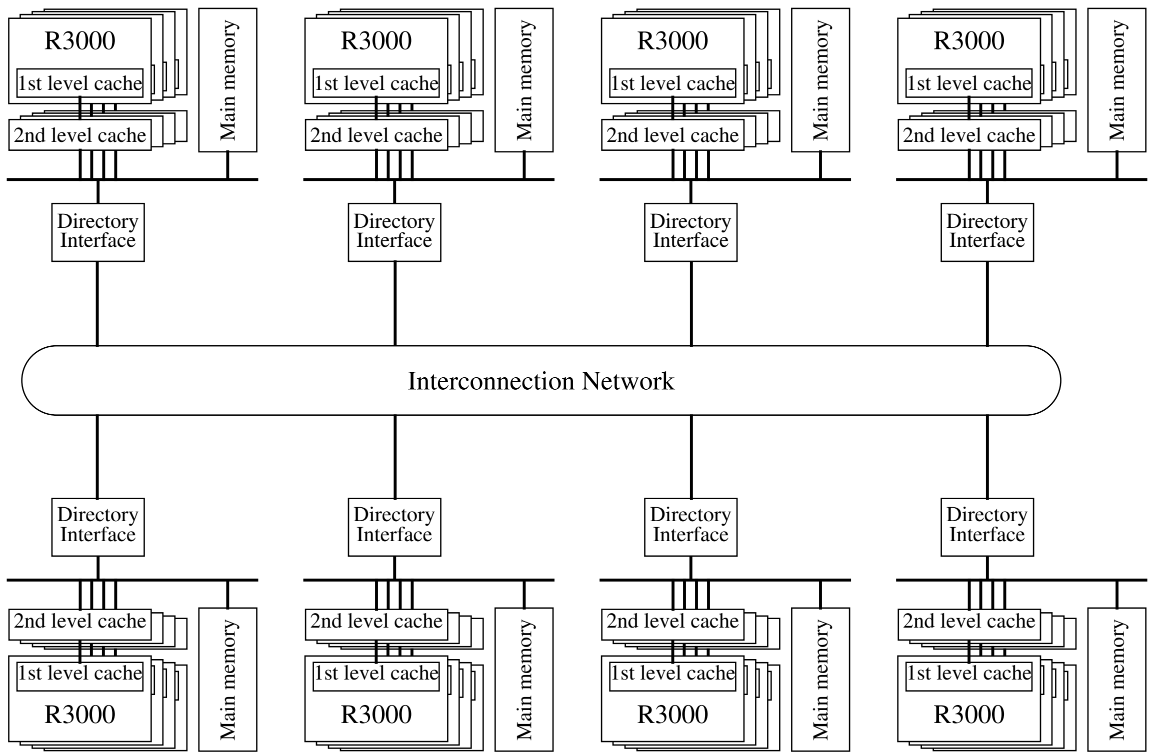


Figure 8-22. 32 node DASH multiprocessor

a) **Par**: We compiled the program with the basic parallelizer pass in the SUIF system. This parallelizer has capabilities similar to a traditional shared-memory compiler. It has a loop optimizer that applies unimodular transformations to one loop at a time to expose outermost loop parallelism and to improve data locality among the accesses within the loop [147,148].

b) **CompDecomp**: We first applied the basic parallelizer to analyze the individual loops, then applied a compiler algorithm to find the computation and the corresponding data decompositions that minimize communication across processors [13]. These computation decompositions are passed to a code generator which schedules the parallel

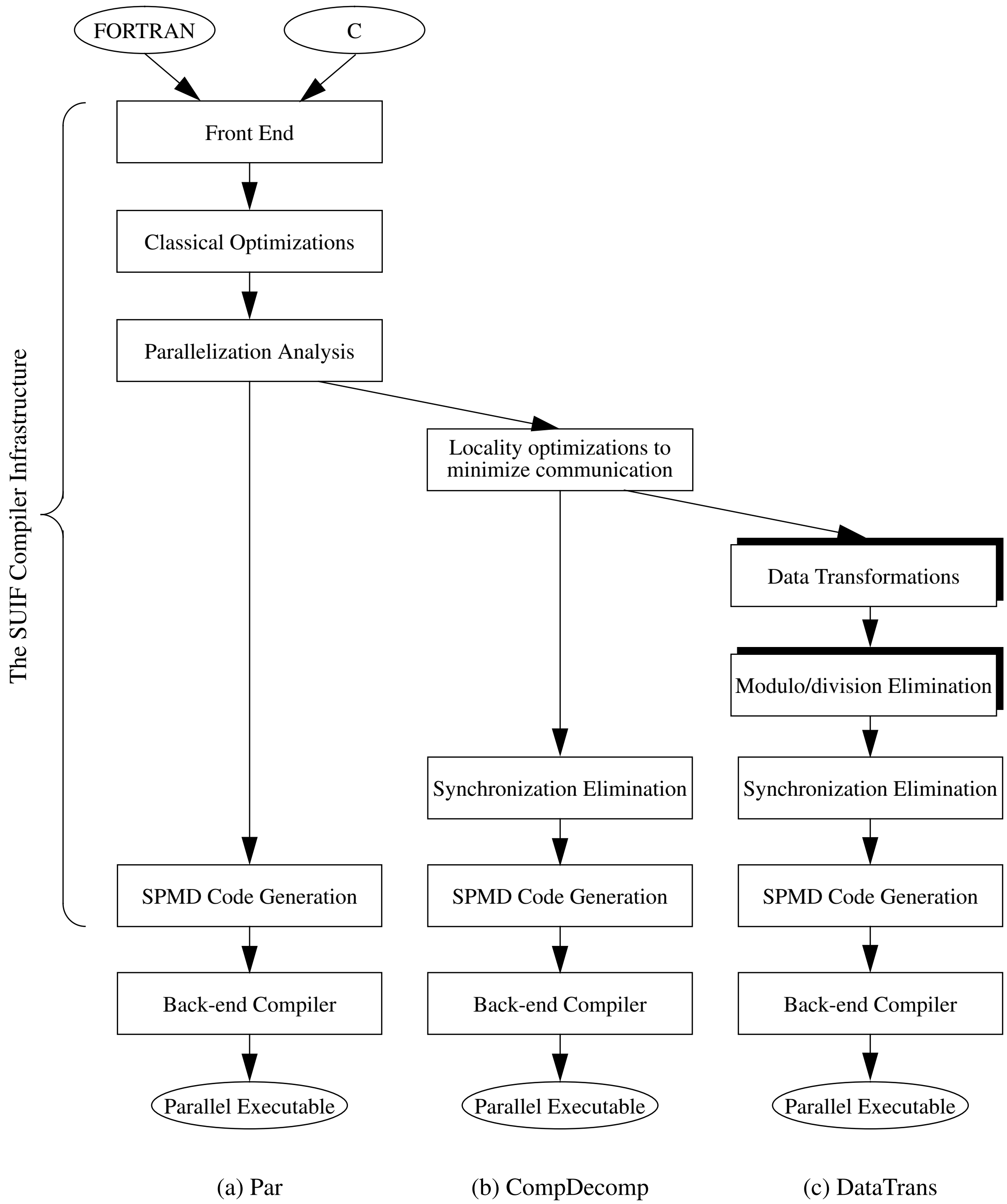


Figure 8-23. Compiler optimizations performed for the experiments

loops and inserts calls to the run-time library. The code generator also takes advantage of this information to minimize synchronization in the parallel program [140]. The data layouts are left unchanged.

c) **DataTrans**: Finally, we include the data transformations described in this chapter. Using the data decompositions calculated by the communication minimization algorithm [13], the compiler reorganizes the arrays in the parallelized code to improve spatial locality. After transforming the array accesses, the access functions are simplified using modulo and division optimizations described in Section 8.3.

8.4.2. Results

We present performance results for six selected kernels and small programs that require data transformations. We briefly describe the programs and discuss the opportunities for optimization. All of these programs have high parallel coverage but poor parallel performance. We show that, using data transformation optimizations, we can obtain a significant improvement in parallel performance in many of these programs.

8.4.2.1. Vpenta

Vpenta is one of the kernels in nasa7, a program in the SPEC92 floating-point benchmark suite. This kernel simultaneously inverts three pentadiagonal matrices. The performance results are shown in Figure 8-24. The base compiler interchanges the loops in the original code so that the outer loop is parallelizable and the inner loop carries spatial locality. Without such optimizations, the program would not even achieve the slight speedup obtained with the base compiler.

For this particular program, the base compiler's parallelization scheme is the same as the results from the computation decomposition algorithm. However, since the compiler can determine that each processor accesses exactly the same partition of the arrays across the loops, the synchronization optimization algorithm can eliminate barriers between some of the loops. This accounts for the slight increase in performance of the computation decomposition version over the base compiler.

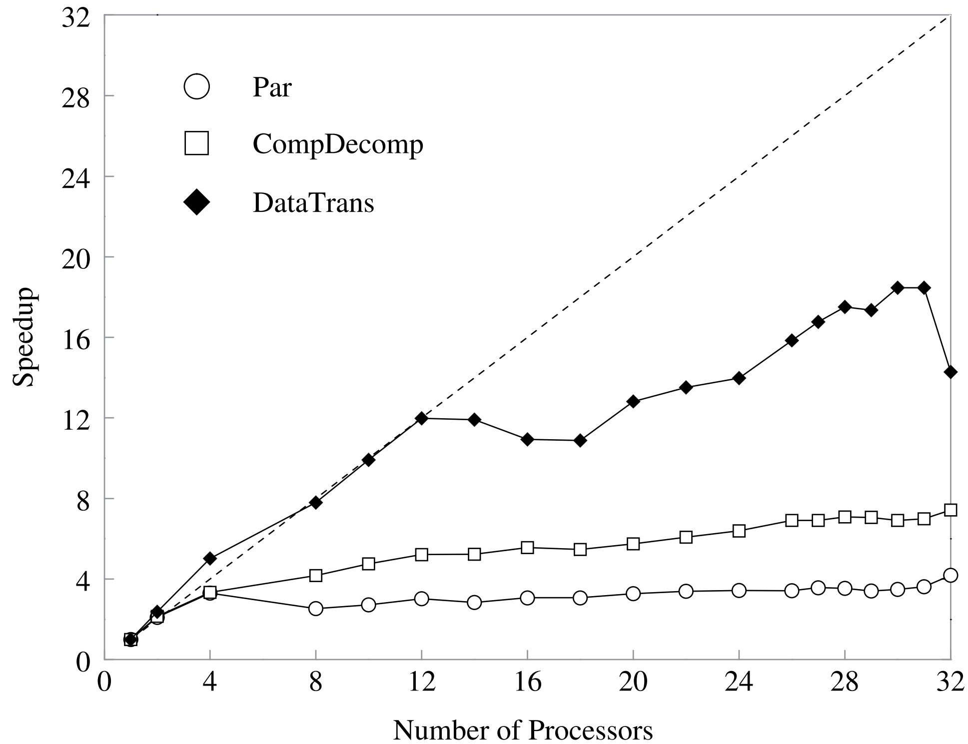


Figure 8-24. Performance of Vpenta

This program operates on a set of two-dimensional and three-dimensional arrays. Each processor accesses a block of columns for the two-dimensional arrays; thus no data reorganization is necessary for these arrays. However, each plane of the three-dimensional array is partitioned into blocks of rows, each of which is accessed by a different processor. This presents an opportunity for our compiler to change the data layout and make the data accessed contiguous on each processor. With the improved data layout, the program finally runs with a decent speedup. We observe that the performance dips slightly when there are about 16 processors, and drops significantly when there are 32 processors. This performance degradation stems from increased cache conflicts between accesses within the same processor. Further data and computation optimizations that focus on operations on the same processor would be useful.

8.4.2.2. LU Decomposition

Our next program is LU decomposition without pivoting. The code is shown in Figure 8-26 and the speedups for each version of LU decomposition are displayed in Figure 8-25 for two different data set sizes (256×256 and 1024×1024).

```
      DOUBLE PRECISION A(N,N)
      DO 10 I1 = 1,N
           DO 10 I2 = I1+1, N
                A(I2,I1) = A(I2,I1) / A(I1,I1)
                DO 10 I3 =I1+1, N
                     A(I2,I3) = A(I2,I3) -A(I2,I1)*A(I1,I3)
 10   CONTINUE
```

Figure 8-26. LU Decomposition code

The parallelization algorithm identifies the second loop as the outermost parallelizable loop nest, and distributes its iterations uniformly across processors in a block fashion. As the number of iterations in this parallel loop varies with the index of the outer sequential loop, each processor accesses different data each time through the outer loop. A barrier placed after the distributed loop is used to synchronize between iterations of the outer sequential loop. The computation decomposition algorithm minimizes true-sharing by assigning all operations on the same column of data to the same processor. To minimize the load imbalance, the columns and operations on the columns are distributed across the processor in a cyclic manner. By fixing the assignment of computation to processors, the compiler replaces the barriers at the end of each execution of the parallel loop with locks. Even though this version has good load balance, good data re-use and inexpensive synchronization, the local data accessed by each processor is scattered in the shared address space,

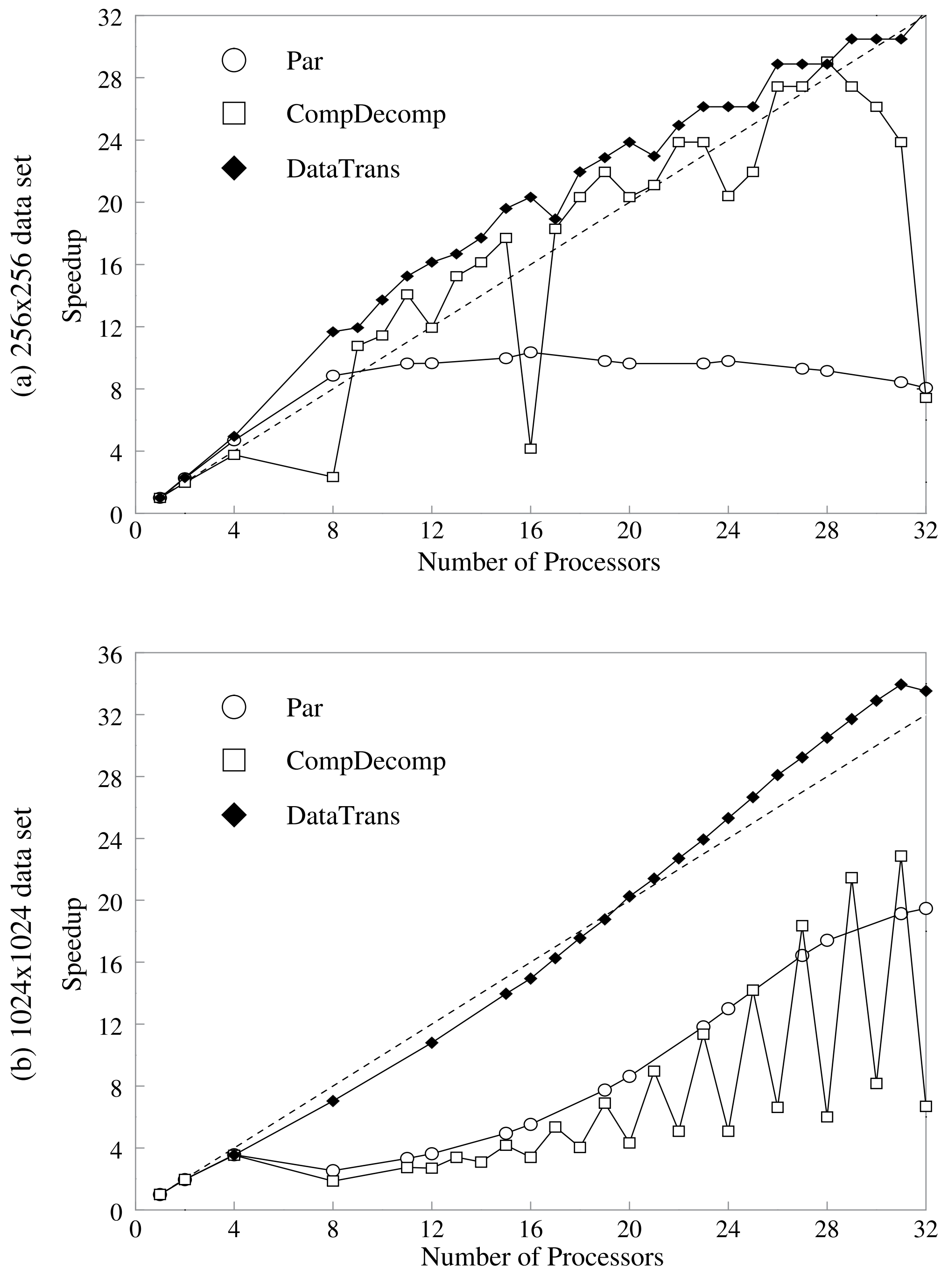


Figure 8-25. Performance of LU decomposition

increasing the chances of interference in the cache between columns of the array. The interference is highly sensitive to the array size and the number of processors; the effect of the latter can be seen in Figure 8-25. This interference effect can be especially pronounced if the array size and the number of processors are both powers of 2. For example, for the 1024×1024 matrix, every 8th column maps to the same location in DASH's direct-mapped 64K cache. The speedup for 31 processors is 5 times better than that for 32 processors.

The data transformation algorithm restructures the columns of the array so that each processor's cyclic columns are put into a contiguous region of memory. After restructuring, the performance stabilizes and is consistently high. In this case the compiler is able to take advantage of inexpensive synchronization and data re-use. Speedups become super-linear in some cases due to the fact that once the data is partitioned among enough processors, each processor's working set fits into local memory.

8.4.2.3. Five-Point Stencil

The code for our next example, a five-point stencil, is shown in Figure 8-27. Figure 8-28 shows the resulting speedups for each version of the code. The parallelization pass simply distributes the outermost parallel loop across the processors, and each processor updates a block of array columns. The values of the boundary elements are exchanged in each time step. The computation decomposition algorithm assigns two-dimensional blocks to each processor, since this mapping has a better computation-to-communication ratio than a one-dimensional mapping. However, without also changing the data layout, the performance is worse than the base version because now each processor's partition is non-contiguous (In Figure 8-28, the number of processors in each of the two dimensions is also shown under the total number of processors).

After the data transformation is applied, the program has good spatial locality as well as less communication, and thus we achieve a speedup of 29 on 32 processors. Note that the performance is very sensitive to the number of processors. This is due to the fact that each DASH cluster has 4 processors and the amount of communication across clusters differs significantly for different two-dimensional mappings.

```
      REAL A(N,N), B(N,N)
C Initialize B
        ...
C Calculate Stencil
      DO 30 time = 1,NSTEPS
            ...
            DO 10 I1 = 1, N
                  DO 10 I2 = 2, N
                       A(I2,I1) = 0.20*(B(I2,I1)+B(I2-1,I1)+
     x                 B(I2+1,I1)+B(I2,I1-1)+B(I2,I1+1))
10          CONTINUE
            ...
30    CONTINUE
```

Figure 8-27. Five-point stencil code

8.4.2.4. Erlebacher

Erlebacher is a 600-line FORTRAN benchmark from ICASE that computes three-dimensional tridiagonal solutions. It includes a number of fully parallel computations, interleaved with multi-dimensional reductions and computational wavefronts in all three dimensions caused by forward and backward substitutions. Partial derivatives are computed in all three dimensions with three-dimensional arrays. Figure 8-29 shows the resulting speedups for each version of Erlebacher.

The parallelization analysis always parallelizes the outermost parallel loop. This strategy yields local accesses in the first two phases of Erlebacher when computing partial derivatives in the X and Y dimensions, but ends up causing non-local accesses in the Z dimen-

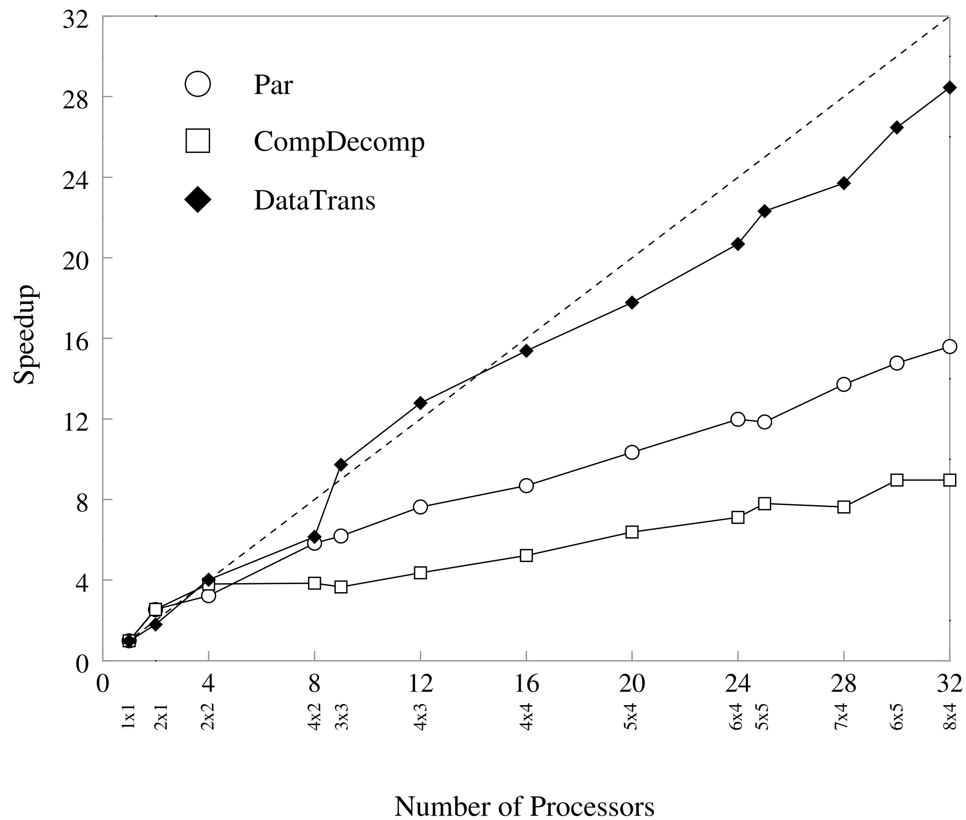


Figure 8-28. Performance of 5-point stencil

sion. The computation decomposition algorithm improves the performance of Erlebacher slightly over the base-line version. It finds a computation decomposition so that no non-local accesses are needed in the Z dimension. The major data structures in the program are the input array and DUX, DUY and DUZ which are used to store the partial derivatives in the X, Y and Z dimensions, respectively. Since it is only written once, the input array is replicated. Each processor accesses a block of columns for arrays DUX and DUY, and a block of rows for array DUZ. Thus in this version of the program, DUZ has poor spatial locality.

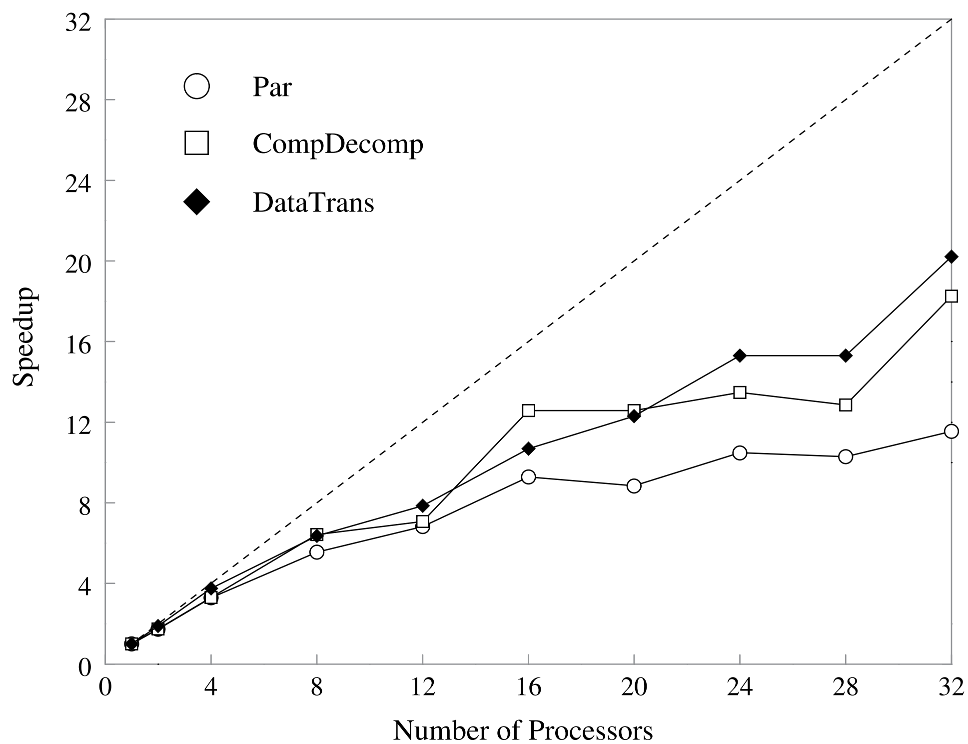


Figure 8-29. Performance of Erlebacher

Our data transformation algorithm restructures DUZ so that local references are contiguous in memory. Because two-thirds of the program are perfectly parallel with all local accesses, the optimizations only realize a modest performance improvement.

8.4.2.5. Swm256

Swm256 is a 500-line program from the SPEC92 benchmark suite. It performs a two-dimensional stencil computation that applies finite-difference methods to solve shallow-water equations. The speedups for swm256 are shown in Figure 8-30.

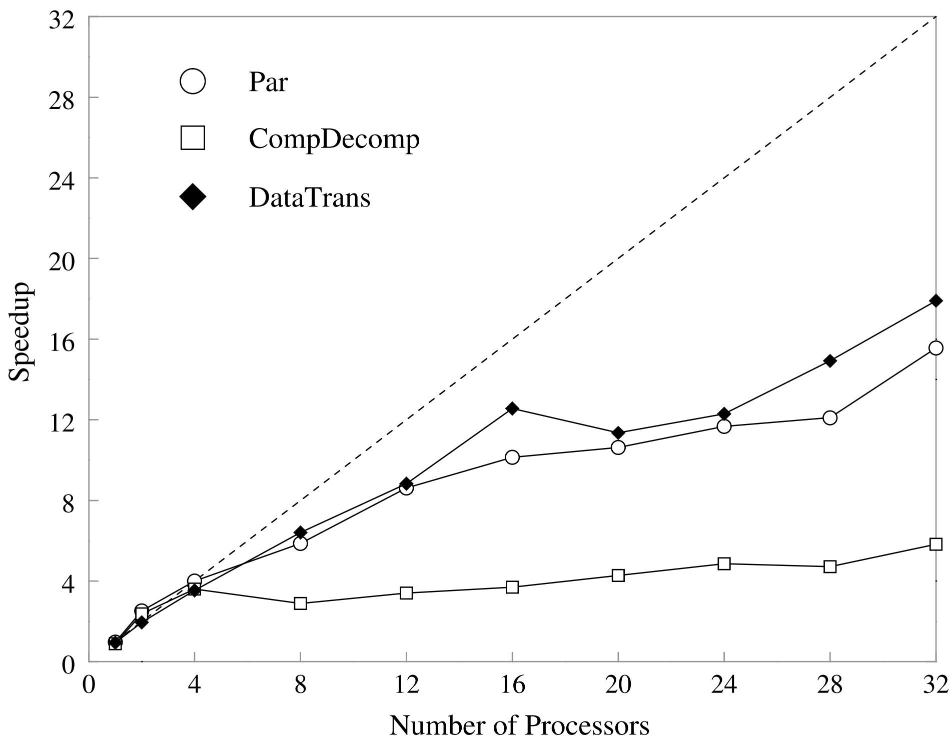


Figure 8-30. Performance of swm256

Our base compiler is able to achieve good speedups by parallelizing the outermost parallel loop in all the frequently executed loop nests. The decomposition phase discovers that it can, in fact, parallelize both of the loops in the 2-deep loop nests in the program, without incurring any major data reorganization. The compiler chooses to exploit parallelism in both dimensions simultaneously in an attempt to minimize the communication to computation ratio. Thus, the computation decomposition algorithm assigns two-dimensional blocks to each processor.

However, the data accessed by each processor is scattered, causing poor cache performance. Fortunately, when we apply both the computation and data decomposition algorithm to the program, the program regains the performance lost and is slightly better than that obtained with the base compiler.

8.4.2.6. Tomcatv

Tomcatv is a 200-line mesh generation program from the SPEC92 floating-point benchmark suite. Figure 8-31 shows the resulting speedups for each version of tomcatv. Tomcatv contains several loop nests that have dependences across the rows of the arrays, and other loop nests that have no dependences. Since the parallelization algorithm always parallelizes the outermost parallel loop, each processor accesses a block of array columns in the loop nests with no dependences. However, in the loop nests with row dependences, each processor accesses a block of array rows. As a result, there is little opportunity for data re-use across loop nests. Additionally, there is poor cache performance in the row-dependent loop nests because the data accessed by each processor is not contiguous in the shared address space. The computation decomposition pass selects a computation decom-

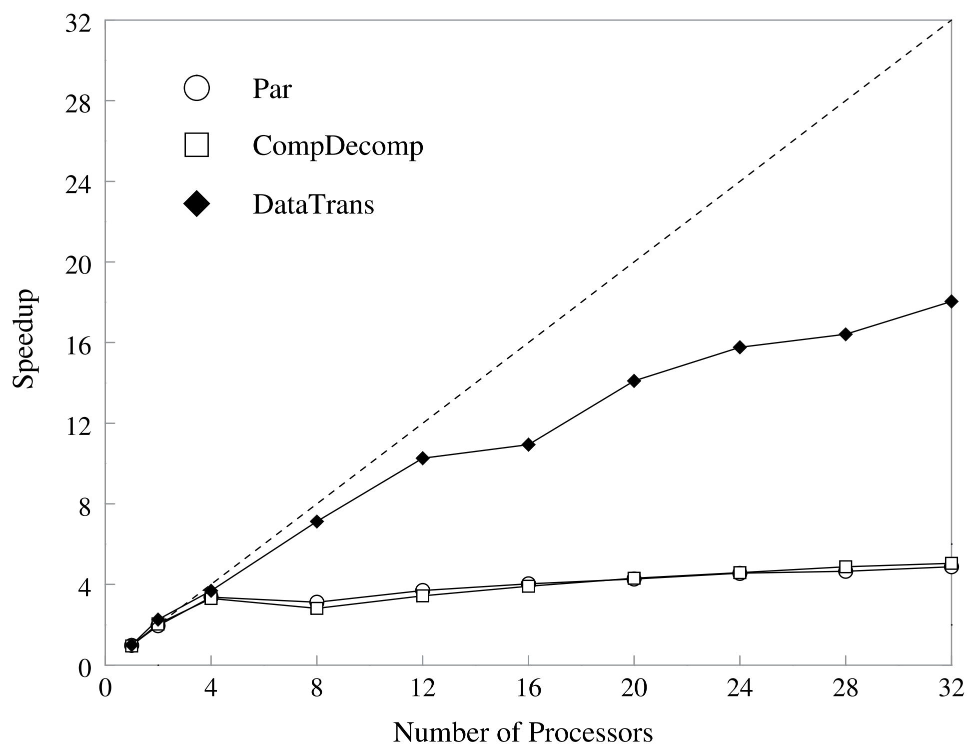


Figure 8-31. Performance of tomcatv

position so that each processor always accesses a block of rows. The row-dependent loop nests still execute completely in parallel.

This version of tomcatv exhibits good temporal locality; however, the speedups are still poor due to poor cache behavior. After transforming the data to make each processor's rows contiguous, the cache performance improves. Whereas the maximum speedup achieved by the base version is 5, the fully optimized tomcatv achieves a speedup of 18.

8.5. Related Work

Previous work on compiler algorithms for optimizing memory hierarchy performance has focused primarily on loop transformations. Unimodular loop transformations, loop fusion, and loop nest blocking restructure computation to increase uniprocessor cache re-use [33,61,147]. Copying data into contiguous regions has been studied as a means for reducing cache interference [103,134].

Several researchers have proposed algorithms to transform computation and data layouts to improve memory system performance [39,89]. The same optimizations are intended to change the data access patterns to improve locality on both uniprocessors and shared address space multiprocessors. These algorithms use only array permutation transformations, they do not consider strip-mining. By using strip-mining in combination with permutation, our compiler is able to optimize spatial locality by making the data used by each processor contiguous in the shared address space. This means, for example, that our compiler can achieve good cache performance by creating cyclic and multi-dimensional blocked distributions.

Compile-time data transformations have also been used to eliminate false-sharing in explicitly parallel C code [88]. The domain of that work is quite different from ours; we consider both data and computation transformations, and the code is parallelized automatically. Their compiler statically analyzes a parallel program to determine the data accessed by each processor, and then tries to group the data together. Two different transformations are used to aggregate the data. Their compiler turns groups of vectors that are accessed by different processors into an array structure. Each structure contains the aggregated data accessed by a single processor. References to the original data structures are replaced with pointers to the newly allocated per-processor data structures. Their compiler also aligns data structures that have no locality (e.g. locks) with cache line boundaries to avoid false-sharing.

Optimizing address calculations with modulo and division operations has been studied in the context of block-cyclic decompositions in HPF compilers for distributed address space machines. The complex access functions are replaced by a finite state machine created by the compiler, that generates the correct access pattern when executed at runtime [37,95]. In contrast, our modulo and division optimizations are capable of completely eliminating many of the modulo and division operations generated in practice. Thus, we do not have to pay for the overhead of executing a finite state machine at runtime for most of the access functions created with modulo and division operations. However, address generation using a finite state machine can be included as a fall-back technique when the access functions are too complex and cannot be optimized.

8.6. Chapter Summary

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

9 Communication Generation and Optimization for Distributed Address-Space Machines

Locating parallelism is sufficient to generate parallel programs for cache-coherent shared address-space machines. However, as we discussed in the previous chapter, much more analysis and optimization is required to obtain good parallel performance. For example, the compiler needs to explicitly decompose the computation and data across the processors to exploit the data locality and minimize the communication overhead.

Finding computation and data decompositions is necessary when generating code for distributed address space machines. Furthermore, the compiler is faced with the additional problem of managing the memory and the communication explicitly. The parallel programs created by the compiler must issue explicit communication instructions. It is also necessary to perform many communication optimizations in order to obtain good parallel performance.

Given a computation and data decomposition, the techniques described in this chapter automatically produce an SPMD program with the necessary receive and send instructions, optimize the communication by eliminating redundant communication and aggregating small messages into large messages, allocate space locally on each processor, and translate global data addresses to local addresses. The communication code generation and communication optimization techniques described in this chapter are again based on our linear inequalities framework.

Since the combined problem of locating coarse-grain parallelism and determining computation and data decompositions that minimize communication is very complex, many compiler systems for distributed address-space machines rely on users to supply the data decompositions. Languages such as High Performance FORTRAN [83], FORTRAN-D[85] and Vienna FORTRAN [36] allow the programmer to annotate the sequential program with data decompositions. Our algorithms can use this decomposition information in lieu of compiler-generated data and computation decomposition information.

The organization of this chapter is as follows. In Section 9.1., we describe two different approaches for generating communication: the traditional location-centric approach with user-specified data decompositions, and a novel value-centric approach based on exact data-flow information and computation decompositions. We formally describe the domain of our technique in Section 9.2. Section 9.3. presents a mathematical representation for communication. We describe our code generation technique and our communication optimizations in Sections 9.4 and 9.5.

9.1. Determining Communication

9.1.1. Location-Centric Approach

Many of the existing compilers developed for distributed memory machines have a similar basic approach to how they generate code from user-specified data decompositions [14,98,85,114,123,139].

For simplicity, in the following discussion we assume that there is only one loop nest that contains one read access and one write access to the same array. The argument obviously holds for the general case with multiple loops and arrays. Three domains are manipulated in the compilation process: the iteration space $\Im$, the array elements space $\mathcal{A}$, and the processor space $\mathcal{P}$. Each array access function in the source program specifies the data used by each iteration in the loop. That is, each read or write access function, denoted by $f_r, f_w : \Im \rightarrow \mathcal{A}$ respectively, maps an iteration to the array indices of the data read or written. The user-specified data decomposition $D: \mathcal{A} \rightarrow \mathcal{P}$ maps each array location to a processor. From the read and write access functions and the data decomposition, the compiler automatically derives the computation decomposition $C: \Im \rightarrow \mathcal{P}$ which maps each iteration in the loop to a processor.

To derive the computation decomposition, the compiler applies the *owner-computes rule*: each assignment statement is performed by the processor that owns the data. Therefore, given a write access function f_w and a data decomposition D, the computation decomposition is $C = Df_w$. Under the owner-computes rule, no communication is needed to implement the write accesses. Communication is needed for a read access in iteration i if the data read is resident on a different processor, i.e. $Ci \neq Df_r i$. The relationships between iteration space, array space, and the processor space are shown in Figure 9-1(a); processor p_r receives data from processor p_s if $p_r \neq p_s$.

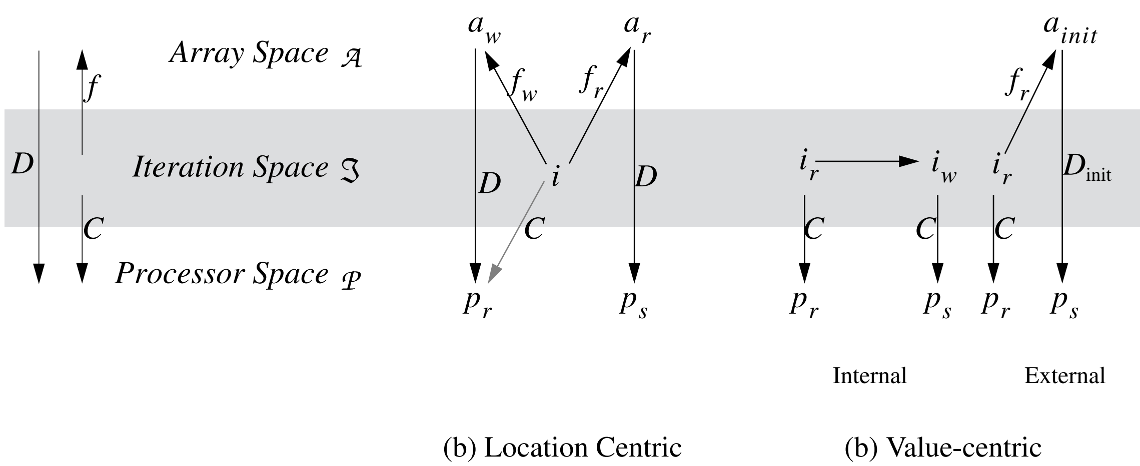


Figure 9-1. Different approaches to code generation for distributed memory machines

To minimize the communication cost, the compiler tries to maximize the intervals between communication. All the data needed within the interval are sent in one message. This optimization is based on data dependence analysis given in Definition 3-1. The *maximum level* of a dependence between two references is simply the maximum loop nest level that carries a dependence between the references. If the maximum level of all dependences involving a read access is k, the compiler needs to communicate only once in each iteration of the k-th loop. Thus, the maximum level information is useful for reducing the communication frequency. All the data accessed within the interval requiring communication are summarized by a regular section description [81]. In this way, the same data used multiple times within the interval are transferred only once.

In summary, this approach deduces the computation decompositions from the user-specified data decompositions, using the owner-computes rule; it uses data dependence analysis to reduce the number of messages; finally, it uses the concept of regular sections to reduce redundant data transfers.

9.1.2. Value-Centric Approach

Instead of location-based data dependence analysis, communication identification can be based on less restrictive value-based data-flow information. Let us use the simple example in Figure 9-2 to illustrate the difference.

```
DO T = 1, 100
      DO I = 3, N
            X[I] = X[I-3]
```

Figure 9-2. Simple 2-deep loop nest

Data dependence analysis on this program will produce the dependence vectors {[+, 3], [0, 3]}, meaning that the read access in iteration $[t_r, i_r]$ may be data dependent on all iterations $[t_w, i_w]$ such that $i_w = i_r - 3$, and $t_w \le t_r$. Exact data-flow analysis, however, is able to determine precisely that the first three iterations of the innermost loop read data defined outside the loop, and the rest of the iterations use the value defined three iterations earlier, i.e., $[t_w, i_w] = [t_r, i_r - 3]$. Also, the first three iterations read data, X[0:2], whose values are not generated by this program. The exact data-flow information is given in Figure 9-3.

$$[t_r, i_r] = \begin{cases} \text{write in iteration } [t_r, i_r - 3] & i_r \geq 6 \\ \text{array locaiton } (i_r - 3) \text{ at start} & i_r < 6 \end{cases}$$

Figure 9-3. The exact data-flow information

The problem of finding precise array data-flow information was first formulated by Feautrier [53,54,55], and is described in Section 4.6.

The exact data-flow information maps an instance of a read operation to the very write instance that produces the value read, provided such a write exists. This mapping is denoted by $\mu : i_r \rightarrow i_w$, where i_r and i_w are the loop indices of the read and write instances respectively. If the instances within a context do not read any value written within the loop, we denote these instances by the mapping $\tau : i_r \rightarrow a$, where i_r is the loop index of the read instance and a is the array location of the data at the beginning of the loop.

This information differs from that produced by data dependence analysis in two major ways. First, exact data-flow information can distinguish between different instances of the same array access. For example, exact data-flow analysis can determine that the first three iterations of the inner loop in Figure 9-2 have dependence relationships different from all the other iterations. Second, the exact data-flow analysis specifies precisely the last write instance that generates the value read by a particular read instance. Data dependence analysis, on the other hand, cannot discriminate between writes to the same location.

The exact data-flow information can be used for communication identification in the following manner. For read instances with the mapping τ, that do not read any of the values written within the code being analyzed, the compiler can simply load all the non-local data onto a processor before executing any of the code. Given a computation decomposition and an initial data decomposition produced by an earlier compiler phase, the technique to generate the necessary communication code is no different from that used in the location-centric approach. Communication and computation are more tightly-coupled for read instances with a corresponding write instance. In this case, the mapping μ specifies all the pairs of iterations that share a producer and consumer relationship. By applying the computation decomposition function on the related iterations, we can derive the identity of the processors that write and read the same value. If the writer and reader are different processors, then communication is necessary. This technique is depicted in Figure 9-1(b).

The data decompositions generated from the earlier compiler phase serve only as interfaces with other sections of the program. In general, we can generate the necessary communication from the exact data-flow information and computation decompositions, and not data decompositions. We can also change the data layout when called for by the computation decompositions. Thus, this approach can support a wider range of data decompositions. Locations written to can be replicated or mapped to different processors over time. Furthermore, this approach does not rely on the restriction of the owner-computes rule.

Using the data-flow information, we can easily eliminate redundant data transfers. While accessing the same *location* may require multiple data transfers since the value at the location may or may not have changed, each *value* needs to be transferred once and only once. Moreover, the perfect producer and consumer information enables the compiler to issue the send immediately after the data is produced, and to issue the receive just before the data is used. This maximizes the chances that the communication is overlapped with computation.

9.2. Problem Domain

In this section, we formally define the scope of our technique. We show how we can represent all the information useful for communication and computation code generation as sets of linear inequalities. This model and our techniques discussed in the next section are useful for both value-centric and location-centric approaches.

The scope of our technique is limited to programs consisting of a set of loop nests, where the bounds of the loop nests are affine expressions of outer loop indices and symbolic constants. The array accesses are also affine functions of loop indices and symbolic constants.

Our technique can also handle conditional statements that contain no loops. Each assignment within the conditional statement is treated as an unconditional assignment; depending on the outcome of the condition, it assigns to the variable either the newly computed value or the variable's current value.

We can handle loops given by Definition 2-2, array index sets given by Definition 2-5 and virtual and physical processors given by Definition 2-6. The read and write access functions, $\bar{f_r} = \left(f_{r_1}, \ldots, f_{r_m}\right)$ and $\bar{f_w} = \left(f_{w_1}, \ldots, f_{w_m}\right)$ are affine functions such that $\bar{f}\ (\bar{v}, i_1, \ldots, i_n) = (a_1, \ldots, a_m)$, where $(i_1, \ldots, i_n) \in I$, $(a_1, \ldots, a_m) \in A$ and $\bar{v}$ is a symbolic constant vector.

To support cyclic decompositions where data or computation are distributed to processors in a round-robin manner, we introduce the notion of a virtual processor array. The computation and data decompositions map the computation and data to the virtual processor space. Let $u_1, \ldots, u_q$ be the dimensions of the virtual processor space, the index set of this virtual processor array is thus

$$P = \{\bar{p} = (p_1, \ldots, p_q) \in \mathcal{P} \ \middle|\ \forall k = 1, \ldots, q \ \ 0 \le p_k < u_k\}$$

Our physical processor array has the same number of dimensions as the virtual processor array. Let $u'_1, \ldots, u'_q$ be the physical processor array dimensions, $u'_k \le u_k, \forall k = 1, \ldots, q$. The physical processor index set is

$$P' = \{\overline{p'} = (p'_1, \ldots, p'_q) \in \mathcal{P} \ \middle|\ \forall k = 1, \ldots, q \ \ 0 \le p_k < u'_k\}$$

The k-th dimension of data elements or loop iterations are distributed across the physical processors in a cyclic manner whenever $u'_k < u_k$. The mapping from the virtual to the physical processor space, $\pi: \mathcal{P} \rightarrow \mathcal{P}$ is defined as $\pi(\bar{p}) = \overline{p'}$ where $\underset{k = 1, \ldots, q}{\forall} \ p'_k = p_k \bmod u'_k$. Since only in the latter stages of the optimizations will the compiler be operating in the physical processor space, we will simply refer to virtual processors as processors.

9.2.1. Data Decompositions

Definition 9-1: *The* ***data decomposition relation*** D *is a set of array element and processor pairs* $(\bar{a}, \bar{p})$ *, such that* $(\bar{a}, \bar{p}) \in D$ *iff the processor* $\bar{p}$ *has a copy of the array element* $\bar{a}$*. Data decompositions can be written as*

$$D = \left\{ (\bar{a}, \bar{p}) \in A \times P \;\middle|\; \begin{array}{c} U(\bar{a} - \bar{t}) \geq (\bar{b} + B\bar{u})\,\bar{p} - \bar{d}_l \\ U(\bar{a} - \bar{t}) < (\bar{b} + B\bar{u})\,(\bar{p} + 1) + \bar{d}_h \end{array} \right\}$$

where U *is an extended unimodular matrix,* $\bar{t}$ *,* $\bar{d}_l, \bar{d}_h$*,* $\bar{b}$*, are integer vectors, B is an integer matrix and* $\bar{u}$ *is a vector of symbolic constants such that* $\bar{b} + B\bar{u} \geq 0$ *and* $\bar{d}_l, \bar{d}_h \geq 0$*.*

The scope of data decompositions defined in this chapter is larger than the decompositions used Chapter 8, which are typically used in existing distributed memory machine compilers. The matrix U determines if the array is reversed or skewed. When the array has more dimensions than that of the processor space, the 0 columns of the *extended* unimodular matrix U chooses the dimensions to be mapped onto the same processor. The entire array can be shifted with respect to the processor array using the integer vector $\bar{t}$. Since the data block size is often a function of the number of processors engaged in the computation, it is useful not to determine the block size at compile time. We can handle some symbolic block sizes of the form $\bar{b} + B\bar{u}$; the scope of our technique is discussed in Section 2.3. The overlap of array elements between processors is determined by vectors $\bar{d}_l, \bar{d}_h \geq 0$. Figure 9-4 illustrates how we can use this scheme to describe several common data decompositions. The 2×2 grid in each example represents the first 2×2 processors in the system; each panel is a picture of the entire data array, and the shaded portion represents the data allocated locally to that specific processor.

9.2.2. Computation Decompositions

Computation decompositions have a scope similar to that of data decompositions, except that an iteration can be mapped onto only one processor.

Definition 9-2: *The* ***computation decomposition relation*** C *is a set of iteration and processor pairs* $(\bar{i}, \bar{p})$ *such that processor* $\bar{p}$ *executes iteration* $\bar{i}$ *iff* $(\bar{i}, \bar{p}) \in C$. *computation decompositions can be written as*

$$C = \left\{ (\bar{i}, \bar{p}) \in I \times P \;\middle|\; (\bar{b} + B\bar{u})\bar{p} \le U(\bar{i} - \bar{t}) < (\bar{b} + B\bar{u})(\bar{p} + \bar{1}) \right\}$$

where U *is an extended unimodular matrix,* $\bar{t}$, $\bar{b}$ *are integer vectors, B is an integer matrix and* $\bar{u}$ *is a vector of symbolic constants such that* $\bar{b} + B\bar{u} > 0$.

The computation decompositions can be either generated automatically by an earlier compiler phase [13] or manually by the user. Theorem 9-1 shows how to derive computation decompositions from user-specified data decompositions.

Theorem 9-1: *Assuming that written data are not replicated, the computation decomposition as derived from data decomposition* D, *using the owner-computes rule, is*

$$C = \left\{ (\bar{i}, \bar{p}) \in I \times P \;\middle|\; \exists a \in A \text{ s.t. } (\bar{a}, \bar{p}) \in D \wedge a = \dot{f}_w(\bar{v}, \bar{i}) \right\}$$

9.3. Communication

In this section we define the communication between processors for both location-centric and value-centric approaches.

Definition 9-3: *A* ***communication set*** M *is a set of elements* $(\bar{i}_r, \bar{p}_r, \bar{i}_s, \bar{p}_s, \bar{a}) \in I \times P \times I \times P \times A$, *where* $(\bar{i}_r, \bar{p}_r, \bar{i}_s, \bar{p}_s, \bar{a}) \in M$ *iff processor* $\bar{p}_s$ *needs to send the value in location* $\bar{a}$ *in iteration* $\bar{i}_s$ *to processor* $\bar{p}_r$ *for use in iteration* $\bar{i}_r$.

9.3.1. Using Data Decompositions and the Owner-Computes Rule

If we use the owner-computes rule, no communication is necessary for write operations. We use Theorem 9-2 to find all the necessary communication for each read access within the loop nest. We use the user-specified data decomposition to find the owner of the data

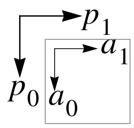
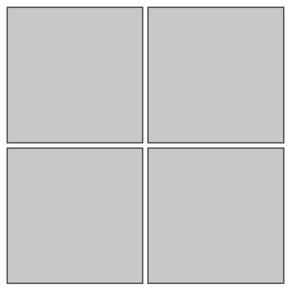
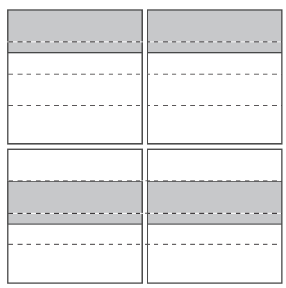
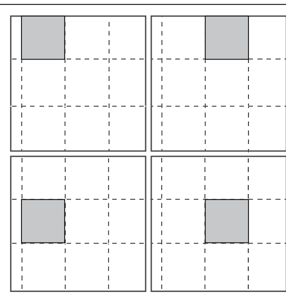
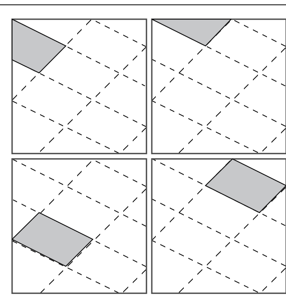
 	**U**		$\overline{\mathbf{b}}+\mathbf{B}\overline{\mathbf{u}}$	**t**	$\overline{\mathbf{d}}_\mathbf{l}$	$\overline{\mathbf{d}}_\mathbf{h}$
	1 0	0 1	0 0	0 0	0 0	N N
(a) Full replication.						
	1 0	0 1	15 0	0 0	0 0	1 N
(a) Blocked rows with overlap.						
	1 0	0 1	x x	0 1	0 0	0 0
(a) Square blocks with symbolic block sizes, shifted right by 1.						
	1 2	-1 1	8 16	1 -1	0 0	0 0
(a) Skewed rectangular blocks.						

Figure 9-4. Examples of some data decompositions for an NxN array onto a 2-dimensional processor space

read. We use the computation decomposition derived from Theorem 9-1 to find the processor reading the data. If these two processors are not the same, communication is needed.

Theorem 9-2: *The communication set required by the access function* $\bar{f}$ *for a set of iterations* ι *under computation decomposition* C *and data decomposition* D *is the set of elements* $(\bar{i}_r, \bar{p}_r, \bar{i}_s, \bar{p}_s, \bar{a}) \in I \times P \times I \times P \times A$, *where* $(\bar{i}_r, \bar{p}_r) \in C$, $(\bar{a}, \bar{p}_s) \in D$, $\bar{i}_r \in \iota$, $\bar{a} = \bar{f}_r(\bar{v}, \bar{i}_r), \bar{i}_s = \bar{i}_r$ *and* $\bar{p}_s \neq \bar{p}_r$

9.3.2. Using Computation Decompositions and the Exact Data-Flow Information

Definition 9-4: *An exact data-flow analysis partitions the iteration set of a loop nest into* ***contexts*** *so that a single mapping function will apply to each context. If the values read by the iterations in a context* $\iota \subseteq I$ *are written within the loop, then the context has a* ***last-write relation*** μ. *The last-write relation* μ *of the context* ι *is a set of iteration pairs* $(\bar{i}_r, \bar{i}_s)$ *such that* $(\bar{i}_r, \bar{i}_s) \in \mu$ *iff* $\bar{i}_r \in \iota$ *and* $\bar{i}_s \in I$ *is the iteration that generates the value read in iteration* $\bar{i}_r$. *A context* ι *can be written as* $\{\bar{i} \in I \mid \bar{q}(\bar{v}, \bar{i}) \geq \bar{0}\}$ *and a read-write relation* μ *can be written as* $\{(\bar{i}_r, \bar{i}_s) \in I \times I \mid \bar{q}'(\bar{v}, \bar{i}_r, \bar{i}_s) \geq \bar{0}\}$, *where* $\bar{q}$ *and* $\bar{q}'$ *are vectors of affine expressions.*

It is sometimes necessary to introduce auxiliary variables so that the last-write relations can be represented as linear inequalities. Some of the read-write relations need to be expressed as $i \equiv \beta \pmod{\alpha}$ or $i \not\equiv \beta \pmod{\alpha}$, where α, β are integers. We can introduce an auxiliary variable u, and rewrite these relations as $i - \beta = \alpha u$ and $\alpha u < i - \beta < \alpha u + \alpha$, respectively.

If an iteration reads data written within the loop, then communication is needed only if the iterations sharing the read-write relation are executed by different processors (Theorem 9-3). If an iteration uses data written outside the loop, then we use the initial data decomposition to determine the owner of the data (Theorem 9-4). Theorem 9-4 is similar to Theorem 9-2, except that the sends can precede the computation of the loop since the values needed are not generated within the loop.

Theorem 9-3: *The communication set that satisfies the last-write relation* μ *under computation decomposition* C *is the set of elements* $\left(\bar{i}_r, \bar{p}_r, \bar{i}_s, \bar{p}_s, \bar{a}\right) \in I \times P \times I \times P \times A$ *where* $\left(\bar{i}_r, \bar{p}_r\right), \left(\bar{i}_s, \bar{p}_s\right) \in C$, $\left(\bar{i}_r, \bar{i}_s\right) \in \mu$, $\bar{a} = \bar{f}_r\left(\bar{v}, \bar{i}_r\right) = \bar{f}_s\left(\bar{v}, \bar{i}_s\right)$, $\bar{p}_s \neq \bar{p}_r$.

Theorem 9-4: *The communication set required by an access function* $\bar{f}$ *within a context* ι *of read iterations where the value used is generated outside the loop nest, under computation decomposition* C *and an initial data decomposition* D*, is the set of elements* $\left(\bar{i}_r, \bar{p}_r, \bar{i}_s, \bar{p}_s, \bar{a}\right) \in I \times P \times I \times P \times A$, *where* $\left(\bar{i}_r, \bar{p}_r\right) \in C$, $\left(\bar{a}, \bar{p}_s\right) \in D$, $\bar{i}_r \in \iota$, $\bar{a} = \bar{f}_r\left(\bar{v}, \bar{i}\right)$, $\bar{i}_s = 0$ *and* $\bar{p}_s \neq \bar{p}_r$.

Communication decompositions, data decompositions, iteration contexts, access functions and last-write relations can all be expressed as systems of linear inequalities. The $\bar{p}_s \neq \bar{p}_r$ constraint, however, cannot be expressed as a conjunction of inequalities. We break down the inequality into a set of disjunctive conditions. For example, for a one-dimensional processor array, the constraint $p_s \neq p_r$ is represented by $p_s > p_r \vee p_s < p_r$. We represent the necessary communication as a set of communication sets, with each one satisfying all the other inequalities and one of the disjunctive conditions.

Suppose the second loop in our program in Figure 9-2 is distributed as blocks of 32 iterations across a linear array of processors. That is, processor p executes iteration [*t, i*] iff

$$32p \le \begin{bmatrix} 0 & 1 \end{bmatrix} \begin{bmatrix} t \\ i \end{bmatrix} < 32\,(p+1)$$

We will use this computation decomposition throughout the rest of the chapter. Figure 9-5 shows the communication sets for first context where the data is produced by a write from Figure 9-3.

9.3.2.1. Finalization

Data produced within the loop nest may need to be written back to its home location in the “final” data layout. We need to identify which written values are live at exit, and this can be derived from the exact data-flow information. For example, this is shown to be a sub-

Context	$t_r \ge 0$ $\quad T - t_r \ge 0$ $i_r - 3 \ge 0$ $\quad N - i_r \ge 0$ $i_r - 6 \ge 0$ $t_s - t_r \ge 0$ $\quad t_r - t_s \ge 0$ $i_s - i_r + 3 \ge 0$ $\quad i_r - i_s - 3 \ge 0$	
Access function	$i_r - 3 - a \ge 0$ $\quad a - i_r + 3 \ge 0$	
Computation decomposition for read iterations	$i_r - 32p_r \ge 0$ $\quad 32p_r + 31 - i_r \ge 0$	
Computation decomposition for write iterations	$i_s - 32p_s \ge 0$ $\quad 32p_s + 31 - i_s \ge 0$	
Constraint $p_s \ne p_r$	$p_s > p_r$	$p_s < p_r$

Figure 9-5. Inequalities defining the communication sets for first context, with producer-consumer relationship, in Figure 9-3

problem in calculating last write trees [112]. The set of inequalities generated, in conjunction with the final data distribution, defines the communication set for finalization.

9.4. Code Generation for Distributed Address-Space Machines

We use the code generation algorithm defined in Section 2.2. for generating SPMD loop nests with communication operations. When block sizes are not known at compile time, linear inequalities with symbolic coefficients, described in Section 2.3., are used.

9.4.1. Generating Computation and Communication Code

To find the computation allotted to each processor, we scan the elements in a computation decomposition relation C lexicographically in $(p_1, \ldots, p_q, i_1, \ldots i_n)$, or simply $(\bar{p}, \bar{i})$, order. The $\bar{p}$ loops enumerate the processors. The inner $\bar{i}$ loops enumerate the iterations to be executed for each value of $\bar{p}$. The SPMD code to be executed by each processor is as follows. Each processor checks to see if its processor number is within the bounds of the $\bar{p}$ loops. If so, the code it executes is simply the $\bar{i}$ loops parameterized by its processor number. In the case where the computation decomposition is cyclic, each processor must iterate through the virtual processors it represents. Examples 9-6(a) and 9-6(b) show the computation code for our example from Figure 9-2. The rest of the figure shows the communication code for the communication sets in Figure 9-5. Note that no communication is necessary when $p_s > p_r$.

To generate the receive and send code for a communication set M, we scan M lexicographically in $\left(\overline{p_r}, \overline{i_r}, \overline{p_s}, \overline{i_s}, \bar{a}\right)$ and $\left(\overline{p_s}, \overline{i_s}, \overline{p_r}, \overline{i_r}, \bar{a}\right)$ order, respectively. In the receive loop nest, the $\overline{p_r}$ loops enumerate the processors involved in receiving data. The $\overline{i_r}$ loops specify the iterations when processor $\overline{p_r}$ needs to receive data. By definition, the $\overline{p_s}$, $\overline{i_s}$ and $\bar{a}$ loops are degenerate loops containing only one iteration. The data to be received is the value in location $\bar{a}$ on processor $\overline{p_s}$ in iteration $\overline{i_s}$. Conversely, the $\overline{p_s}$ loops in the send loop nest enumerate all the senders. The $\overline{i_s}$ loops specify the iterations when processor $\overline{p_s}$ needs to send some messages. The $\overline{p_r}$ loops identify the receivers of each message. The $\overline{i_r}$ loops specify the iterations when processor $\overline{p_r}$ needs the data. The $\bar{a}$ loop is a degenerate loop containing the address of the data to be sent. If auxiliary variables have been introduced to handle modulo constraints, the auxiliary variables are placed last in the lexicographic order for both loops.

9.4.2. Merging Loop Nests

To generate the complete program for a processor, we need to merge a processor's computation code with its receive and send codes for each communication set.

A naive technique is to make each processor iterate through the entire loop nest in the source program. In each iteration, a processor checks whether the iteration belongs to its computation domain, and whether it is to take part in each of the communication sets. Checking such conditions in the innermost loop would be exorbitantly expensive. Some of this inefficiency can be eliminated by standard data-flow optimizations such as algebraic

```
if p >= 0 and p <= N / 32 then
    DO t = 0, T
        DO i = MAX(32 p, 3), MIN(32 p + 31, N)
            X[i] = X[i - 3]
```

(a) Computation code: scanning C in (p, t, i) order.

```
DO p_v = p_p, N / 32 step P
    DO t = 0, T
        DO i = MAX(32 p_v, 3), MIN(32 p_v + 31, N)
            X[i] = X[i - 3]
```

(b) Computation code when virtual processors p_v are mapped to P physical processors (p_p).

```
if p_r >= 1 and p_r <= N / 32 then
    DO t_r = 0, T
        DO i_r = 32 p_r, MIN(32 p_r + 2, N)
            p_s = p_r - 1
            t_s = t_r
            i_s = i_r - 3
            a = i_r - 3
            receive X[a] from iteration (t_s, i_s) in processor (p_s)
```

(c) Receive code: scanning first context in $(p_r, t_r, i_r, p_s, t_s, i_s, a)$ order.

```
if p_s >= 0 and p_s <= N/ 32 - 1 then
    DO t_s = 0, T
        DO i_s = 32 p_s + 29, MIN(32 p_s + 31, N - 3)
            p_r = p_s + 1
            t_r = t_s
            i_r = i_s + 3
            a = i_s
            send X[a] to iteration (t_r, i_r) in processor (p_r)
```

(d) Send code: scanning first context in $(p_s, t_s, i_s, p_r, t_r, i_r, a)$ order.

Figure 9-6. Computation and communication code for the Example 9-2.

simplification, invariant code motion, strength reduction, and common subexpression elimination.

Since all the conditions tested are affine expressions, we can potentially eliminate all run-time checks by splitting loops. Suppose we need to merge the loops given in Figure 9-7(a). Instead of generating the code in Figure 9-7(b) with excessive guards, we generate multiple loop nests, shown in Figure 9-7(c)

If the relative magnitude between the bounds of the individual loops is not known at compile time, loop splitting can expand the program size by a significant amount. Our compiler uses loop splitting only on inner loops, and also when the relative magnitudes between the loop bounds are known. We have also developed a dynamic splitting scheme that we use on the outer loops. The compiler does not generate all the possible combinations statically. Instead, each processor determines its bounds for all the iteration sets, sorts the bounds, and interprets the sorted list to determine the loops it has to execute. Finally, for iteration sets that are a function only of outermost loop variables, we insert dynamic checks into the bodies of the outer loops.

9.4.3. Local Address Space

Typically, a processor on a parallel machine touches only a part of an array. Since data sets processed by these programs are often very large, it is essential that the compiler allocates, on each processor, only enough storage for the data used by the processor.

The following is a simple approach to the memory allocation problem. We allocate on each processor the smallest rectangular region that covers all the data read or written by the processor, and we copy all the received data from the communication buffer into its respective home location in the array before it is accessed. Given a computation decomposition C and an access function f, the set of locations touched by processor p is $\{\bar{a} \in A \ | \exists \bar{i} \left(\bar{i}, \bar{p}\right) \in C \wedge a = \bar{f}\left(\bar{v}, \bar{i}\right)\}$. By scanning the inequalities lexicographically in $\left(\bar{p}, a_k, \bar{i}\right)$ order, the bounds we obtain on a_k are the bounds for the k-th dimension of the bounding box covering access f. If there are multiple accesses to the same array,

```
DO i = 0, 200
     receive(...)

DO i = 100, 300
     send(...)
```

(a) Two loop nests to be merged.

```
DO i = 0, 300
     if 0 <= i and i <= 200 then
          receive (...)
     if 100 <= i and i <= 300 then
          send (...)
```

(b) Naive merge of the loop nests with excessive checking.

```
DO i = 0, 99
     receive(...)
DO i = 100, 200
     receive(...)
     send(...)
DO i = 201, 300
     send(...)
```

(c) Optimized merge

Figure 9-7. Merging multiple loop nests.

we simply find the bounding box of the rectangular boxes for all the accesses to that array. Note that this formulation allows local data spaces on different processors to overlap.

The above algorithm is inadequate if the rectilinear bounding box of the data accessed is larger than the available local memory on the processor, while the data actually used fit in the local memory. Also, a processor's local memory may not be large enough to fit all the data that a processor will eventually use in a computation. In that case, we need to manage the memory dynamically.

The exact data-flow information provides a more efficient way to manage the data that has been received from other processors. The compiler knows precisely which values are read by every instance of the read access. Instead of first copying all the received data to its home locations, a processor can simply read the values directly from the communication buffers. The compiler also has information that tells when the buffer is no longer needed, and can manage the buffer space effectively.

9.5. Communication Optimizations

Since the above algorithms generate a receive and a send message for every read access to remote data, the code is correct but inefficient. It is essential that we eliminate the redundant messages and amortize the message sending overhead by batching the communication.

9.5.1. Eliminating Redundant Communication

Ancourt has also studied the problem of eliminating redundant communication [11]. Given a set of iterations and accesses, Ancourt's algorithm can construct a set of loop nests that fetches all the data touched without any duplication. This algorithm is adequate for removing redundant traffic if no communication is required within the loop nest. In general, transfers of data with the same address are redundant only if the values transferred are identical.

We separate redundancy into two categories. We say that there is *self reuse* when multiple instances of a single read access use the same data and *group reuse* when instances of different read accesses use the same data. We discuss each of these in turn below.

9.5.1.1. Redundant communication due to self reuse

Read instances that have different data-flow relationships often are amenable to different communication optimizations. By partitioning the read instances into different contexts according to their data-flow patterns, the exact data-flow information makes it easier to detect and eliminate redundancy. Our algorithm applies Theorem 9-5 to the communication set of each context to detect redundancy caused by self reuse.

Theorem 9-5: *Given a communication set M, communications* $\left(\bar{i}_r, \bar{p}_r, \bar{i}_s, \bar{p}_s, \bar{a}\right)$, $(\bar{i}'_r, \bar{p}'_r, \bar{i}'_s, \bar{p}'_s, \bar{a}') \in M$ *are redundant due to self reuse if* $\bar{p}_r = \bar{p}'_r$, $\bar{p}_s = \bar{p}'_s$, $\bar{i}_s = \bar{i}'_s$ *and* $\bar{a} = \bar{a}'$.

All elements in a communication set with identical i_s, $\bar{p}_s$ and $\bar{a}$ refer to the same values; all elements with identical $\bar{i}_s$, $\bar{p}_s$, $\bar{p}_r$ and $\bar{a}$ are redundant messages. Thus, we wish to replace the set of redundant messages with $\left(min\left(\bar{i}_r\right), \bar{p}_r, \bar{i}_s, \bar{p}_s, \bar{a}\right)$. This can be achieved by projecting the set onto the $\left(\bar{p}_s, \bar{i}_s, \bar{p}_r, \bar{a}\right)$ space, and constraining the upper bound of $\bar{i}_r$ to be identical to its lower bound. There are two complications. First, if the lower bound of $\bar{i}_r$ is expressed as a conjunction of multiple inequalities involving outer loop indices, then the communication set containing the minimum $\bar{i}_r$'s is no longer convex. The algorithm needs to divide the communication set into multiple convex sets. The second complication arises from the fact that a projected image may contain points that do not correspond to a solution in the original system. In many cases, a simple test can determine that no such degeneracies are present [112].

9.5.1.2. Redundant communication due to group reuse

Detection of reuse between arbitrary accesses to the same matrix can be expensive. However, one prevalent form of reuse can be incorporated and exploited easily within our model: the set of *uniformly generated references* [60]. Array index functions of uniformly generated references are affine functions of loop indices and symbolic constants, and they differ only in the constant terms. For example, X[i] and X[i+3] are uniformly generated references; so are B[2i+3j+1, 3j+n+3] and B[2i+3j+10, 3j+n+2], but not C[i] and C[j]. Reuse between uniformly generated references has been exploited successfully in improving cache locality [147,146]. Uniformly generated references are quite common in real programs, so much so that specialized languages and compilers have been built to translate them into efficient code [29,77].

We can represent a set of uniformly generated references by their convex hull, and describe the data flow information with a single mapping from write to read iterations. For the example in Figure 9-8, the set of accesses X[i], X[i - 1], X[i - 2] and X[i - 3] can be represented by the access function $f(i)$, where $f(i) = i - u$ and $0 \le u \le 3$. The exact data-flow information for all the accesses are given in Figure 9-9.

```
DO T = 1, 100
      DO I = 3, N
            X[I] = f(X[I], X[I-1],X[I-2],X[I-3])
```

Figure 9-8. The Example 9-2 with multiple read accesses

$$[t_r, i_r] = \begin{cases} \text{array locaiton } (i_r - u) \text{ at start} & \begin{matrix} i_r - u < 3 \quad or \\ t_r = 1, u = 0 \end{matrix} \\ \text{write in iteration } [t_r - 1, i_r] & t_r > 1, u = 0 \\ \text{write in iteration } [t_r, i_r - u] & i_r - u \ge 3, u > 0 \end{cases}$$

Figure 9-9. The exact data-flow information for the example from Figure 9-8

Note that the convex hull may contain more data than that accessed within an iteration; however, since a processor is typically responsible for a contiguous block of iterations, this method is unlikely to cause any significant unnecessary traffic.

9.5.1.3. Other forms of redundancies

Redundancy may also arise from cyclic decompositions, where a physical processor emulates multiple virtual processors. Given a virtual to physical processor mapping $\pi : \mathcal{P} \to \mathcal{P}$, communication $\left(\bar{i}_r, \bar{p}_r, \bar{i}_s, \bar{p}_s, \bar{a}\right) \in M$ can be eliminated if $\pi\left(\bar{p}_r\right) = \pi\left(\bar{p}_s\right)$. Also, communications $\left(\bar{i}_r, \bar{p}_r, \bar{i}_s, \bar{p}_s, \bar{a}\right)$, $(\bar{i}'_r, \bar{p}'_r, \bar{i}'_s, \bar{p}'_s, \bar{a}') \in M$ are redundant if $\pi\left(\bar{p}_r\right) = \pi(\bar{p}'_r)$, $\bar{p}_s = \bar{p}'_s$, $\bar{i}_s = \bar{i}'_s$ and $\bar{a} = \bar{a}'$.

Communication sets derived from data decompositions that replicate data may also contain redundancy. In our definition of communication sets, we consider communication to be necessary as long as there is a processor that owns a copy of the data needed by another processor. That means communication is generated even if the processor already owns a copy of the data. To eliminate this redundancy, we eliminate all the communication elements $\left(\bar{i}_r, \bar{p}_r, \bar{i}_s, \bar{p}_s, \bar{a}\right) \in M$ such that $\left(\bar{a}, \bar{p}_r\right) \in D$. Furthermore, two communication elements $\left(\bar{i}_r, \bar{p}_r, \bar{i}_s, \bar{p}_s, \bar{a}\right)$, $(\bar{i}'_r, \bar{p}'_r, \bar{i}'_s, \bar{p}'_s, \bar{a}') \in M$ are redundant due to replicated data if $\bar{p}_r = \bar{p}'_r$, $\bar{p}_s \neq \bar{p}'_s$, $\bar{i}_s = \bar{i}'_s$ and $\bar{a} = \bar{a}'$. The technique to eliminate this redundancy is similar to that of removing redundant communication due to self-reuse.

9.5.2. Communication Aggregation

Whether aggregation of small messages into large messages is necessary depends on the machine architecture. For example, machines such as the iWarp [65] and CM-5 [118] support fine-grain communication, while machines such as the Intel iPSC have significant overhead in processing every message. Again, we classify message aggregation into two kinds: *self aggregation*, where messages generated by different instances of the same access are aggregated, and *group aggregation*, where messages generated by different accesses are aggregated. For group aggregation, we simply aggregate all messages that have the same sender, receiver and dependence level into one message.

While group aggregation reduces the number of messages by a small constant, self aggregation can potentially eliminate many more messages. Our self aggregation algorithm also takes advantage of the partitions created by the exact data-flow analysis. All instances within the same communication set have the same dependence level. If the dependence level of a communication set is *k,* it is obviously legal to batch all the messages within an iteration of loop *k* and send the data at the end of the iteration. This can result in significant overhead reduction if loop *k* is not the innermost loop.

The algorithm to aggregate the communication of a communication set at level *k* is as follows. To generate the send code, we scan the communication set lexicographically in $\left(\bar{p}_s, i_{s_1}, \ldots, i_{s_{k-1}}, \overline{p_r}, i_{s_k}, \ldots, i_{s_n}, \bar{i}_r, \bar{a}\right)$ order. Each instance of the loops $\overline{p_s}, i_{s_1}, \ldots, i_{s_{k-1}}, \overline{p_r}$ produces one message, and each instance of the loops $i_{s_k}, \ldots, i_{s_n}$ contributes an item to the message. Redundancy elimination would have caused $\overline{i_r}$ to take on only the value of the earliest iteration on the receiver side using the value. Similarly, we create the receive loop nest by scanning the polyhedron in $\left(\overline{p_r}, i_{r_1}, \ldots, i_{r_{k-1}}, \overline{p_s}, \bar{i}_s, i_{r_k}, \ldots, i_{r_n}, \bar{a}\right)$ order. Iterations in loops $\bar{i}_s, i_{r_k}, \ldots, i_{r_n}$ use the data from the same message. Note that for each message, the order in which the sender packs the data is the same as the unpacking order. Figure 9-10 shows the receive and send code for the first context in Figure 9-3 after communication aggregation.

9.5.2.1. Multi-casting

Many systems provide optimized routines for multi-casting. To take advantage of these routines, we need to determine if the same message is sent to multiple processors. We scan a communication set to be aggregated at level k lexicographically in $\left(\overline{p_s}, i_{s_1}, \ldots, i_{s_{k-1}}, \overline{p_r}, \bar{a}, i_{s_k}, \ldots, i_{s_n}, \bar{i}_r\right)$ order. If the bounds of $\bar{a}$ are independent of $\overline{p_r}$, the data sent to each processor are identical.

9.6. Related Work

There is a large body of research on language extensions and compiler support for distributed memory machines. Some notable projects are, Al [141], Blaze [100], Crystal [106], FORTRAN-D [85,139], Id Nouveau [123], Kali [114,98], Pandore [15], Pandore II [14],

```
if pr >= 1 and pr <= N / 32 then
     DO tr = 0, T
          ps = pr - 1
          receive data into buffer from processor ps
          index = 0
          DO is = 32 pr - 3, MIN(32 pr - 1, N - 3)
               ts = tr
               ir = is + 3
               a = ir - 3
               X[a] = buffer[index]
               index = index + 1
```

(a) Receive code after aggregated communication.

```
     if ps >= 0 and ps <= (N - 32)/ 32 then
     DO ts = 0, T
          pr = ps + 1
          index = 0
          DO is = 32 ps + 29, MIN(32 ps + 31, N - 3)
               tr = ts
               ir = is + 3
               a = is
               buffer[index] = X[a]
               index = index + 1
          send the data in buffer to processor pr
```

(b) Send code after aggregated communication

Figure 9-10. Aggregated communication

SUPERB [62] and Vienna Fortran [35,36]. Many of these efforts converged on the development of High Performance Fortran (HPF) as an industry-wide standard language to support distributed memory machines, which extends FORTRAN-90 with data decomposition information [83,99].

The current HPF compilers [27,69,17], as well as most of the previous compilers for distributed memory machines, use regular section descriptors [81] to summarize iteration and data spaces as well as communication. However, regular sections can be used only to precisely represent a limited domain of rectilinear, triangular or diagonal spaces, creating spurious communication. Our approach for communication code generation can handle any iteration and data spaces and communication patterns that can be represented using systems of linear inequalities. A recent compiler for HPF also uses a similar linear algebra framework [40]. However, our extension to linear inequalities, to allow symbolic coefficients, further expands this domain such that we can represent distributions with symbolic block sizes.

Two algorithms for merging loop nests were proposed contemporaneously by [34,41]. These algorithms use linear inequalities to identify the common ranges of iterations and split the iteration space. In addition, they introduce heuristics to limit the exponential growth of the program. A similar algorithm was later introduced by [93].

All the compilers for distributed address space machines use a location-centric approach to communication identification. Array privatization present the only opportunity for reducing spurious communication created by this approach. The value-centric approach we introduced creates communication only when there is a producer-consumer relationship. Recently, many studies have taken a fundamentally different approach for minimizing communication based on producer-consumer relationships [56,92,109]. These algorithms optimally reschedule each instance of each statement while maintaining the producer-consumer relationships.

9.7. Chapter Summary

This chapter presents three main results. First, we have developed a systematic approach, based on a mathematical model, for communication code generation. We can handle a large class of computation and data decompositions as well as complex array access functions within this framework. We represent data decompositions, computation decompositions, and communication as systems of linear inequalities. We have shown that the various code generation and communication optimization problems can be solved by projecting the polyhedra represented by systems of inequalities onto lower dimensional spaces. This method is applicable to both the location- and value-centric approaches. Many optimizations can be expressed within this framework.

Second, we have developed several communication optimizations within the same unified framework. These optimizations include eliminating redundant messages, aggregating messages, and hiding the communication latency by overlapping the communication with computation. These optimizations are essential to achieving an acceptable performance on distributed memory machines [123].

Third, we have proposed a value-centric approach to deriving the fine-grain communication for machines with a distributed address space. Previous approaches are location-centric: communication is derived from data decompositions; optimizations are performed using data dependence tests [19], an analysis that determines if accesses may refer to the same location. In this approach, code generation is performed from computation decompositions using a data-flow analysis technique that is based on values instead of locations. This approach enables a more general set of data and computation decompositions and allows for more communication optimizations.

10 Conclusion

From the inception of the first electronic computer, architects have been striving to design the ultimate computer by simply connecting many smaller ones [82]. Such multiprocessors, which can bypass many of the physical limitations of uniprocessor performance, were expected to become ubiquitous in computing. However, so far they have not achieved the predicted performance gains for general purpose computing, mainly because of the inability to create parallel software, either explicitly by a programmer or automatically by a compiler [59]. Parallel programs are hard to develop, difficult to debug and expensive to maintain. The current generation of parallelizing compilers cannot extract parallel performance from sequential programs even with extensive user intervention. Thus, the adoption of parallel computing has been much slower than that anticipated 30 years ago [59].

Recent developments in compiler technologies have the potential to deliver the much anticipated breakthrough in parallel computing [8]. Compilers have played a critical role in two recent major breakthroughs in performance for general purpose computing: reduced instruction set computers (RISC) and instruction level parallelism (ILP) in microprocessors. Compilers translate complex operations into simple instructions for RISC processors [38] and schedule instructions for parallel execution in microprocessors with ILP [102,58]. Since the compilers are able to perform these techniques consistently and reliably without any user intervention, these technologies have been widely used, helping to revolutionize the microprocessor.

For multiprocessors to be widely accepted as general purpose computers, compilers must consistently, predictably, and transparently, deliver good parallel performance on sequential programs. Achieving this goal presents a series of difficult challenges for the compiler writer. Scientific applications require the development of compilers that identify coarse-grain parallelism and perform memory optimizations. Non-scientific applications written in languages such as C and C++, require development of compilers with advanced techniques such as pointer alias analysis [145].

This thesis represents a step towards making multiprocessors accepted as general purpose computers. In this thesis, we have developed a set of compiler techniques that extract parallel performance from sequential dense matrix scientific applications. We have shown that a parallelizing compiler can obtain parallel speedups consistently without user intervention for this class of applications. This thesis makes the following contributions:

- We have shown that the linear inequalities framework is effective in parallelizing and optimizing scientific applications, and that general solutions to many of these compiler problems can be found in a systematic manner. We have used this framework extensively for many purposes, such as representing array summaries in interprocedural data-flow analysis, solving for the array reshapes, identifying modulo and division optimizations, generating communication code for distributed address-space machines, and performing communication optimizations. This framework is being used at Stanford and elsewhere for developing many other compiler algorithms, such as synchronization optimizations [140], interprocedural propagation in computation and data co-location information, predicated data-flow analysis, and communication analysis for software DSM [91]. This framework allowed us to rapidly prototype and test many algorithms and ideas for next-generation compilers.

- We have implemented an array analysis algorithm using an array summary representation based on lists of systems of linear inequalities. This approach is driven by the need to compute both location-based and value-based dependences. 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 designed the first algorithm capable of handling simple reshape patterns that occur in practice. Using integer projections, this algorithm handles array reshapes that occur in parameter passing, equivalences, and different common block declarations.

- We have developed a fully functional interprocedural parallelizer incorporating many advanced array analysis techniques, such as array privatization, array reduction and array reshape analysis. We have evaluated their effectiveness by parallelizing more than 115,000 lines of FORTRAN code from 39 programs in four benchmark suites, and obtaining a parallel coverage over 80% for more than three fourths of the programs. The robustness of the system enabled us to perform such a large and realistic experiment.

- We have developed the first compiler that automatically performs a full suite of data transformations on the array layouts to improve memory system performance of cache-coherent multiprocessors. Our data transformation model uses a combination of strip-mining and permutation transformations to restructure the layout of the data in the shared address space such that each processor is assigned a contiguous segment of memory.

- We have created a unified approach for communication code generation and optimizations for distributed address-space machines. We showed that optimizations such as eliminating redundant messages, aggregating messages, and hiding communication latency by overlapping the communication with computation can be formulated using the linear inequalities framework. We also proposed a novel value-centric approach to deriving fine-grain communication.

We have shown that a powerful and complete set of analyses and optimization techniques can significantly improve the parallel performance of sequential applications.

Bibliography

[1] A. Agarwal, D. Chaiken, G. D'Souza, K. Johnson, and D. Kranz et. al. "The MIT Alewife machine: A large-scale distributed memory multiprocessor." In *Scalable Shared Memory Multiprocessors*. Kluwer Academic Publishers, 1991.

[2] A. Agarwal, D. Kranz, and V. Natarajan. "Automatic paritioning of parallel loops for cache-coherent multiprocessors." In *Proceedings of the 1993 International Conference on Parallel Processing*, St. Charles, IL, August 1993.

[3] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, MA, second edition, 1986.

[4] J. R. Allen. "Unifying vectorization, parallelization, and optimization: The Ardent compiler." In L. Kartashev and S. Kartashev, editors, *Proceedings of the Third International Conference on Supercomputing*, 1988.

[5] E. R. Altman, R. Govindarajan, and G. R. Gao. "Scheduling and mapping: Software pipelining in the presence of structural hazards." In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation '95*, June 1995.

[6] S. P. Amarasinghe, J. M. Anderson, M. S. Lam, and A. W. Lim. "An overview of a compiler for scalable parallel machines." In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[7] S. P. Amarasinghe, J. M. Anderson, M. S Lam, and C.-W. Tseng. "An overview of the SUIF compiler for scalable parallel machines." In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 662–667, San Francisco, CA, February 1995.

[8] S. P. Amarasinghe, J. M. Anderson, C. S. Wilson, S.-W. Liao, R. S. French, M. W. Hall, B. R. Murphy, and M. S. Lam. "Multiprocessors from a software perspective." *IEEE Micro*, 16(3):52–61, June 1996.

[9] S. P. Amarasinghe and M. S. Lam. "Communication optimization and code generation for distributed memory machines." In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 126–138, Albuquerque, NM, June 1993.

[10] C. Ancourt and F. Irigoin. “Scanning polyhedra with do loops.” In *Proceedings of the Third ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 39–50, Williamsburg, VA, April 1991.

[11] M. Ancourt. *Génération Automatique de Codes de Transfert pour Multiprocesseurs à Mémoires Locales*. PhD thesis, Université Paris VI, March 1991.

[12] J. M. Anderson, S. P. Amarasinghe, and M. S. Lam. “Data and computation transformations for multiprocessors.” In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 166–178, Santa Barbara, CA, July 1995.

[13] J. M. Anderson and M. S. Lam. “Global optimizations for parallelism and locality on scalable parallel machines.” In *Proceedings of the SIGPLAN ’93 Conference on Programming Language Design and Implementation*, pages 112–125, Albuquerque, NM, June 1993.

[14] F. Andr’e, O. Ch’eron, and J.-L. Pazat. “Compiling sequential programs for distributed memory parallel computers with Pandore II.” In *Proceedings of the Third Workshop on Compilers for Parallel Computers*, pages 213–242, Vienna, Austria, July 1992.

[15] F. Andr’e, J. Pazat, and H. Thomas. “Pandore: A system to manage data distribution.” In J. Saltz and P. Mehrotra, editors, *Languages, Compilers, and Run-Time Environments for Distributed Memory Machines*. North-Holland, Amsterdam, The Netherlands, 1992.

[16] B. Appelbe and B. Lakshmanan. “Optimizing parallel programs using affinity regions.” In *Proceedings of the 1993 International Conference on Parallel Processing*, pages 246–249, St. Charles, IL, August 1993.

[17] Applied Parallel Research, Placerville, CA. *Forge 90 Distributed Memory Parallelizer: User’s Guide*, version 8.0 edition, 1992.

[18] V. Balasundaram and K. Kennedy. “A technique for summarizing data access and its use in parallelism enhancing transformations.” In *Proceedings of the SIGPLAN ’89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989.

[19] U. Banerjee. *Dependence Analysis for Supercomputing*. Kluwer Academic Publishers, Boston, MA, 1988.

[20] U. Banerjee, R. Eigenmann, A. Nicolau, and D. Padua. “Automatic program parallelization.” *Proceedings of the IEEE*, 81(2):211–243, February 1993.

[21] B. Bixby, K. Kennedy, and U. Kremer. “Automatic data layout using 0-1 integer programming.” In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 111–122, Montreal, Canada, August 1994.

[22] W. Blume, R. Doallo, R. Eigenmann, J. Grout, J. Hoeflinger, T. Lawrence, J. Lee, D. Padua, Y. Paek, B. Pottenger, L. Rauchwerger, and P. Tu. “Parallel programming with polaris.” *IEEE Computer*, 29(12):78–82, December 1996.

[23] W. Blume and R. Eigenmann. “Performance analysis of parallelizing compilers on the Perfect Benchmarks programs.” *IEEE Transactions on Parallel and Distributed Systems*, 3(6):643–656, November 1992.

[24] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. “Effective automatic parallelization with Polaris.” *International Journal of Parallel Programming*, May 1995.

[25] W. Blume et al. “Polaris: The next generation in parallelizing compilers,.” In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994.

[26] W. J. Bolosky and M. L. Scott. “False sharing and its effect on shared memory performance.” In *Proceedings of the USENIX Symposium on Experiences with Distributed and Multiprocessor Systems (SEDMS IV)*, pages 57–71, San Diego, CA, September 1993.

[27] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. “A compilation approach for Fortran 90D/HPF compilers on distributed memory MIMD computers.” In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[28] T. Brandes. “The importance of direct dependences for automatic parallelism.” In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[29] M. Bromley, S. Heller, T. McNerney, and G. Steele, Jr. “Fortran at ten gigaflops: The Connection Machine convolution compiler.” In *Proceedings of the SIGPLAN ’91 Conference on Programming Language Design and Implementation*, Toronto, Canada, June 1991.

[30] E. Bugnion, J. M. Anderson, T. C. Mowry, M. Rosenblum, and M. S. Lam. “Compiler-directed page coloring for multiprocessors.” In *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VII)*, Cambridge, MA, October 1996.

[31] M. Burke and R. Cytron. “Interprocedural dependence analysis and parallelization.” In *Proceedings of the SIGPLAN ’86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[32] D. Callahan, K. Kennedy, and U. Kremer. “A dynamic study of vectorization in PFC.” Technical Report TR89-97, Dept. of Computer Science, Rice University, July 1989.

[33] S. Carr, K. S. McKinley, and C.-W. Tseng. “Compiler optimizations for improving data locality.” In *Proceedings of the Sixth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-VI)*, pages 252–262, San Jose, CA, October 1994.

[34] Z. Chamski. “Nested loop sequences: Towards efficient loop structures in automatic parallelization.” Technical Report RR-2094, INRIA Rennes, October 1993.

[35] B. Chapman, P. Mehrotra, and H. Zima. "Handling distributed data in Vienna Fortran procedures." In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[36] B. Chapman, P. Mehrotra, and H. Zima. "Programming in Vienna Fortran." *Scientific Programming*, 1(1):31–50, Fall 1992.

[37] S. Chatterjee, J. R. Gilbert, F. J. E. Long, R. Schreiber, and S.-H. Teng. "Generating local addresses and communication sets for data-parallel programs." *Journal of Parallel and Distributed Computing*, 26(1):72–84, April 1995.

[38] F. C. Chow. *A Portable Machine-Independent Global Optimizer–Design and Measurements*. PhD thesis, Stanford University, December 1983.

[39] M. Cierniak and W. Li. "Unifying data and control transformations for distributed shared memory machines." In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation '95*, June 1995.

[40] Fabien Coelho. *Contributions to HPF Compilation*. PhD thesis, Ecole des Mines de Paris, October 1996.

[41] J.-F. Collard, P. Feautrier, and T. Risset. "Construction of do loops from systems of affine constraints." Technical Report 93-15, Ecole normale supÈrieure de Lyon, May 1993.

[42] R. P. Colwell, R. P. Nix, J. J. O'Donnell, D. B. Papworth, and P. K. RodmanRodman. "A VLIW architecture for a trace scheduling compiler." In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, October 1987.

[43] K. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. "The ParaScope parallel programming environment." *Proceedings of the IEEE*, 81(2):244–263, February 1993.

[44] K. Cooper, M. W. Hall, and K. Kennedy. "A methodology for procedure cloning." *Computer Languages*, 19(2):105–117, February 1993.

[45] B. Creusillet and F. Irigoin. "Interprocedural array region analyses." In *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, August 1995.

[46] B. Creusillet and F. Irigoin. "Exact vs. approximate array region analyses." In *Proceedings of the 9th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, August 1996.

[47] G. Dantzig. *Linear Programming and Extensions*. Princeton University Press, Princeton, NJ, 1963.

[48] G. Dantzig and B. Eaves. "Fourier-Motzkin elimination and its dual." *Journal of Combinatorial Theory (A)*, 14:288–297, 1973.

[49] J. H. Edmondson et al. “Internal organization of the Alpha 21164, a 300-MHz 64-bit quad-issue CMOS RISC microprocessor.” *Digital Technical Journal*, 7(1), 1995. Special Edition.

[50] S. J. Eggers and T. E. Jeremiassen. “Eliminating false sharing.” In *Proceedings of the 1991 International Conference on Parallel Processing*, pages 377–381, St. Charles, IL, August 1991.

[51] S. J. Eggers and R. H. Katz. “The effect of sharing on the cache and bus performance of parallel programs.” In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-III)*, pages 257–270, Boston, MA, April 1989.

[52] R. Eigenmann, J. Hoeflinger, Z. Li, and D. Padua. “Experience in the automatic parallelization of four Perfect benchmark programs.” In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, Santa Clara, CA, August 1991. Springer-Verlag.

[53] P. Feautrier. “Array expansion.” In *Proceedings of the Second International Conference on Supercomputing*, St. Malo, France, July 1988.

[54] P. Feautrier. “Parametric integer programming.” *Operationnelle/Operations Research*, 22(3):243–268, September 1988.

[55] P. Feautrier. “Dataflow analysis of scalar and array references.” *International Journal of Parallel Programming*, 20(1):23–52, February 1991.

[56] P. Feautrier. “Towards automatic distribution.” Technical Report 92.95, Institut Blaise Pascal/Laboratoire MASI, December 1992.

[57] D. M. Fenwick, D. J. Foley, W. B. Gist, S. R. VanDoren, and D. Wissell. “The Alphaserver 8000 series: High-end server platform development.” *Digital Technical Journal*, 7(1), 1995. Special Edition.

[58] J. Fisher. “Trace scheduling: A technique for global microcode compaction.” *IEEE Transactions on Computers*, C-30(7):478–490, July 1981.

[59] M. J. Flynn. “Parallel processors were the future...and may yet be.” *IEEE Computer*, 29(12):151–152, December 1996.

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

[61] D. Gannon, W. Jalby, and K. Gallivan. “Strategies for cache and local memory management by global program transformation.” *Journal of Parallel and Distributed Computing*, 5(5):587–616, October 1988.

[62] M. Gerndt. *Automatic Parallelization for Distributed-Memory Multiprocessing Systems*. PhD thesis, University of Bonn, December 1989.

[63] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics*. Addison-Wesley, Reading, MA, 1989.

[64] E. D. Granston and A. Veidenbaum. “Detecting redundant accesses to array data.” In *Proceedings of Supercomputing ’91*, Albuquerque, NM, November 1991.

[65] T. Gross, S. Hinrichs, G. Lueh, D. O’Hallaron, J. Stichnoth, and J. Subhlok. “Compiling task and data parallel programs for iWarp.” In *Proceedings of the Workshop on Languages, Compilers, and Run-Time Environments for Distributed Memory Multiprocessors*, Boulder, CO, October 1992.

[66] T. Gross and P. Steenkiste. “Structured dataflow analysis for arrays and its use in an optimizing compiler.” *Software—Practice and Experience*, 20(2):133–155, February 1990.

[67] J. Grout. “Inline expansion for the polaris research compiler.” Master’s thesis, University of Illinois at Urbana-Champaign, May 1995.

[68] M. Gupta and P. Banerjee. “Demonstration of automatic data partitioning techniques for parallelizing compilers on multicomputers.” *IEEE Transactions on Parallel and Distributed Systems*, 3(2):179–193, March 1992.

[69] M. Gupta, S. Midkiff, E. Schonberg, V. Seshadri, K. Wang, D. Shields, W.-M. Ching, and T. Ngo. “An hpf compiler for the ibm sp2.” In *Proceedings of Supercomputing ’95*, San Diego, CA, December 1995.

[70] M. Haghighat and C. Polychronopoulos. “Symbolic analysis: A basis for parallelization, optimization, and scheduling of programs.” In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[71] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. “Detecting coarse-grain parallelism using an interprocedural parallelizing compiler.” In *Proceedings of Supercomputing ’95*, San Diego, CA, December 1995.

[72] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam. “Interprocedural parallelization analysis: Preliminary results.” Technical Report CSL-TR-95-665, Dept. of Computer Science, Stanford University, March 1995.

[73] M. W. Hall, J. M. Anderson, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, E. Bugnion, and M. S. Lam. “Maximizing multiprocessor performance with the suif compiler.” *IEEE Computer*, 29(12):84–89, December 1996.

[74] M. W. Hall, J. Mellor-Crummey, A. Carle, and R. Rodriguez. “FIAT: A framework for interprocedural analysis and transformation.” In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993.

[75] M. W. Hall, B. R. Murphy, and S. P. Amarasinghe. “Interprocedural analysis for parallelization: Design and experience.” In *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing*, pages 650–655, San Francisco, CA, February 1995.

[76] M. W. Hall, B. R. Murphy, S. P. Amarasinghe, S.-W. Liao, and M. S. Lam. "Interprocedural analysis for parallelization." In *Proceedings of the 8th International Workshop on Languages and Compilers for Parallel Computing*. Springer-Verlag, August 1995.

[77] L. Hamey, J. Webb, and I. Wu. "An architecture independent programming language for low-level vision." *Computer Vision, Graphics, and Image Processing*, 48(2):246–264, November 1989.

[78] S. W. Haney. "Is c++ fast enough for scientific computing?" *Computers in Physics*, 8(6):690–694, November 1994.

[79] W.L. Harrison. "The interprocedural analysis and automatic parallelization of Scheme programs." *Lisp and Symbolic Computation*, 2(3/4):179–396, October 1989.

[80] P. Havlak. *Interprocedural symbolic analysis*. PhD thesis, Rice University, Dept. of Computer Science, May 1994.

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

[82] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[83] High Performance Fortran Forum. "High Performance Fortran language specification." *Scientific Programming*, 2(1-2):1–170, 1993.

[84] M. Hind, M. Burke, P. Carini, and S. Midkiff. "An empirical study of precise interprocedural array analysis." *Scientific Programming*, 3(3):255–271, 1994.

[85] S. Hiranandani, K. Kennedy, and C.-W. Tseng. "Compiling Fortran D for MIMD distributed-memory machines." *Communications of the ACM*, 35(8):66–80, August 1992.

[86] F. Irigoin. "Interprocedural analyses for programming environments." In *NSF-CNRS Workshop on Evironments and Tools for Parallel Scientific Programming*, September 1992.

[87] F. Irigoin, P. Jouvelot, and R. Triolet. "Semantical interprocedural parallelization: An overview of the PIPS project." In *Proceedings of the 1991 ACM International Conference on Supercomputing*, Cologne, Germany, June 1991.

[88] T. Jeremiassen and S. Eggers. "Reducing false sharing on shared-memory multiprocessors through compile-time data transformations." In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 179–188, Santa Barbara, CA, July 1995.

[89] Y. Ju and H. Dietz. "Reduction of cache coherence overhead by compiler data layout and loop transformation." In U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing, Fourth International Workshop*, pages 344–358, Santa Clara, CA, August 1991. Springer-Verlag.

[90] J. Kam and J. Ullman. “Global data flow analysis and iterative algorithms.” *Journal of the ACM*, 23(1):159–171, January 1976.

[91] P. Keleher and C.-W. Tseng. “Improving the compiler/software dsm interface: Preliminary experiences.” In *Proceedings of the First SUIF Compiler Workshop*, Stanford University, Stanford, CA, January 1996.

[92] W. Kelly and W. Pugh. “Minimizing communication while preserving parallelism.” In *Proceedings of the 1996 ACM International Conference on Supercomputing*, pages 52–60, May 1996.

[93] W. Kelly, W. Pugh, and E. Rosser. “Code generation for multiple mappings.” Technical Report CS-TR-3317.1, Dept. of Computer Science, University of Maryland, December 1994.

[94] Kendall Square Research, Waltham, MA. *KSR1 Principles of Operation*, revision 6.0 edition, October 1992.

[95] K. Kennedy, N. Nedeljkovic, and A. Sethi. “A linear-time algorithm for computing the memory access sequence in data-parallel programs.” In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, July 1995.

[96] B. W. Kernighan and R. Pike. *The UNIX Programming Environment*. Prentice Hall Inc, Eaglewood Cliffs, NJ, 1984.

[97] D. Klappholz, K. Psarris, and X. Kong. “On the perfect accuracy of an approximate subscript analysis test.” In *Proceedings of the 1990 ACM International Conference on Supercomputing*, Amsterdam, The Netherlands, June 1990.

[98] C. Koelbel. “Compile-time generation of regular communications patterns.” In *Proceedings of Supercomputing ’91*, pages 101–110, Albuquerque, NM, November 1991.

[99] C. Koelbel, D. Loveman, R. Schreiber, G. Steele, Jr., and M. Zosel. *The High Performance Fortran Handbook*. The MIT Press, Cambridge, MA, 1994.

[100] C. Koelbel, P. Mehrotra, and J. Van Rosendale. “Semi-automatic domain decomposition in BLAZE.” In S. Sahni, editor, *Proceedings of the 1987 International Conference on Parallel Processing*, pages 521–524, St. Charles, IL, August 1987.

[101] J. Kuskin, D. Ofelt, M. Heinrich, , J. Heinlein, R. Simoni, K. Charachorloo, J. Chapin, D. nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. “The stanford flash multiprocessor.” In *Proceedings of the 21th International Symposium on Computer Architecture*, pages 302–313, Chicago, IL, April 1994.

[102] M. S. Lam. “Software pipelining: An effective scheduing technique for VLIW machines.” In *Proceedings of the SIGPLAN ’88 Conference on Programming Language Design and Implementation*, Atlanta, GA, June 1988.

[103] M. S. Lam, E. E. Rothberg, and M. E. Wolf. "The cache performance and optimizations of blocked algorithms." In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-IV)*, pages 63–74, Santa Clara, CA, April 1991.

[104] W. Landi, B. Ryder, and S. Zhang. "Interprocedural modification side effect analysis with pointer aliasing." In *Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, Albuquerque, NM, June 1993.

[105] D. Lenoski, J. Laudon, T. Joe, D. Nakahira, L. Stevens, A. Gupta, and J. Hennessy. "The DASH prototype: Implementation and performance." In *Proceedings of the 19th International Symposium on Computer Architecture*, pages 92–105, Gold Coast, Australia, May 1992.

[106] J. Li and M. Chen. "Compiling communication-efficient programs for massively parallel machines." *IEEE Transactions on Parallel and Distributed Systems*, 2(3):361–376, July 1991.

[107] J. Li and M. Chen. "The data alignment phase in compiling programs for distributed-memory machines." *Journal of Parallel and Distributed Computing*, 13(2):213–221, October 1991.

[108] Z. Li and P. Yew. "Efficient interprocedural analysis for program restructuring for parallel programs." In *Proceedings of the ACM SIGPLAN Symposium on Parallel Programming: Experience with Applications, Languages, and Systems (PPEALS)*, New Haven, CT, July 1988.

[109] A. W. Lim and M. S. Lam. "Maximizing parallelism and minimizing synchronization with affine transforms." In *Proceedings of the Twenty-forth Annual ACM Symposium on the Principles of Programming Languages*, January 1997.

[110] D. E. Maydan. *Accurate Analysis of Array References*. PhD thesis, Dept. of Computer Science, Stanford University, September 1992.

[111] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. "Data dependence and data-flow analysis of arrays." In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, New Haven, CT, August 1992.

[112] D. E. Maydan, S. P. Amarasinghe, and M. S. Lam. "Array data-flow analysis and its use in array privatization." In *Proceedings of the Twentieth Annual ACM Symposium on the Principles of Programming Languages*, pages 2–15, Charleston, SC, January 1993.

[113] D. E. Maydan, J. L. Hennessy, and M. S. Lam. "Effectiveness of data dependence analysis." In *Proceedings of the NSF-NCRD Workshop on Advanced Compilation Techniques for Novel Architectures*, 1992.

[114] P. Mehrotra and J. Van Rosendale. "Programming distributed memory architectures using Kali." In *Advances in Languages and Compilers for Parallel Computing*, Irvine, CA, August 1990. The MIT Press.

[115] P. Michielse. "Programming the convex exemplar series spp system." In J. Dongarra and J. Wasniewski, editors, *Proceedings of the first International Workshop in Parallel Scientific Computing*, pages 374–382. Springer-Verlag, June 1994.

[116] T. Mowry, M. S. Lam, and A. Gupta. "Design and evaluation of a compiler algorithm for prefetching." In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-V)*, pages 62–73, Boston, MA, October 1992.

[117] E. Myers. "A precise inter-procedural data flow algorithm." In *Conference Record of the Eighth Annual ACM Symposium on the Principles of Programming Languages*, Williamsburg, VA, January 1981.

[118] J. Palmer and G. Steele, Jr. "Connection Machine model CM-5 system overview." In *Frontiers '92: The 4th Symposium on the Frontiers of Massively Parallel Computation*, McLean, VA, October 1992.

[119] W. Pugh. "The Omega test: A fast and practical integer programming algorithm for dependence analysis." In *Proceedings of Supercomputing '91*, Albuquerque, NM, November 1991.

[120] W. Pugh. "A practical algorithm for exact array dependence analysis." *Communications of the ACM*, 35(8):102–114, August 1992.

[121] W. Pugh and D. Wonnacott. "Eliminating false data dependences using the Omega test." In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, San Francisco, CA, June 1992.

[122] H. Ribas. "Obtaining dependence vectors for nested-loop computations." In *Proceedings of the 1990 International Conference on Parallel Processing*, St. Charles, IL, August 1990.

[123] A. Rogers and K. Pingali. "Process decomposition through locality of reference." In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, Portland, OR, June 1989.

[124] C. Rosene. *Incremental Dependence Analysis*. PhD thesis, Dept. of Computer Science, Rice University, March 1990.

[125] R. G. Scarborough and H. G. Kolsky. "A vectorizing Fortran compiler." *IBM Journal of Research and Development*, 30(2):163–171, March 1986.

[126] M. Schlansker and M. McNamara. "The Cydra 5 computer system architecture." In *Proceedings of the 1988 IEEE International Conference on Computer Design: VLSI in Computers and Processors (ICCD '88)*, October 1988.

[127] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley and Sons, Chichester, Great Britain, 1986.

[128] M. Sharir and A. Pnueli. "Two approaches to interprocedural data flow analysis." In S. Muchnick and N.D. Jones, editors, *Program Flow Analysis: Theory and Applications*. Prentice Hall Inc, 1981.

[129] T. J. Sheffler, R. Schreiber, J. R. Gilbert, and S. Chatterjee. "Aligning parallel arrays to reduce communication." In *Frontiers '95: The 5th Symposium on the Frontiers of Massively Parallel Computation*, pages 324–331, McLean, VA, February 1995.

[130] O. Shivers. *Control-Flow Analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, School of Computer Science, Pittsburgh, PA, May 1991.

[131] J. P. Singh and J. L. Hennessy. "An empirical investigation of the effectiveness of and limitations of automatic parallelization." In *Proceedings of the International Symposium on Shared Memory Multiprocessors*, Tokyo, Japan, April 1991.

[132] J.P. Singh, T. Joe, A. Gupta, and J. L. Hennessy. "An empirical comparison of the Kendall Square Research KSR-1 and Stanford DASH multiprocessors." In *Proceedings of Supercomputing '93*, pages 214–225, Portland, OR, November 1993.

[133] Stanford SUIF Compiler Group. "SUIF: A parallelizing & optimizing research compiler." Technical Report CSL-TR-94-620, Computer Systems Laboratory, Stanford University, May 1994.

[134] O. Temam, E. D. Granston, and W. Jalby. "To copy or not to copy: A compile-time technique for assessing when data copying should be used to eliminate cache conflicts." In *Proceedings of Supercomputing '93*, pages 410–419, Portland, OR, November 1993.

[135] J. Torrellas, M. S. Lam, and J. L. Hennessy. "Shared data placement optimizations to reduce multiprocessor cache miss rates." In *Proceedings of the 1990 International Conference on Parallel Processing*, pages 266–270, St. Charles, IL, August 1990.

[136] E. Torrie, C-W. Tseng, M. Martonosi, and M. W. Hall. "Evaluating the impact of advanced memory systems on compiler-parallelized codes." In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, June 1995.

[137] R. Triolet, F. Irigoin, and P. Feautrier. "Direct parallelization of CALL statements." In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, Palo Alto, CA, June 1986.

[138] R. Triolet, F. Irigoin, and P. Feautrier. "Direct parallelization of call statements." In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, m SIGPLAN Notices 21(7)*, pages 176–185. ACM, July 1986.

[139] C.-W. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines*. PhD thesis, Dept. of Computer Science, Rice University, January 1993.

[140] C-W. Tseng. "Compiler optimizations for eliminating barrier synchronization." In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 144–155, Santa Barbara, CA, July 1995.

[141] P.-S. Tseng. "A parallelizing compiler for distributed memory parallel computers." In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, White Plains, NY, June 1990.

[142] P. Tu. *Automatic Array Privatization and Demand-Driven Symbolic Analysis*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1995.

[143] J. Uniejewski. "SPEC Benchmark Suite: Designed for today's advanced systems." SPEC Newsletter Volume 1, Issue 1, SPEC, Fall 1989.

[144] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. "SUIF: An infrastructure for research on parallelizing and optimizing compilers." *ACM SIGPLAN Notices*, 29(12):31–37, December 1994.

[145] Robert P. Wilson and Monica S. Lam. "Efficient context-sensitive pointer analysis for C programs." In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 1–12, June 1995.

[146] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Dept. of Computer Science, Stanford University, August 1992.

[147] M. E. Wolf and M. S. Lam. "A data locality optimizing algorithm." In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 30–44, Toronto, Canada, June 1991.

[148] M. E. Wolf and M. S. Lam. "A loop transformation theory and an algorithm to maximize parallelism." *IEEE Transactions on Parallel and Distributed Systems*, 2(4):452–471, October 1991.

[149] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. The MIT Press, Cambridge, MA, 1989.

[150] X3J3 Subcommittee. *American National Standard Programming Language Fortran (X3.9-1978)*. American National Standards Institute, New York, NY, 1978.

[151] H. Zima and B. Chapman. *Supercompilers for Parallel and Vector Computers*. Addison-Wesley, New York, NY, 1991.

Index