COMPILING FOR DISTRIBUTED MEMORY MULTIPROCESSORS BASED ON ACCESS REGION ANALYSIS

BY

YUNHEUNG PAEK

B.S., Seoul National University, 1988
M.S., Seoul National University, 1990

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1997

Urbana, Illinois

COMPILING FOR DISTRIBUTED MEMORY MULTIPROCESSORS
BASED ON ACCESS REGION ANALYSIS

Yunheung Paek, PhD
Department of Computer Science
University of Illinois at Urbana-Champaign, 1997
David A. Padua, Advisor

Distributed memory multiprocessors provide a scalable and affordable solution for high performance scientific computing. Accessing the data efficiently in these machines often requires complex program transformations based on accurate analysis of the memory access pattern. This work can be done manually; however, automatic transformation by a compiler is very important not only because it facilitates the development of new programs in the familiar sequential paradigm, but also because it may help port the vast amount of legacy code to new multiprocessor systems with little human effort.

This dissertation explores the applicability of fully automatic parallelizing techniques for distributed memory multiprocessors. In this research, an ordinary Fortran 77 program is assumed as input, and no information is required from the programmers. Only the abilities of the compiler are used to detect parallelism from the input program, to distribute data, and to control communication in the system. To achieve this goal, a variety of compiler techniques are used, including traditional compiling techniques and new techniques developed specifically with distributed memory architectures as the target. Combining these traditional and new techniques, various sequential benchmark programs are tested on one of the distributed memory machines.

ACKNOWLEDGMENTS

I would like give my special thanks to my advisor, David Padua, for inspiring me to start this work and for his continuous support throughout my research at Illinois. His insight and suggestions proved to be invaluable to me in this work.

I would like to thank Dr. Hwu and Dr. Kale for serving as members of the committee for the final oral examination of my doctoral degree. I am grateful to Jay Hoeflinger and Angeles Navarro for their valuable comments and assistance. Some of the work presented in this dissertation is based on cooperative studies with them. Also, I would like to thank Sheila Clark for helping me with many administrative affairs and for proofreading this dissertation. I would be remiss not to thank the former and current members of the Polaris group.

My gratitude also goes to the Korean Ministry of Education for the financial support they provided for the first three years, and to the Cray Research for generously granting machine time for the experiments reported in this dissertation. I am especially grateful to Thomas MacDonald for helping me access and understand the Cray T3D.

Most importantly, I praise God by dedicating this dissertation to Him. Without His consistent and unlimited love and grace to me for the past five years, this dissertation would never have been possible. Finally, I would like to thank my beloved wife, Yookyoung for her dedication to me within God's love.

TABLE OF CONTENTS

CHAPTER PAGE

1 INTRODUCTION 1
- 1.1 Code Transformation for Multiprocessor Systems 1
- 1.2 The Polaris Compiler 5
- 1.3 Code Transformation for Distributed Memory Machines in Polaris 8
  - 1.3.1 Overview of Code Transformation Strategies 9
  - 1.3.2 Research Issues 11
- 1.4 Organization of Dissertation 13

2 ACCESS REGION ANALYSIS 15
- 2.1 Motivation for Better Access Representations 16
  - 2.1.1 Dependence Analysis 16
  - 2.1.2 Communication Analysis 19
- 2.2 Description of Access Regions 19
  - 2.2.1 Components of Access Regions 20
  - 2.2.2 Abstract Access Form 22
  - 2.2.3 Generating Abstract Access Form with Coalescing 28
- 2.3 Characteristics of Access Regions 32
  - 2.3.1 Conjunctive Regions 34
  - 2.3.2 Complementary Regions 38
  - 2.3.3 Subregions 41
- 2.4 Basic Operations on Access Regions 45
  - 2.4.1 Intersection 46
  - 2.4.2 Subtraction 49
  - 2.4.3 Aggregation 52
- 2.5 Implementation in Polaris 53
- 2.6 Usage of Access Region Analysis 55
  - 2.6.1 Dependence Analysis 56
  - 2.6.2 Communication Analysis 61

3 AUTOMATIC CODE TRANSFORMATION 63
- 3.1 Target Parallel Programming and Machine Models 63
- 3.2 Polaris Modules for Code Transformation 65
  - 3.2.1 Parallelism Detection 66

3.2.2 Work Partitioning . . . 67
3.2.3 Data Privatization . . . 70
3.2.4 Data Distribution and Localization . . . 73

4 EXPERIMENTS AND PERFORMANCE ANALYSIS . . . 84
4.1 The Cray T3D . . . 85
4.1.1 Hardware Characteristics . . . 85
4.1.2 Shared Memory Programming . . . 87
4.2 Code Transformation for the Cray T3D . . . 89
4.2.1 Renaming . . . 91
4.2.2 Linearization . . . 92
4.2.3 Array Reshaping and Procedure Cloning . . . 92
4.3 Benchmark Programs . . . 94
4.4 Preliminary Experiments . . . 95
4.4.1 Code Transformation with Basic Techniques . . . 96
4.4.2 Experimental Results . . . 98
4.4.3 Performance Analysis . . . 99
4.5 Experiments with Advanced Techniques . . . 103
4.5.1 Experimental Results and Analysis . . . 104
4.5.2 Effectiveness of Data Localization . . . 107

5 CONCLUSIONS AND FUTURE WORK . . . 110
5.1 Achievements . . . 110
5.2 Additional Techniques to Further Improve Performance . . . 112
5.2.1 Redundant PUT/GET Elimination . . . 113
5.2.2 Access Sensitive Data Distribution . . . 114
5.2.3 Manual Experiments with Additional Techniques . . . 115
5.3 Conclusions . . . 117

BIBLIOGRAPHY . . . 119

VITA . . . 125

LIST OF TABLES

2.1 Comparison of parallel execution times 58

3.1 Loop execution times using the MDG benchmark on the Cray T3D 67
3.2 Comparison of reduction parallel loops execution times 70

4.1 Barrier performance on the T3D: times are the average of 5000 executions 86
4.2 Comparison of CRAFT and HPF 88
4.3 Benchmark programs used in the experiments 95
4.4 INTERF_do1000 and INTRAF_do1000 on the T3D 102
4.5 Impact of the techniques on the benchmarks 104
4.6 Performance improvement due to data localization 107

LIST OF FIGURES

1.1 Components of the Polaris compiler 6
1.2 Parallel performance of Polaris on SGI Challenge 8 processors 7
1.3 Code example to illustrate the data distribution issue 10
1.4 Code transformation for distributed memory multiprocessors in Polaris 11

2.1 Simplified code from MDG 17
2.2 Code from TFFT2 after inlining and induction variable substitution 18
2.3 Code example for the Access Region representation 21
2.4 READ accesses to array `X` within subroutine `foo` 23
2.5 Isomorphic regions for the access to array `Z` 25
2.6 Simplified code example from TFFT2 in SPEC95fp benchmarks 27
2.7 Simplified code example from TRFD in Perfect benchmarks 28
2.8 A loop in the SDOT routine from the BLAS library 29
2.9 Algorithm sketch for coalescing a region 30
2.10 Triangular loop example 31
2.11 Procedure for the operations on the Access Region 34
2.12 Access descriptors for the accesses in Figure 2.4 34
2.13 Aggregation of two regions $\mathcal{R}_1$ and $\mathcal{R}_2$ 37
2.14 Decompositions of access region $\bar{\mathcal{R}} = (\mathcal{A}^2_{16} + 2, \cdots)$ 38
2.15 $\mathcal{R}_1 \cap \mathcal{R}_2 = (\mathcal{A}^2_{18}, \cdots) \cap (\mathcal{A}^3_{12} + 3, \cdots) \implies (\mathcal{A}^6_6 + 6, \cdots)$ 47
2.16 $\mathcal{R}_1 - \mathcal{R}_2 \implies (\mathcal{A}^2_2, \cdots), (\mathcal{A}^6_6 + 4, \cdots), (\mathcal{A}^6_6 + 8, \cdots), (\mathcal{A}^2_2 + 16, \cdots)$ 50
2.17 Simplified code example in ARC2D from the Perfect benchmarks 51
2.18 Region processing in Polaris 54
2.19 Use of the Region Processor 56
2.20 Simplified code example from TURB3D program 59
2.21 Code from MDG, with the induction variables substituted 62

3.1 Code example with a parallel loop in Polaris 66
3.2 SPMD code with the **cyclic** scheduled parallel loop from Figure 3.1 68
3.3 A parallel loop with reduction 68
3.4 SPMD code with the **block** scheduled parallel loop from Figure 3.3 69
3.5 Loop from the MDG benchmark, with induction variables substituted 71
3.6 Simplified code from the SU2COR benchmark after work partitioning 72
3.7 Code after applying data privatization to the code from Figure 3.6 73
3.8 Conceptual view of the shared data copying scheme for four processors 74
3.9 Subroutine `foo` in Figure 3.7 after data localization 76
3.10 Loop from Figure 3.5, parallelized after anti dependence has been removed . . . 77

3.11 Choices of copy-levels for a GET operation on `X` 78
3.12 Heuristic algorithm for the `polaris_put/get` generation 80

4.1 Code example with a parallel loop in CRAFT 88
4.2 Aliasing in the simplified code from HYDRO2D 91
4.3 Sequence and storage association rules for the dummy array `B` 93
4.4 Code before procedure cloning 93
4.5 Code after procedure cloning 94
4.6 Basic code transformation for the T3D in Polaris 96
4.7 Results of the preliminary experiments with 11 benchmarks 97
4.8 Speedup Analysis for SWIM 100
4.9 Speedup Analysis for MDG 101
4.10 Speedup Analysis for TRFD 102
4.11 Improved parallel speedups for 6 benchmarks on the T3D 105
4.12 Before applying the data copying scheme 108
4.13 After applying the data copying scheme 108

5.1 Code example for redundant PUT/GET elimination 114
5.2 Impact of data distribution strategies on the T3D 116

CHAPTER 1

INTRODUCTION

1.1 Code Transformation for Multiprocessor Systems

In the past two decades, as the limits of semiconductor technology are approached [42], techniques for the architectural design of multiprocessor systems have been in great demand, and significant progress has been made in that research. As a result, these techniques are now commonly referred to as major breakthroughs in high-performance computing.

In contrast, the programming of these multiprocessor systems, whose purpose is to fully utilize these systems, still remains in its infancy. As a consequence, until now, programmers were usually required to write explicitly parallel programs by hand, posing many difficulties for the users who needed a multiprocessor system for high-performance computing.

One of these difficulties was that programmers needed to have a thorough understanding of the characteristics of the target multiprocessor system and the applications they were using. This increased software development costs and, even worse, made parallel programs hard to port to other multiprocessor systems because they sometimes had to be rewritten to take advantage

of the new systems. Above all, because programmers, by nature, are not adept at writing programs in parallel form, parallel programming was error-prone, and also, maintaining the programs written in parallel form was difficult. Without addressing these difficulties, progress in computational science and engineering would have remained hampered.

Parallelizing compilers, which are designed to automatically transform conventional sequential codes into parallel form, have been developed for the past twenty years as a solution for these problems. Such compilers consist of diverse modules that automate the code transformation procedure so as to minimize as much as possible the need for information from users. In general, these modules of parallelizing compilers can be divided largely into two components: *frontend* and *backend*. The frontend contains several compiler modules which attempt to expose or to identify implicit *parallelism* embedded in a program. Based on the exposed parallelism information, the backend tries to produce efficient parallel programs for the target machine, addressing many crucial code generation issues which are quite often machine-dependent.

The code transformation procedure of parallelizing compilers can be differentiated according to their target machines, as discussed above. In this section, we divide the multiprocessor systems into two classes, subject to their memory organization, and compare several code transformation issues of parallelizing compilers between these classes of machines.

The first class of machines we consider is *shared-memory multiprocessors with Uniform Memory Access(UMA) architecture*. Much work [13, 43, 58, 73] has been done in recent years on automatic parallelization of conventional sequential codes for these machines. This work has led to significant progress and has opened the door to several new strategies currently under study.

Parallelization techniques for UMA machines are better understood than those for other classes of machines, mainly because of the simplicity of programming UMA machines. In the code transformation for UMA machines, parallelism is much more important than any other issue. Therefore, the parallelizing compiler's frontend tasks for these machines are usually very complex, while the backend tasks may be as simple as directly mapping the parallelism information exposed by the frontend to target machine parallel codes through the use of various language constructs. For this reason, most research on code transformation for UMA machines has focused on developing compiler techniques that are primarily for the frontend modules of parallelizing compilers, such as:

- effective dependence detection techniques [14, 30, 60, 61, 73],
- dependence elimination techniques [21, 59, 69],
- the optimization techniques for parallel job and loop scheduling [46, 57], and
- the techniques for optimizing data locality and reuse in caches [38, 40, 72].

The work on parallelizing compilers for UMA machines also has given us the foundations necessary to tackle the study of compiler algorithms for more complex classes of machines like *distributed memory multiprocessors*, the target of the compiler project reported in this dissertation.

Distributed memory machines provide a scalable and affordable solution for high performance scientific computing. Due to the non-uniformity of memory access in these machines, the cost of local and remote memory accesses may differ from one to several orders of magnitude, thus significantly affecting parallel performance.

Consequently, in distributed memory architecture, parallelism is no longer the only factor determining good performance of parallel programs; another crucial factor that affects the efficiency of data access in the system, and therefore must be considered, is *communication*. This often requires the backend modules to be very complicated, because they need to perform program transformations based on the accurate analysis of the memory access pattern, and a thorough understanding of the machine architecture. This requirement of somewhat machine-dependent backend modules makes it more difficult to port parallel codes between distributed memory machines than between UMA machines. In all, these facts form a more compelling argument for the importance of parallelizing compilers for distributed memory machines over UMA machines.

The most common approach to program distributed memory systems is to use

- standard message-passing libraries [45, 49], or
- shared-memory programming models [3, 22, 27, 48, 64].

The message-passing programs based on the standard libraries are easy to port, but developing them requires sophisticated programming skills and a great deal of time because this work is usually done manually. Compared with message-passing models, programming in shared-memory models is easier because the software or underlying hardware hides from the programmers most of the complicated details involved in the data movement and distribution. However, obtaining good performance from shared-memory programs still requires much work on the part of the programmer. In order to improve upon this situation, research is actively underway to automate as much as possible the process of generating efficient parallel programs [6, 8, 9, 19, 41, 56].

This dissertation presents our recent work on the development of compiler algorithms to automatically transform full sequential Fortran 77 codes into parallel form for distributed memory multiprocessors. These algorithms have been implemented in *Polaris* [13, 51, 52], a parallelizing compiler developed by the authors and others. The code transformation does not rely on directives or any other type of information from the programmer. The compiler gathers from the source code all the information needed to expose parallelism, distribute data, and control data movement. This fully automatic parallelization is very important not only because it facilitates the development of new programs in the familiar sequential paradigm, but also because it may help to port the vast amount of legacy code to new multiprocessor systems.

1.2 The Polaris Compiler

As discussed in the previous section, parallelizing compilers have been extensively studied during the past twenty years or so. Especially for UMA machines, academic prototypes and commercial products of the compilers have been available for many years [71]. The Polaris compiler is one of these academic prototypes developed at the University of Illinois. Like most others, the main objective of Polaris is to develop and implement effective parallelization techniques for scientific programs.

Polaris was developed to overcome limitations in the analysis and transformation techniques implemented in other systems and to be robust enough to allow serious experimental studies. Figure 1.1 shows the two main phases of Polaris.

Taking a sequential Fortran 77 program as input, Polaris converts the program into its internal representation (IR) form [29]. The IR form includes an extensive collection of powerful

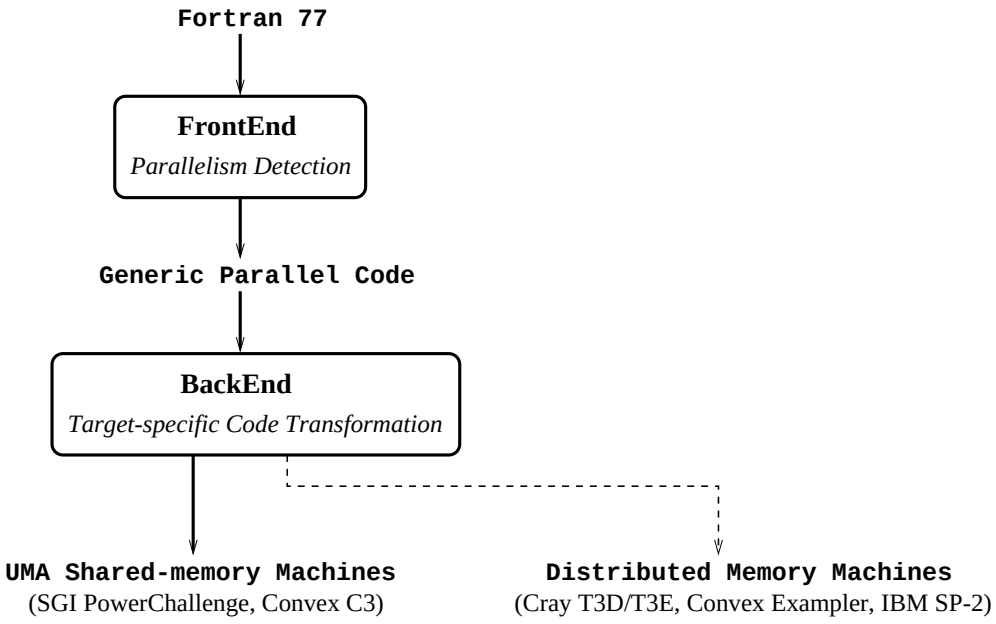

Figure 1.1: Components of the Polaris compiler

program manipulation operations that facilitate the implementation of compiler transformations in Polaris.

After a program is converted into IR form, it is analyzed and transformed by a sequence of compiler passes, the modules implemented in Polaris. The frontend passes, whose function is to identify implicit parallelism, include data dependence analysis, inlining, induction variable substitution, reduction recognition, and privatization [14, 31, 59, 69]. To support data dependence analysis expressions with symbolic values, additional modules for powerful symbolic constant and range manipulation [12] have been implemented in Polaris. The frontend outputs the parallel program in IR form extended with annotations that identify the parallelism detected by these modules.

Using the parallelism identified by the frontend, the backend applies machine-specific transformations and generates the target parallel program as output. As shown in Figure 1.1, Polaris has been used to restructure code for a variety of UMA shared memory multiprocessors and,

due mainly to the simplicity of the structure of UMA machines, good performance results do not require sophisticated backend optimizations. Therefore, during the past several years, the main focus of the Polaris group has been only on the frontend in order to accurately identify parallelism, with very little effort devoted to the backend. The experiment result has been that, on an extensive collection of programs gathered from the Perfect benchmarks, SPEC95fp benchmarks, and other sources from the National Center for Supercomputing Applications(NCSA), Polaris produce excellent speedups for the commercial UMA machines listed in Figure 1.1. Figure 1.2 demonstrates that Polaris substantially outperforms PFA, the native parallelizer of the SGI multiprocessors [13].

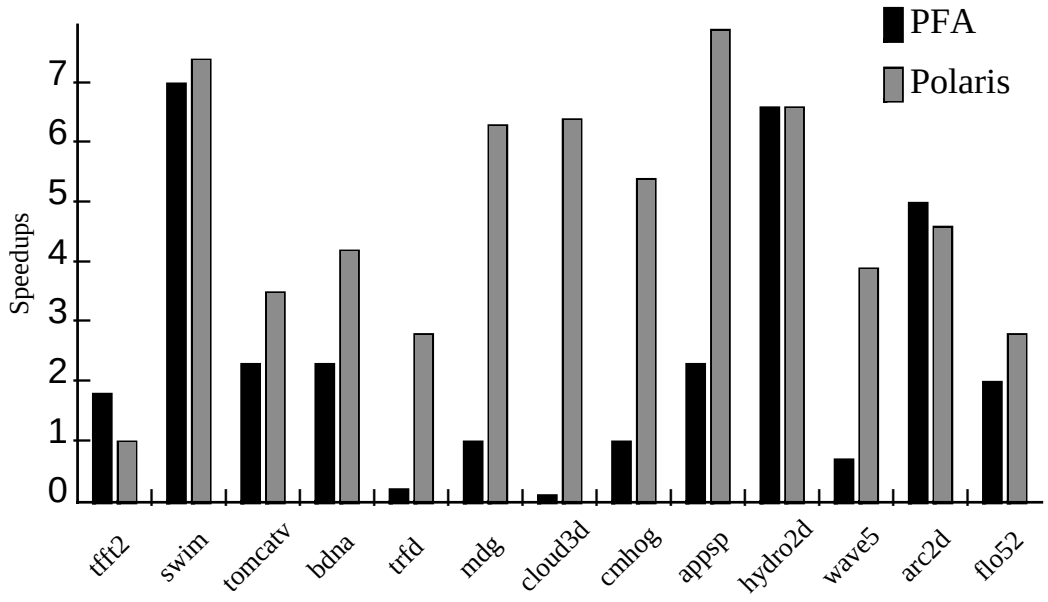

Figure 1.2: Parallel performance of Polaris on SGI Challenge 8 processors

Despite these successful results on UMA machines, the original version of Polaris was not capable of generating efficient parallel codes for distributed memory machines listed in Figure 1.1 because of the naive code transformation algorithms implemented in its backend. To achieve

reasonable parallel performance on these machines, it is necessary to deal with data distribution, data movement, and other issues involving communication. For the work presented in this dissertation, we have extended both the frontend and backend of Polaris (without particular efforts devoted to the backend) to generate efficient code for distributed memory machines as well. The distributed memory machines that Polaris is currently considering are listed in Figure 1.1.

1.3 Code Transformation for Distributed Memory Machines in Polaris

As discussed in the previous sections, speedups on distributed memory multiprocessors depend mainly on two important factors: parallelism and communication. If the programs are intrinsically serial or if the compiler cannot identify parallelism embedded within them, we cannot expect good parallel performance. Also, we have seen that high communication costs resulting from frequent remote memory accesses are a serious obstacle to attaining good performance in these systems. Good performance is possible only when both of these factors are dealt with through powerful dependence analysis, and through proper data and work partitioning across the system.

In this section, we will first provide an overview of several well-known code transformation strategies for distributed memory systems. Based on the analysis of these strategies, we chose to follow a different approach, which we also briefly discuss in this section. Finally, in this section, we will discuss several research issues, including the Access Region, a new region analysis technique that enhances the effectiveness of our approach.

1.3.1 Overview of Code Transformation Strategies

A frequently-followed approach when programming distributed memory systems can be called *data-partition oriented*. The approach consists of trying to find a good data partition [41, 6] that leads to low communication costs along with maximum possible parallelism. Work partitioning or computation assignment to processors is usually done as a function of the data layout, following strategies such as the owner-compute rule [68]. In some cases, these work partitioning rules may sacrifice load balancing for low communication costs and simplification of loop scheduling. Sometimes parallelism is reduced to obtain better data locality [4].

The data-partition oriented approach often involves complex data and work partitioning, plus occasional data redistribution [19, 56] which may be needed when the access patterns change. The techniques based on this approach work well on applications with a regular memory access pattern. Finding an optimal data partition, however, is often problematic because many scientific applications contain several different algorithms that require conflicting distribution strategies for fast execution. For example, in the subroutine `foo` in Figure 1.3, an array `X` is accessed horizontally in the first loop nest and vertically in the second. Conventional techniques may start the program with `X` distributed by rows, and redistribute `X` by columns somewhere between the two loop nests. If `M` is less than `N`, this redistribution will cause additional communication to move data unrelated to this computation. Also, it is impossible to have a communication-free data partition for `Y` because the access regions are overlapped across loop iterations.

Aliasing is another common factor that makes data partitioning difficult. In the example, if the dummy arrays `X` and `Y` are aliased to the same array (a common style of programming practice), then it becomes difficult, or impossible in some cases, to have different data distributions

```
subroutine foo(X,Y,Z,M,N)
real X(N,N), Y(*), Z(*)
   ...
doall I = 1, M
   X(I,1) = Z(2**(M-I))
   do J = 1, M
      X(I,J+1) = X(I,J) + Y(I+J)
   enddo
enddo
   ...
doall J = 1, M
   do I = 1, J
      X(I+1,J) = X(I,J) + Y(J-I+1)
   enddo
enddo
```

Figure 1.3: Code example to illustrate the data distribution issue

for them. Above all, to determine the proper data placement, we need additional information about all other arrays aliased with `X` and `Y` in other subroutines, which requires a very complex interprocedural analysis. Complex subscripting patterns, including subscripted-subscripts and non-affine expressions, are also well-known factors that prevent the identification of optimal solutions, as in the case of the array `Z` in the example.

The data-partition oriented approach is particularly effective when remote memory accesses in the target machine are quite expensive. This is the case in some systems, such as loosely-coupled multicomputers, and networks of workstations. But, our experience [54] tells us that we do not have to follow this approach for several existing machines [23, 25, 32, 33] which support very low remote memory latency; although the data partition issue still remains important in these machines, we have found that it is possible to obtain reasonable speedups for a wide spectrum of scientific applications on these machines even without sophisticated data partition algorithms.

Based on all these observations, we decided to follow an approach, summarized in Figure 1.4, that is not data-partition oriented. First, parallelism is exposed from the input program. Then, using the parallelism information, our algorithm tries to evenly distribute parallel work prior to considering any data distribution. Given the work distribution, the data regions that each individual processor accesses are identified. Based on the memory access pattern, data are privatized to reduce communication overhead and improve locality, and all non-privatized arrays are simply block-distributed across the distributed memories. Finally, most of the non-local accesses in the code are localized by copying shared data to private data using data prefetching and poststoring strategies.

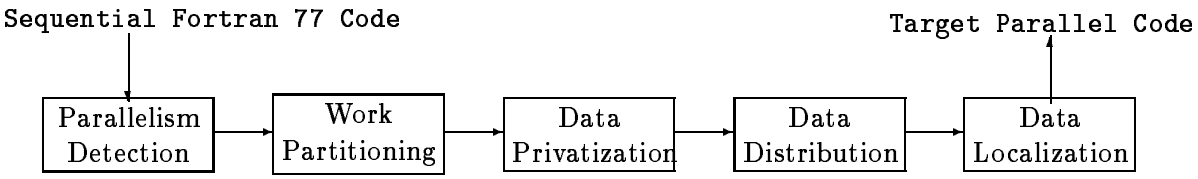

Figure 1.4: Code transformation for distributed memory multiprocessors in Polaris

We discuss in detail this transformation procedure in Chapter 3. As can be seen from the performance results presented in Chapter 4, data localization and other techniques discussed in this thesis compensate for the lack of sophisticated data distribution techniques at least in the T3D, one of the target machines we consider in this study.

1.3.2 Research Issues

The code transformation algorithm shown in Figure 1.4 requires three important compiler analysis techniques:

1. data dependence analysis

2. communication analysis

3. symbolic access region analysis

Data dependence analysis and communication analysis are the techniques used to address the two key issues that a parallelizing compiler for distributed memory machines should consider: parallelism and communication. In fact, detection of parallelism in sequential codes is the key ingredient to successful automatic parallelization for all types of modern multiprocessors. The increasing use of computers with large numbers of fast processors is making it even more important for compilers to find large amounts of parallelism within a program.

The data localization phase generates data prefetching and poststoring operations between local and remote memory. However, naive data copying operations may increase communication overhead too much and offset the advantages of the prefetching and poststoring strategies. Clearly, a sophisticated communication analysis is required to keep this overhead at a reasonable level.

To simultaneously support powerful dependence analysis and communication analysis, a parallelizing compiler must have an efficient and flexible way of representing and manipulating memory access patterns. We found that most existing representations, such as triplet notation (also called *regular section descriptor*) [14, 20, 68, 70] and convex regions [7, 26] sometimes must discard critical information. This prevents the detection of parallelism in certain loops as well as prevents efficient communication analysis for data copying. For this purpose, we have conducted an in-depth study of access patterns in many scientific benchmark programs to develop a new representation called the *Access Region* [53], a generalization of traditional triplet notation.

Access region analysis often needs effective symbolic analysis because it must handle in its analysis many symbolic terms (for example, program variables whose values are unknown at compile time). In our access region analysis, we employ various techniques already implemented in Polaris for this purpose, such as constant propagation, induction variable substitution, symbolic range propagation, and simplification of symbolic expressions [12].

1.4 Organization of Dissertation

The rest of this dissertation is organized as follows. In Chapter 2, we will discuss the access region analysis used in our code transformation algorithm. In the discussion, we will first present the motivation for a more powerful representation to be introduced to our access region analysis. Then, we will describe the Access Region and several basic operations on it. Finally, we will show how the Access Region can be used for dependence analysis and communication analysis.

In Chapter 3, we will discuss in detail our code transformation strategies for distributed memory multiprocessors. In the presentation, we first discuss the shared-memory programming model and the distributed memory machines that we are targeting by the transformation. After this, we will detail each phase of the code transformation procedure shown in Figure 1.4.

To evaluate the effectiveness of our techniques, we have conducted experiments on one of our target machines using various sequential codes from the SPEC95fp [62] and Perfect [11] benchmarks. We chose the Cray T3D [23] as our target machine in the experiment. In Chapter 3, the specification of the T3D and several code transformation issues that are T3D machine-dependent are first presented. The experimental results and the analysis on the results follows.

In Chapter 5, we will discuss a few additional techniques, and conclude this dissertation. Although these techniques are not used to obtain the results reported in Chapter 3, they have been found helpful to further improve performance in our hand experiments [47].

CHAPTER 2

ACCESS REGION ANALYSIS

Communication optimization [54, 55] for distributed memory systems depends heavily on *access region analysis*, the analysis of array subscripting patterns in a program. There are also many other important compiler problems depending on the analysis, including dependence detection [12] and array privatization [69].

For access region analysis, it is necessary to represent the array accesses in some standard fashion. For instance, Tu and Padua [70] approximated access regions with the *triplet notation* in their array privatization work. The same notation was used by Tseng [68] and Chatterjee, and Gilbert and Long [20] for message generation. To gain simplicity, Blume and Eigenmann [14] excluded the stride from the triplet notation in their dependence test, but this was at the expense of accuracy. *Convex regions* [26] express the geometrical shape of array accesses. They can be used with Fourier-Motzkin-based dependence tests [60, 67]. Balasundaram and Kennedy [7] proposed a simplified form of the convex region representation to detect task parallelism.

Such representations are designed to balance their compile-time efficiency and accuracy. Generally, design decisions leading to these forms have come down on the side of reducing

accuracy, or limiting accuracy to that needed for a specific compiler module in order to increase efficiency. But, limited accuracy can prevent compiler transformations. We have found that, in some important cases, compiler techniques based on traditional access region representations are not effective due to either the inefficiency or inaccuracy of the representations.

Based on our experience with the Perfect [11] and SPEC95fp [62] benchmarks, and other full scientific codes, we have developed the *Access Region*, a region access representation which takes advantage of the clear structure inherent in the array accesses of most scientific programs. Our design attempts to allow maximum accuracy without sacrificing efficiency.

Section 2.1 shows several instances encountered in our empirical experience that indicated the limitation of the triplet notation as the region representation, which motivated us to develop a new region representation for Polaris. Section 2.2 and 2.3 describe this new representation, and Section 2.4 discusses operations on the representation. Section 2.5 discusses the module being implemented in Polaris to process these representations, and Section 2.6 shows how it is used for the access region analysis in the parallelization for distributed memory multiprocessors.

2.1 Motivation for Better Access Representations

Our experiments indicate that compiler modules in the Polaris parallelizer which uses the triplet notation for array access patterns often are not effective due to the limited accuracy of the triplet notation. In this section, we will discuss some examples found in our experiments.

2.1.1 Dependence Analysis

Polaris has been successfully obtaining speedups for many scientific applications on a variety of shared-memory multiprocessors. However, we have seen that Polaris still fails to obtain

good speedups on some applications. We carefully studied these programs and found one reason is to be that the limited accuracy of the triplet notation made unable to handle several common access patterns occurring in these programs, thus leading to preventing Polaris from parallelizing loops in these programs. For instance, these programs commonly contain complex access patterns such as those involving multiple strides and diagonal access patterns. Also, these access patterns typically depend on many symbolic values in the program, often unknown at compile time. Consider the loop in Figure 2.1 which is extracted from `INTRAF_do1000`, one of the most time consuming loops in the MDG benchmark.

```
do I=1,M,N
    ...
   FX(I)=FX(I)*FHM
   FX(I+2)=FX(I+2)*FHM
   FX(I+1)=FX(I+1)*FOM
    ...
enddo
```

Figure 2.1: Simplified code from MDG

Polaris was not able to parallelize this loop because the value of `N` is statically unknown. However, it is easy to see that the loop is parallelizable for $N > 2$. In order to handle this case, we need a representation which can handle multiple strides, with conditions, and some mechanism to manipulate them to generate predicates for run-time tests. The triplet notation used by Polaris cannot support this manipulation and Polaris, therefore, simply serializes the loop. Without this mechanism, we might have to employ expensive run-time techniques [61, 63] in order to parallelize this loop.

As another example, consider the loop in Figure 2.2, which can be found in FFT programs such as the TFFT2 benchmark. It is the most important loop in TFFT2 and contains several complications:

- both the K and J loops are triangular loops;
- the subscript expressions of array references to X are non-affine; and
- the access pattern for X has multiple varying strides: 1 and 2^{L-1}.

```
do I = 0, 2**M-1
   do L = 1, (1+M)/2
      do J = 0, 2**(1+M-L)-1
         do K = 1, 2**(L-2)
              ...
            X(K+J*2**(L-1)) = ···
            X(K+J*2**(L-1)+2**(L-2)) = ···
              ...
         enddo
      enddo
      do J = 0, 2**(M-L)-1
         do K = 1, 2**(L-1)
              ...
            ··· = X(K+J*2**(L-1))
            ··· = X(K+J*2**(L-1)+2**M/2)
              ...
         enddo
      enddo
   enddo
enddo
```

Figure 2.2: Code from TFFT2 after inlining and induction variable substitution

The portion of the array X which is used in this loop is privatizable because the part of the array which is read is completely covered by the part that is written. However, any representation which cannot handle non-affine expressions cannot represent this access region. Although the Polaris dependence analyzers [12, 69] can handle non-affine expressions, the other complexities involved in these access patterns prevent them from privatizing X to parallelize this loop.

2.1.2 Communication Analysis

Communication optimization for distributed memory machines, such as the Cray T3D [5, 23, 39] and the SGI Origin [65], showed a need for gathering even more precise array access information for supporting efficient data movement and copying between distributed memories. For instance, for data movement in these systems, the communication analyzer often needs to selectively decide between small exact(MUST) regions and a large approximate(MAY) region in order to reduce remote memory latency. To meet this requirement, an access region representation must support the notion of accuracy.

2.2 Description of Access Regions

In our quest for a better access pattern representation, we considered the convex regions. However, forms that represent access patterns by sets of constraints typically must use a more general dependence test [60], which is not effective for non-affine expressions. Forms that use the triplet notation cannot handle such complicated access patterns, as discussed earlier, but lend themselves to more efficient manipulation in many parts of a compiler. We, therefore, decided to develop a new representation by combining the accuracy of convex regions with the efficiency of triplet notation, along with information about accuracy to inform the compiler modules of the accuracy of the region they are dealing with.

In our approach, we attempt to keep the exact region access pattern as long as possible. There are times when we cannot avoid losing accuracy; when this is the case, we mark that the access information is approximate.

2.2.1 Components of Access Regions

Within a program section such as a loop or a procedure, a reference to an array X is represented by $X(s(\mathcal{I}))$ where $s(\mathcal{I})$ is the subscript function defined on the set of indices $\mathcal{I} = \{i_1, i_2, \cdots, i_m\}$ in which each index i_k varies from l_k to u_k with stride s_k, denoted as $[l_k{:}u_k{:}s_k]$, within the program section. $s(\mathcal{I})$ need not be affine. If X is multi-dimensional, then the subscript expression is *linearized* [17] to generate $s(\mathcal{I})$. The array region $\mathcal{R}$ accessed by the array reference is represented by the four-tuple

$$\mathcal{R} = (\textit{Access Descriptor, Accuracy, Access Type, Predicates})$$

where the access descriptor is represented by two parts, the *access function* and *index ranges*:

$$(f(\mathcal{I}))[i_1 = l_1 : u_1 : s_1][i_2 = l_2 : u_2 : s_2] \cdots [i_m = l_m : u_m : s_m].$$

The accuracy is either

MUST or MAY

where accuracy MUST indicates that $\mathcal{R}$ is accurate and MAY indicates that it could be an overestimation. An access region $\mathcal{R}$ may have three values as the access type:

READ, WRITE, and $*$

where $\mathcal{R}$ has access type READ if $\mathcal{R}$ is only read by the corresponding reference(s). Similarly, it has access type WRITE if $\mathcal{R}$ is only written. Otherwise, $\mathcal{R}$ has access type $*$ (e.g., if $\mathcal{R}$ is both read and written). The predicates are conditions under which $\mathcal{R}$ is valid.

The access function $f(\mathcal{I})$ is the same as the corresponding subscript function $s(\mathcal{I})$, as long as $s(\mathcal{I})$ is a *monotonic function* [74] within the index ranges. Although many subscript functions encountered in scientific programs may not be affine, most are monotonic so that they can be used to generate monotonic access functions directly. Those few which are not monotonic may be converted to a monotonic function with a possible accuracy loss. For instance, in

Figure 2.3, the access region for IV is ((J-2)[J=1:N:1],MUST,READ,**T**[†]) since J-2 is clearly monotonic. However, for the access region for V, ((IV(J-2))[J=1:N:1],MUST,READ,**T**), we cannot determine whether IV(J-2) is monotonic unless we have knowledge about the contents of IV. In this case, we may convert the access region for V to ((J)[J=VLOW:VHIGH:1],MAY,$\cdots$), at the cost of accuracy of access information, to obtain a simplified monotonic function J which covers all the possible access region by V(IV(J-2)).

```
subroutine foo(X,Y,Z,W,M)
real V(VLOW:VHIGH),W(M,*),X(*),Y(M,*),Z(*) ···
 ···
do J = 1, N
   do I= 1 ,J
      W(2*J,I) = V(IV(J-2))
   enddo
   if (P) then
      Q(3*I) = ···
   endif
enddo
···
do I = 1, N, 1
   do J = 1, N, 1
      do K = 1, 4*J, 2
         X(I+5*J+K) = Z(N*J+I)
      enddo
      Z(J+N*I) = Y(C,D) + Y(I+1,I)
   enddo
enddo
```

Figure 2.3: Code example for the Access Region representation

Predicates add precision to $\mathcal{R}$. In Figure 2.3, even when the compiler cannot determine the value of P, the access region for Q can be represented by ((3I)[I=1:N:1],MUST,WRITE,P). Alternatively, for flow-insensitive analysis, we may represent this as ((3I)[I=1:N:1],MAY,WRITE,**T**). Also, when we linearize the subscript expression of the multi-dimensional array W in the figure,

[†]**T** represents TRUE, which means that there is no constraint on this region.

we might lose the original array dimension information. In order to avoid that, we can extract predicates from the array dimension information to produce the region

$$((2J+(I\text{-}1)M)[J=1:N:1][I=1:J:1],MUST,WRITE,1 \leq 2J \leq M).$$

In the operations in Section 2.4, this extra predicate will be used through the *range dictionary* [12] combined with various symbolic manipulation modules to supply accurate symbolic range information.

2.2.2 Abstract Access Form

We have seen that, for the most part, within limited sections of a program, the access regions of interest have a *regularity* of structure. By this we mean that the common accesses are very structured in real cases. Furthermore, quite often, related access regions have a *similarity* of structure. This is generally true because several references to a single array within a loop nest are generally accessed using the same loop indices and with similar subscript expressions. Experimental evidence for the regularity and similarity of array subscripting may be gleaned from the work of others [14, 60, 66] who note that their dependence tests, built to be effective and fast specifically for regular and similar access patterns, are successful for most access patterns encountered in real scientific programs. For a more specific description of similarity and regularity, consider the example in Figure 2.4.

We try to capture the properties of regularity in our region access representation in terms of two important characteristics: *strides* and *spans*. A regular access region $\mathcal{R}$ for an array is composed of a finite number of discrete array elements, arranged in a region according to regular strides. The collection of elements separated by the same stride stretches for a finite distance which we call a span. In Figure 2.4, the access made by array reference `X(5*I-4)`

```
subroutine foo(X)
   ...
do I = 1, 4
   ... = X(5*I-4)
   do J = 1, 3
      ... = X(5*I+J-4)
      ... = X(5*I+J-3)
   enddo
enddo
   ...
do K = 1, 5
   ... = X(4*K)
   ... = X(4*K-2)
enddo
   ...
```

Figure 2.4: READ accesses to array `X` within subroutine `foo`

has a stride of 5 and a corresponding span of 15 since the elements accessed made by the reference stretch from `X(1)` to `X(16)` with regular stride 5; this can be represented equivalently by triplet notation `X(1:16:5)`. Similarly, the accesses made by two references `X(5*I+J-4)` and `X(5*I+J-3)` are composed of two strides, 5 and 1, due to loop indices `I` and `J`, respectively, with the strides having their matching spans, 15 and 2.

We can find the similarity of accesses shared by three references, `X(5*I-4)`, `X(5*I+J-4)` and `X(5*I+J-3)`. Likewise, the accesses made by two references, `X(4*K)` and `X(4*K-2)`, can be found to be similar. These examples illuminate to us that those similarities are closely related to the strides and spans involved in those access patterns; that is,

"*Access patterns composed of the same (or almost same) strides and spans are similar*".

For the formal description of strides and spans, suppose that $f(\mathcal{I})$ is a monotonic access function defined on the set of indices $\mathcal{I} = \{i_1, i_2, \cdots, i_m\}$ where $l_k \leq i_k \leq u_k, 1 \leq k \leq m$. We first define the span of $f(\mathcal{I})$ due to $i_k \in \mathcal{I}$ to be the maximum distance moved by varying only i_k:

$$\delta_{i_k} = |f(i_1, \cdots, i_{k-1}, u_k, i_{k+1}, \cdots, i_m) - f(i_1, \cdots, i_{k-1}, l_k, i_{k+1}, \cdots, i_m)|.$$

In our Access Region notation, a zero span indicates a single element region, and a negative span indicates an empty region. For $\delta_{i_k} > 0$, the stride of the access due to i_k is defined to be

$$\sigma_{i_k} = |f(i_1, \cdots, i_{k-1}, i_k + s_k, i_{k+1}, \cdots, i_m) - f(i_1, \cdots, i_{k-1}, i_k, i_{k+1}, \cdots, i_m)|,$$

and it can be any integer number for $\delta_{i_k} \leq 0$.

Using the characteristics of span and stride, the access region $\mathcal{R}$ can be represented by another form, which we call the *abstract access form*, as follows:

$$\mathcal{R} = (\ \mathcal{A}^{\sigma_{i_1},\sigma_{i_2},\cdots,\sigma_{i_m}}_{\delta_{i_1},\delta_{i_2},\cdots,\delta_{i_m}} + l,\ \textit{Accuracy, Access Type, Predicates}\ )$$

where l is the *lower bound* of $\mathcal{R}$, which is the offset from zero of the first element in $\mathcal{R}$. In this form, the access descriptor is represented by a list of the spans and strides where we define $(\sigma_{i_k}, \delta_{i_k})$ to be the k-th *stride/span pair* of $\mathcal{R}$.

Using the abstract access form, we try to capture the similarity of the structures of different access regions. First, we define the 'degree of similarity' between access regions in terms of stride/span pairs and lower bounds.

Definition 2.1 *Access regions are* **isomorphic** *if they represent the same access pattern.*

As an example of isomorphic regions, consider the access pattern in Figure 2.5, which covers the range from `Z(0)` to `Z(11)` with stride 1. This access pattern could be represented by triple notation `Z(0:11:1)`, which is represented equivalently by access region $\mathcal{R}_z$ with descriptor $\mathcal{A}^1_{11}(+0)$, denoted in upper solid lines in the figure. The lower solid lines represent the access region $\mathcal{R}'_z$ with descriptor $\mathcal{A}^{1,2}_{1,10}$, and we see that $\mathcal{R}_z$ and $\mathcal{R}'_z$ are isomorphic because they represent the same access pattern. A region with access descriptor $\mathcal{A}^{3,1}_{9,2}$, denoted in dashed lines, is another isomorphic region of $\mathcal{R}_z$ and $\mathcal{R}'_z$. So is a region with access descriptor $\mathcal{A}^{1,3,6}_{2,3,6}$.

From the above examples, we learn that an access pattern can be represented by numerous access descriptors with different stride/span pairs. In fact, there are an infinite number of

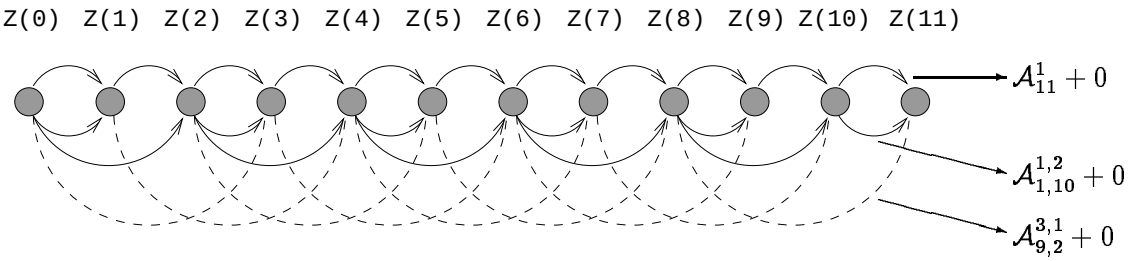

Figure 2.5: Isomorphic regions for the access to array Z

access descriptors for a region which represent the same region. To show this, let's consider the following theorem.

Theorem 2.1 *An access descriptor* $\mathcal{A}^{\sigma_1,\cdots,\sigma_k,\cdots,\sigma_m}_{\delta_1,\cdots,\delta_k,\cdots,\delta_m} + l$, *which represents a region* $\mathcal{R}$, *can be expanded to form other descriptors which also represent* $\mathcal{R}$ *by adding a stride/span pair* $(\sigma^*, 0)$ *in any position, such as* $\mathcal{A}^{\sigma_1,\cdots,\sigma^*,\sigma_k,\cdots,\sigma_m}_{\delta_1,\cdots,0,\delta_k,\cdots,\delta_m} + l$, *and* $\mathcal{A}^{\sigma_1,\cdots,\sigma_k,\sigma^*,\cdots,\sigma_m}_{\delta_1,\cdots,\delta_k,0,\cdots,\delta_m} + l$, *where* σ^* *can be any integer number.*

PROOF: By the definition of a zero span, it is obvious that adding a stride/span pair $(\sigma^*, 0)$ to an arbitrary access descriptor does not add any new elements to the access region which is represented by the descriptor, nor does it subtract any elements from the region. Therefore, the access region represented by the access descriptor with $(\sigma^*, 0)$ has the same elements as the original access region. For instance, consider a region represented by $\mathcal{A}^{1,2}_{1,10}$ in Figure 2.5. It is clear that none of the access descriptors, $\mathcal{A}^{\sigma^*,1,2}_{0,1,10}$, $\mathcal{A}^{1,\sigma^*,2}_{1,0,10}$ or $\mathcal{A}^{1,2,\sigma^*}_{1,10,0}$, change the original access pattern at all (this can be seen by drawing the access regions for each descriptor). □

By Theorem 2.1, we can see that a region represented by $\mathcal{A}^{1,2}_{1,10}$ also can be represented in infinite ways (e.g., by descritors $\mathcal{A}^{i_1,1,2}_{0,1,10}$, $\mathcal{A}^{i_1,i_2,1,2}_{0,0,1,10}$, $\mathcal{A}^{i_1,i_2,i_3,1,2}_{0,0,0,1,10}$, and so on, where i_k, $k \geq 1$, can be any number).

Given access regions $\mathcal{R}_1$ and $\mathcal{R}_2$, with lower bounds l_1 and l_2 respectively, we characterize the regions based on their stride/span pairs and lower bounds as follow;

1. $|l_1 - l_2|$ is the **distance** between the regions.

2. If the distance is 0, then the two regions are **aligned**. Otherwise, they are **non-aligned**.

3. Stride/span pairs (σ_1, δ_1) from $\mathcal{R}_1$ and (σ_2, δ_2) from $\mathcal{R}_2$ **match** if $\sigma_1 = \sigma_2$ and $\delta_1 = \delta_2$.

From these definitions on access regions, we draw the following theorem.

Theorem 2.2 *Suppose that two access descriptors $\mathcal{A}$ and $\mathcal{A}'$ have the same number of stride/span pairs and the same lower bounds. If each stride/span pair in $\mathcal{A}$ has a unique matching stride/span pair in $\mathcal{A}'$, then the regions represented by the two access descriptors are identical and, therefore, the two regions are isomorphic.*

PROOF: It is trivial to show that a region represented by access descriptor $\mathcal{A}^{\sigma_1,\cdots,\sigma_i,\cdots,\sigma_j,\cdots,\sigma_m}_{\delta_1,\cdots,\delta_i,\cdots,\delta_j,\cdots,\delta_m} + l$ can be represented equivalently by access descriptor $\mathcal{A}^{\sigma_1,\cdots,\sigma_j,\cdots,\sigma_i,\cdots,\sigma_m}_{\delta_1,\cdots,\delta_j,\cdots,\delta_i,\cdots,\delta_m} + l$. This tells us that the order of stride/span pairs in the access descriptor does not change the representation of the access region itself. □

Converting the access region to the abstract form helps us to identify the similarity between access patterns using Theorem 2.2. For instance, in Figure 2.3, the abstract access form

$$(\mathcal{A}^{N,1}_{N^2-N,N-1} + N + 1, \text{MUST}, \text{READ}, \mathbf{T})$$

is for the access region of `Z(N*J+I)`, ((NJ+I)[J=1:N:1][I=1:N:1],MUST,READ,**T**). Similarly, the form

$$(\mathcal{A}^{1,N}_{N-1,N^2-N} + N + 1, \text{MUST}, \text{WRITE}, \mathbf{T})$$

is for the access region of `Z(J+N*I)`, ((J+NI)[J=1:N:1][I=1:N:1],MUST,WRITE,**T**). Since we

can find corresponding matching stride/span pairs, the two regions are isomorphic. Also, they have the same lower bound $N+1$; thus, we can prove that the WRITE region of `Z(J+N*I)` and the READ region of `Z(N*J+I)` are exactly the same.

This abstract access form is not limited by the original array dimensionality. By linearization, for instance, we can represent access patterns for multi-dimensional arrays. In Figure 2.3, the array `Y` is accessed along a diagonal. We apply linearization to the subscript expression and represent it accurately as $\mathcal{A}^{M+1}_{(N-1)(M+1)} + 2$. Traditional triplet notation would have to express such a diagonal access pattern as `Y(2:N+1:1,1:N:1)`, which is inaccurate. The abstract form here makes it obvious that the access is regular, with a single stride. Also, notice that the access region for the single element `Y(C,D)` can be represented by $\mathcal{A}^{\sigma^*}_0 + \mathtt{C} + (\mathtt{D}-1)\mathtt{M}$ where σ^* can be any integer number.

Sometimes, a span or stride may not be a constant, varying on the values of indices in $\mathcal{I}$. Figure 2.6 shows a simplified version of code in Figure 2.2. The access region for `X(2**I+J)` is $((2^I+\mathrm{J})[\mathrm{I}{=}1{:}\mathrm{N}{:}1][\mathrm{J}{=}1{:}2^I{:}1],\mathrm{MUST},\mathrm{WRITE},\mathbf{T})$, and the access descriptor of its corresponding abstract form is $\mathcal{A}^{2^I,1}_{N,2^I-1} + 3$ where index I varies from 1 to N. Here, σ_I and δ_J vary depending on the value of index `I`, while σ_J and δ_I are a constant or a symbolic constant. This case typically happens in triangular loops or when the subscript functions are non-affine.

```
do I = 1, N, 1
   do J = 1, 2**I, 1
      X(2**I+J) = ···
   enddo
enddo
```

Figure 2.6: Simplified code example from TFFT2 in SPEC95fp benchmarks

Definition 2.2 *Suppose that $\mathcal{I}$ is a set of indices $\{i_1, i_2, \cdots, i_m\}$ within a program section, on which an access descriptor $\mathcal{A}_{\delta_{i_1},\delta_{i_2},\cdots,\delta_{i_m}}^{\sigma_{i_1},\sigma_{i_2},\cdots,\sigma_{i_m}} + l$ is defined. An index $i_k \in \mathcal{I}$ is defined to be* **region-dependent** *on an index $i_{k'} \in \mathcal{I}$ if either δ_{i_k} or σ_{i_k} is an expression containing $i_{k'}$.*

In Figure 2.6, index J is region-dependent on I because σ_J contains index I, and I is also region-dependent on itself. As another example in Figure 2.7, J is region-dependent on I because δ_J(=I-1) contains I.

```
do I = 1, N, 1
   do J = 1, I, 1
      X(I*(I-1)/2+J) = ···
   enddo
enddo
```

Figure 2.7: Simplified code example from TRFD in Perfect benchmarks

In order to apply the region operation techniques discussed in Section 2.4, we need to handle the region-dependent indices in $\mathcal{R}$ so that all strides and spans in the access descriptor are constant. In the next section, we describe how we could handle the region-dependent indices when we generate the abstract access form.

2.2.3 Generating Abstract Access Form with Coalescing

Although many access regions encountered in scientific applications are generated by multiple indices, they usually form stream accesses with simpler structures when they are aggregated. That is the case where the references to an array make a series of contiguous accesses with one index and jump over those accesses with another index to start a new section of accesses. As an example, note the loop in Figure 2.8. The references to both arrays SX and SY access five contiguous elements with stride 1 and jump the elements with stride 5 to the next section,

resulting in contiguous access to the arrays from the M-th element to the (N+4)-th element. Similar access patterns also can be commonly found in full scientific programs [13].

```
      do 50 I = M,N,5
         STEMP = STEMP + SX(I)*SY(I) + SX(I+1)*SY(I+1) +
     *       SX(I+2)*SY(I+2) + SX(I+3)*SY(I+3) + SX(I+4)*SY(I+4)
50    continue
```

Figure 2.8: A loop in the SDOT routine from the BLAS library

The access pattern generated by multiple indices with various strides is usually represented by an access region with the same number of stride/span pairs. However, as we discussed in Section 2.2.2, an access pattern can be represented by various isomorphic regions with a different number of stride/span pairs. In the case of contiguous access, like that shown in Figure 2.8, the pattern always can be represented by an access region with a single index. For instance, the access region representation ((I+J)[I=1:N:4][J=0:2:2],$\cdots$) has two indices with strides 2 and 4. Here, the index I is redundant because we can represent the same access region only with index J after combining the index ranges into the form ((J)[J=1:N+2:2],$\cdots$).

Coalescing is a technique for simplifying the region representation by eliminating these redundant indices, as described in Figure 2.9, where the form $\delta_{i_j}(u_k)$ refers to the value of the expression δ_{i_j} with u_k substituted for i_k. In the algorithm, the access region $\mathcal{R}$ is of the abstract access form, and indices i_j and i_k are defined in $\mathcal{R}$ with index ranges $[l_j{:}u_j{:}s_j]$ and $[l_k{:}u_k{:}s_k]$, respectively. `coalesce_region` determines whether i_k is redundant. If so, it removes i_k and combines the original index ranges to generate the new range for i_j. In order to determine whether i_k is redundant, the stride of i_k and stride/span pair of i_j must be examined to test if i_j can represent the same access region without i_k by adjusting the span δ_{i_j}.

```
coalesce_region(R, i_j, i_k) {
     if (σ_{i_j} divides σ_{i_k} and σ_{i_k} ≤ δ_{i_j} + σ_{i_j}) {
        remove redundant index i_k from R
        if (i_j is region-dependent on i_k)
           δ_{i_j} = δ_{i_j}(u_k) + δ_{i_k}
        else
           δ_{i_j} = δ_{i_j} + δ_{i_k}
     }
  }
```

Figure 2.9: Algorithm sketch for coalescing a region

Consider the example shown in Figure 2.3 to illustrate the colealescing operation. In the example, the access region for `Z(N*J+I)` has the access descriptor $\mathcal{A}^{N,1}_{N^2-N,N-1} + N + 1$. The `coalesce_region` routine converts the region to a form with a simpler descriptor $\mathcal{A}^{1}_{N^2-1}+N+1$, since $\sigma_I(=1)$ divides $\sigma_J(=N)$, and $\sigma_J \leq \delta_I+\sigma_I(=N)$. Similarily, we obtain the same coalesced region for `Z(J+N*I)` in the example.

When we gather array region information from the program and convert the information to an abstract form, we apply coalescing to eliminate unnecessary indices from $\mathcal{R}$. The coalescing operation for a region with m stride/span pairs needs, in the worst case, $m^2 - m$ invocations of routine `coalesce_region` for pair-wise comparisons.

According to our study, coalescing in many cases [12] helps us to remove non-constant strides or spans from the access descriptor in the abstract form. The access descriptor $\mathcal{A}^{2^I,1}_{N,2^I-1} + 3$ from TFFT2 in Figure 2.6, for example, can be coalesced to a simpler form $\mathcal{A}^{1}_{2(N+1)-1} + 3$ by removing the index I. The resulting abstract form, therefore, contains only constant strides and spans. Similarly, the original descriptor $\mathcal{A}^{1,I}_{\frac{I-1,N^2-N}{2}-1} + 3$ from TRFD in Figure 2.7 can be coalesced to $\mathcal{A}^{1}_{\frac{N^2+N}{2}-1} + 3$.

In some cases where coalescing cannot remove those indices from the access descriptor, we may convert the original access region to an overestimated MAY region in order to eliminate

them. For instance, if index j is region-dependent on a set of indices $i_1, \cdots, i_n$, then we apply the following heuritic rules;

1. the new stride $\sigma'_j = 1$

2. the new span $\delta'_j = max(\delta_j)$ subject to the index ranges $[i_1{=}l_1{:}u_1{:}s_1]\cdots[i_n{=}l_n{:}u_n{:}s_n]$

Consider the triangular loop in Figure 2.10 as an example. The access region for `X` is

$$(\mathcal{A}^{1,M}_{I-1,(N-1)M} + 1, \text{MUST}, \text{WRITE}, \mathbf{T})$$

after linearization. The `coalesce_region` algorithm cannot remove the index `I` in the access descriptor without converting the original region to

$$(\mathcal{A}^{1,M}_{N-1,(N-1)M} + 1, \text{MAY}, \text{WRITE}, \mathbf{T})$$

since $max(I-1) = N-1$ for I is in the range [I=1:N:1].

```
real X(M,*)
 ...
do I = 1, N, 1
   do J = 1, I, 1
      X(J,I) = ···
   enddo
enddo
```

Figure 2.10: Triangular loop example

Although we may lose accuracy of region information by the conversion to a MAY region in this example, this approximate information is no less accurate than can be represented by the triplet notation `X(1:I:1,1:N:1)` which can be converted to an approximate region `X(1:N:1,1:N:1)` [69]. We have found that this approximate information can still be useful in many flow-insentive analyses, including communication analysis for data block prefetching and poststoring.

2.3 Characteristics of Access Regions

In general, it is very difficult to obtain the exact solution for the problem of the operations between arbitrarily-shaped access regions. The algorithm to deal with this problem would be too complicated or sometimes even intractable. For instance, consider the intersection of two access regions.

Theorem 2.3 *The problem of intersection between m arbitary access regions is NP-hard.*

PROOF: Let C_i, $1 \leq i \leq m$ and D be n-tuples of integers; that is, $C_i = (c_{i1}, c_{i2}, \cdots, c_{in})$ and $D = (d_1, d_2, \cdots, d_n)$. Suppose that we are provided with an integer programming problem to find an n-tuple of integers $(x_1, x_2, \cdots, x_n)$ such that a system of m^2 equalities

$$(c_{j1} - c_{i1})x_1 + (c_{j2} - c_{i2})x_2 + \cdots + (c_{jn} - c_{in})x_n = 0$$

holds subject to the constraints $0 \leq x_k \leq d_k$, $1 \leq k \leq n$ and $1 \leq i, j \leq m$. By the definition of the Access Region, we can transform this integer programming problem into a problem of intersection between m access regions R_i, $1 \leq i \leq m$, represented by m access descriptors $\mathcal{A}_{c_{i1}\, d_1, c_{i2} d_2, \cdots, c_{in} d_n}^{c_{i1},\ c_{i2}, \cdots,\ c_{in}}$, respectively. It follows that the integer programming problem has a solution if and only if the access regions intersect. □

Theorem 2.3 proves that the general problem of intersection between arbitrary access regions is no less difficult than the integer programming problem, the well-known NP-complete problem. However, our access analysis on a real program revealed that most regions of interest are usually simple and regular and have similar shapes. Above all, the numbers of strides and spans to represent these access regions are small in most cases.

These observations lead to the conclusion that we do not need such complex algorithms for many problems encountered in the real world. Instead, we have developed various polynomial *region operation* algorithms whose main focus is simple or similar input regions.

The main issue here then is what criteria we can use to determine what input regions are "simple" or "similar". These criteria, called the *input conditions*, must be determined based on common access patterns in real programs. If conditions for simple regions are too general, they may increase the complexity of the operation algorithms for these regions. If conditions are too strict, they may make the algorithms useless for many important cases we must handle.

There are, of course, many cases containing access patterns that are more complex than those that can be handled by polynomial time algorithms. For these complicated cases, we can simplify the regions themselves to satisfy the given input conditions by converting to MAY regions, at the cost of accuracy of results, to apply polynomial time algorithms.

If a loss of accuracy is not allowed, then we could convert the problem of operations on access regions to an equivalent integer programming problem which different techniques based on Fourier-Motzkin variable elimination [60, 67] can handle since, as suggested in Theorem 2.3, both the problems can be interchangeably transformed. Although these integer programming techniques use worst-case exponential time algorithms, these also have proven to be efficient and successful in handling integer programming problems in real cases.

In Figure 2.11, we summarize the whole procedure discussed above for processing the input regions depending on the input conditions, given Fourier-Motzkin based algorithms and polynomial time region operation algorithms.

The notion of the similarity and simplicity of regions, which is determined upon given input conditions, is embodied in several terms, such as *conjunctive regions*, *complementary regions*

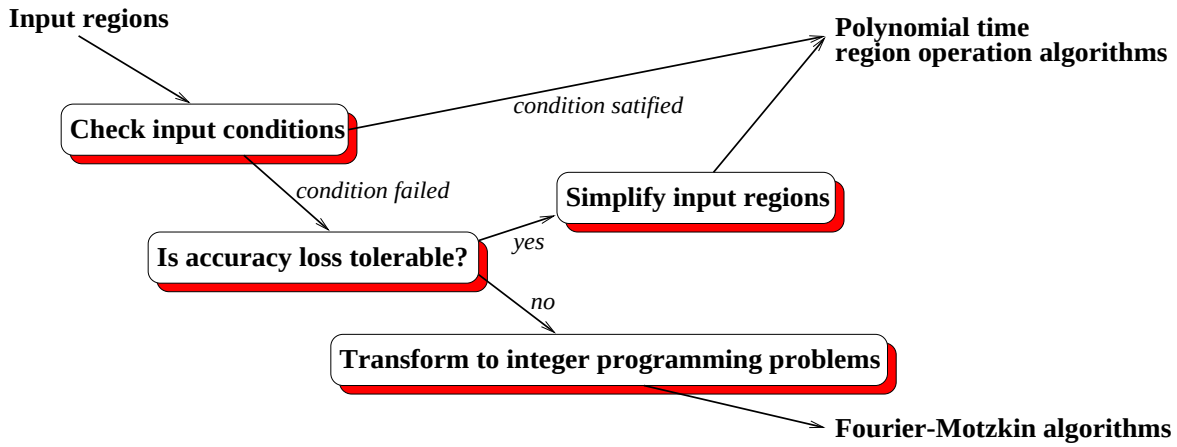

Figure 2.11: Procedure for the operations on the Access Region

and *subregions*. In this section, we discuss these various characteristics of regions for qualifying the similarity of input regions. The characteristics will be used in the basic operations on Access Regions to determine the input conditions for a given operation.

2.3.1 Conjunctive Regions

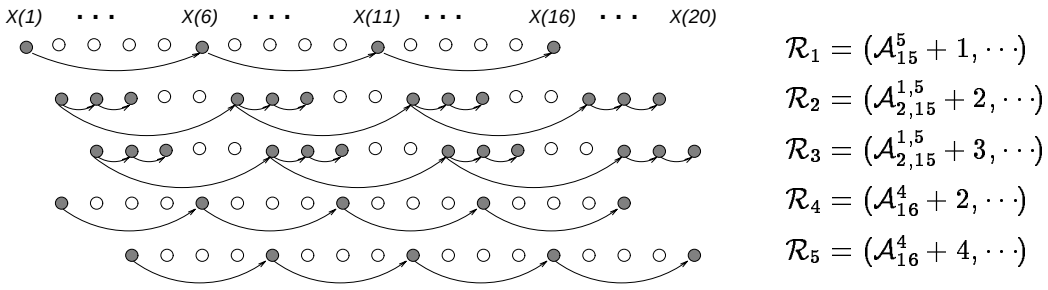

Figure 2.12: Access descriptors for the accesses in Figure 2.4

In Figure 2.12, we can clearly find the similarity between $\mathcal{R}_2$ and $\mathcal{R}_3$. In fact, they are identical except for their lower bounds differing by 1. In the same figure, the similarity between $\mathcal{R}_4$ and $\mathcal{R}_5$ can be noticed as well. These similarities are more or less immediately noticeable because the regions have exactly the same stride/span pairs. However, we also can see that there is some

similarity between $\mathcal{R}_1$ and $\mathcal{R}_2$ (or $\mathcal{R}_3$), although they have a different number of stride/span pairs.

From these examples, we can learn that the stride is one important component which determines the similarity between two regions in the Access Region notation.

Definition 2.3 *If two regions can be represented by the access descriptors with the same strides (not necessarily spans), then they are defined to be* **compatible**.

Theorem 2.4 *The relation 'compatible' is commutative and associative.*

PROOF: Suppose two regions $\mathcal{R}$ and $\mathcal{R}'$ are compatible. Then, by definition, the descriptors of $\mathcal{R}$ and $\mathcal{R}'$ have the same strides; that is, the number of stride/span pairs of the regions are the same and there is a unique matching stride σ in $\mathcal{R}$ for every stride σ' in $\mathcal{R}'$ such that $\sigma = \sigma'$, and vice versa. Since the operation '=' is commutative, it is trivial to show that $\mathcal{R}'$ and $\mathcal{R}$ are also compatible. Now, let $\mathcal{R}'$ and $\mathcal{R}''$ be compatible. Then, the access descriptors of $\mathcal{R}'$ and $\mathcal{R}''$ also have the same strides. Similarly, since the operation '=' is associative, it is clear that the access descriptors of $\mathcal{R}$ and $\mathcal{R}''$ should have the same strides. It follows that $\mathcal{R}$ and $\mathcal{R}''$ are compatible. □

Based on Definition 2.3, we can calculate two groups of compatible regions from Figure 2.12:

$$(\mathcal{R}_1, \mathcal{R}_2, \mathcal{R}_3) \text{ and } (\mathcal{R}_4, \mathcal{R}_5),$$

where compatible regions are put in the same parentheses. It is straightforward to show that $\mathcal{R}_4$ and $\mathcal{R}_5$ are compatible, and that $\mathcal{R}_2$ and $\mathcal{R}_5$ are compatible. However, to show that $\mathcal{R}_1$ and $\mathcal{R}_2$ (or $\mathcal{R}_3$) are compatible, we first have to use Theorem 2.1 to compute a double-stride access descriptor $\mathcal{A}^{5,\sigma^*}_{15,0} + 1$ from the original single-stride descriptor $\mathcal{A}^5_{15} + 1$, both of which equivalently represent $\mathcal{R}_1$ where σ^* can be any number. This enables us to prove that $\mathcal{R}_1$ and

$\mathcal{R}_2$ are compatible because the two strides of the two regions are now identical; that is, $5 = 5$ and $\sigma^* = 1$.

Definition 2.4 *Let a region $\mathcal{R}$ be represented by the access descriptor $\mathcal{A}^{\sigma_1,\sigma_2,\cdots,\sigma_m}_{\delta_1,\delta_2,\cdots,\delta_m} + l$, and $\mathcal{R}'$ be $\mathcal{A}^{\sigma'_1,\sigma'_2,\cdots,\sigma'_m}_{\delta'_1,\delta'_2,\cdots,\delta'_m} + l'$ such that $l \leq l'$. Let d be $l' - l$, the distance between $\mathcal{R}$ and $\mathcal{R}'$. If there exist stride/span pairs (σ_i, δ_i) in $\mathcal{R}$ and (σ'_j, δ'_i) in $\mathcal{R}'$, satisfying $\sigma_i = \sigma'_j$, $\sigma_i \backslash d$[†] and $d \leq \sigma_i + \delta_i$, then they are* **conjunctive**, *as long as all other stride/span pairs match.*

Intuitively, we can see that conjunctive regions are compatible while the converse is not necessarily true. For instance, consider two regions $\mathcal{R}_1$ and $\mathcal{R}_2$. To show they are conjuntive, we represent $\mathcal{R}_2$ with the access descriptor $\mathcal{A}^{5,\sigma^*}_{15,0} + 1$, as described above, because the regions must have the same numbers of stride/span pairs in order to prove the conjunctiveness using Definition 2.4. We choose $(\sigma^*, 0)$ and $(1, 2)$ each for comparison because the other pairs of stride/span, $(5, 15)$, already match. Since σ^* can be any number by definition, we set $\sigma^* = 1$ to match the counterpart stride of $(1, 2)$. Now, since the distance d between the two regions is 1 $(= 2 - 1)$, we can show that $\sigma^* \backslash d$ and $d \leq \sigma^* + 0$, thereby proving that $\mathcal{R}_1$ and $\mathcal{R}_2$ are conjunctive. As described earlier, it is easy to see that these conjunctive regions are compatible as well.

Theorem 2.5 *Let $\mathcal{R}$ and $\mathcal{R}'$ be two conjuntive regions, as defined in Definition 2.4. The union of the two regions can be represented by an access descriptor $\mathcal{A}^{\sigma'_1,\cdots,\sigma'_j,\cdots,\sigma'_m}_{\delta'_1,\cdots,\delta'_j+d,\cdots,\delta'_m} + l$, which has the lower bound of region $\mathcal{R}$ and the same stride/span pairs as region $\mathcal{R}'$ except a single pair $(\sigma'_j, \delta'_j + d)$ substituted for the original pair (σ'_j, δ'_j) in $\mathcal{R}'$.*

[†] $\sigma_i \backslash d$ denotes that σ_i divides d.

PROOF: Since $\mathcal{R}$ and $\mathcal{R}'$ are conjunctive, all other stride/span pairs match and the remaining pairs, (σ_i, δ_i) and (σ'_j, δ'_j), satisfy the conditions in Definition 2.4, this proves that the regions are compatible, but shifted by d. Thus, the union of the regions must have the same stride/span pairs except for one pair whose span must encompass the total range stretched by both (σ_i, δ_i) and (σ'_j, δ'_j). The total range computes $\delta_i + \delta'_j - (\delta_i - d)$ where $\delta_i - d$ is the length of the overlapping area by the two pairs, which results in $\delta'_j + d$ by simplication. □

Using the notion of the conjuntive region, Theorem 2.5 provides a theoretical ground upon which we identify the similarity of different regions and aggregate them into a single region. It can be noticed that the aggregated region generated by Theorem 2.5 is still compatible with the original regions. For example, $\mathcal{R}_1$ and $\mathcal{R}_2$ can be aggregated to a region which can be represented by a new access descriptor $\mathcal{A}^{1,5}_{3,15} + 1$, as shown in Figure 2.13. Clearly, this new region $\mathcal{R}_1 \cup \mathcal{R}_2$ is compatible with $\mathcal{R}_1$ and $\mathcal{R}_3$.

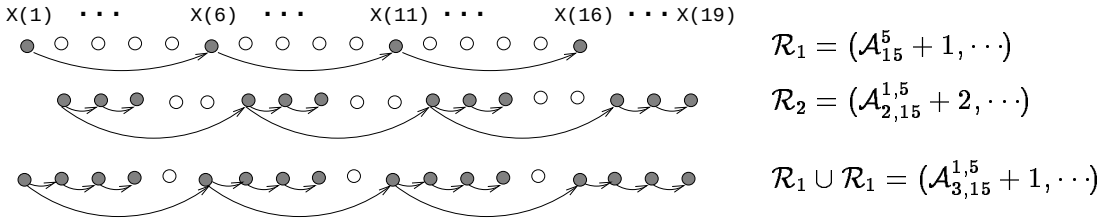

Figure 2.13: Aggregation of two regions $\mathcal{R}_1$ and $\mathcal{R}_2$

The aggregated region $\mathcal{R}_1 \cup \mathcal{R}_2$ and $\mathcal{R}_3$ in Figure 2.12 again can be shown to be conjuntive by Definition 2.4. Thus, using Theorem 2.5, we can obtain a larger region $\mathcal{R}_1 \cup \mathcal{R}_2 \cup \mathcal{R}_3$ which can be represented by an access descriptor $\mathcal{A}^{1,5}_{4,15} + 1$. This aggregated region is compatible with its subregions, that is, $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$, $\mathcal{R}_1 \cup \mathcal{R}_2$ and $\mathcal{R}_2 \cup \mathcal{R}_3$.

The regions from Figure 2.12 can be classified into three groups where conjunctive regions are put in the same group:

$(\mathcal{R}_1, \mathcal{R}_2)$, $(\mathcal{R}_2, \mathcal{R}_3)$ and $(\mathcal{R}_4, \mathcal{R}_5)$.

Notice that $(\mathcal{R}_1, \mathcal{R}_2)$ and $(\mathcal{R}_2, \mathcal{R}_3)$ do not imply $(\mathcal{R}_1, \mathcal{R}_3)$. This is because, unlike compatible regions, the relation '*conjunctive*' is not associative. Therefore, $\mathcal{R}_1 \cup \mathcal{R}_3$ is not compatible with its subregions $\mathcal{R}_1$ or $\mathcal{R}_3$ because the union of non-conjunctive regions is not necessarily compatible with the original regions.

2.3.2 Complementary Regions

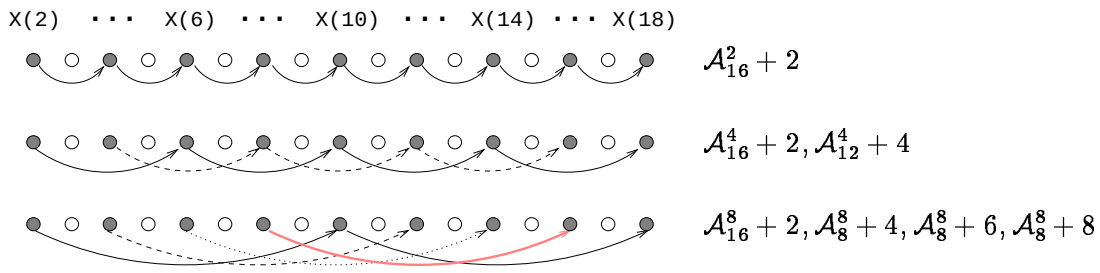

Figure 2.14: Decompositions of access region $\bar{\mathcal{R}} = (\mathcal{A}^2_{16} + 2, \cdots)$

In Figure 2.12, we can see that the access pattern of region $\bar{\mathcal{R}}$, originally represented by an access descriptor $\mathcal{A}^2_{16} + 2$, is actually identical to a union of two regions with the descriptors $\mathcal{A}^4_{16} + 2$ and $\mathcal{A}^4_{12} + 4$, which are, repectively, denoted as a solid line and a dashed line in the middle of the figure. Similarly, $\bar{\mathcal{R}}$ can be represented by a union of four regions with forms $\mathcal{A}^8_{16} + 2$, $\mathcal{A}^8_8 + 4$, $\mathcal{A}^8_8 + 6$, and $\mathcal{A}^8_8 + 8$, as shown at the bottom of the figure.

This example shows that region $\bar{\mathcal{R}}$ can be decomposed into n subregions such that a single stride/span pair of each subregion is $(2n, \delta)$ where δ is some integer less than or equal to the span 16 of $\bar{\mathcal{R}}$, while their other stride/span pair is the same as that of $\bar{\mathcal{R}}$. We call these n subregions of $\bar{\mathcal{R}}$ *n-complementary* regions of $\bar{\mathcal{R}}$. The regions, with forms $\mathcal{A}^4_{16} + 2$ and $\mathcal{A}^4_{12} + 4$, are 2-complementary regions of $\bar{\mathcal{R}}$. Likewise, the regions, with forms $\mathcal{A}^8_{16} + 2$, $\mathcal{A}^8_8 + 4$, $\mathcal{A}^8_8 + 6$, and $\mathcal{A}^8_8 + 8$, are 4-complementary regions of $\bar{\mathcal{R}}$.

Theorem 2.6 *Given an arbitrary region $\mathcal{R}$, n-complementary regions of $\mathcal{R}$ are compatible.*

PROOF : The proof is obvious from the definition of n-complementary regions. □

Theorem 2.7 *Let n regions $\mathcal{R}_i$, $1 \leq i \leq n$, be n-complementary regions of a region $\mathcal{R}$ represented by access descriptor $\mathcal{A}^{\sigma_1,\cdots,\sigma_k,\cdots,\sigma_m}_{\delta_1,\cdots,\delta_k,\cdots,\delta_m} + l$. Then, for an arbitrary k, $1 \leq k \leq m$, region $\mathcal{R}$ can be represented equivalently by a union of the n regions, each with access descriptor $\mathcal{A}^{\sigma_1,\cdots,\sigma'_k,\cdots,\sigma_m}_{\delta_1,\cdots,\delta'_k,\cdots,\delta_m} + l + (i-1)\sigma_k$ where $\sigma'_k = n\sigma_k$ and $\delta'_k = \lfloor \frac{\delta_k - i\sigma_k}{\sigma'_k} \rfloor \sigma'_k$.*

PROOF : As can be seen in the example from Figure 2.14, n-complemented regions are, in fact, the interleaved subregions of a region; thus, the strides of the n subregions must be of n times σ_k for a certain k. The lower bounds of each subregion are increased by σ_k at a time starting from l. The spans of the regions can be calculated from the starting address of each region and their new strides. □

Theorem 2.7 provides a glimpse of the concept of the aggregation operations on access regions with 'similar' structure; that is, if we have any n-complementary regions of access region $\mathcal{R}$, we can aggregate the n regions into one region $\mathcal{R}$. For example, in Figure 2.12, the two regions, with descriptors $\mathcal{A}^4_{16} + 2$ and $\mathcal{A}^4_{16} + 4$, can be aggregated to the region with descriptor $\mathcal{A}^2_{18} + 2$ since $\mathcal{A}^4_{16} + 2$ and $\mathcal{A}^4_{16} + 4$ form 2-complementary regions of the region with $\mathcal{A}^2_{18} + 2$. We will discuss further how to use the notion of complementary regions in the aggregation operation in Section 2.4.3.

In Section 2.2.3, we discussed how we use the coalescing operation to generate, given an access region as input, some isomorphic regions with fewer stride/span pairs by eliminating redundant indices involved in the original access pattern. In a similar but more general way, complementary regions also can be used to derive, from a given access region, its isomorphic

access regions having different stride/span pairs. For example, in Figure 2.5, 3-complementary regions of $\mathcal{R}_z$ are the regions with descriptors $\mathcal{A}_9^3 + 0$, $\mathcal{A}_9^3 + 1$, and $\mathcal{A}_9^3 + 2$ through which we can identify $\mathcal{R}_z$ as a union of the three regions with distance 1 between the regions. Let d be the distance and n be the number of complementary regions. If we want to combine the three access descriptors to generate a double-stride region represented by a descriptor $\mathcal{A}_{9,\delta}^{3,\sigma} + l$, then we first expand the descriptors of these regions to apply Theorem 2.7:

$$\mathcal{A}_{9,0}^{3,\sigma^*} + 0,\ \mathcal{A}_{9,0}^{3,\sigma^*} + 1, \text{ and } \mathcal{A}_{9,0}^{3,\sigma^*} + 2,$$

where σ^* can be any number. First, we can calculate $l = 0$ and $\sigma = 1$ since the equality

$$l + (i-1)\sigma = i - 1 \text{ for } 1 \leq i \leq 3$$

can be obtained by applying Theorem 2.7 to this example. Now, in order to calculate δ, we have to use the equation

$$\lfloor \frac{\delta - i}{\sigma^*} \rfloor \sigma^* = 0 \text{ for } 1 \leq i \leq 3,$$

where $\sigma^* = 3$ because $\sigma^* = 3 \cdot \sigma$ according to Theorem 2.7. Using this equation, we can identify that the span δ is 2. Thus, we obtain the descriptor $\mathcal{A}_{9,\mathbf{2}}^{3,\mathbf{1}} + \mathbf{0}$ as the result of these computations for this example. As a consequence, we can see that $\mathcal{R}$ in Figure 2.5 can be represented by either $\mathcal{A}_9^1 + 0$ or $\mathcal{A}_{9,2}^{3,1} + 0$.

The above example shows how we use complementary regions to expose the components of the original access region for the analysis of the access pattern of the region. This will help us to determine subregions of the region and to perform intersection and subtraction operations on access regions.

2.3.3 Subregions

In previous sections, we use the term *subregion* to mean a region whose elements accessed by the region are a subset of the elements accessed by another region. In other words, we say that $\mathcal{R}'$ is a subregion of $\mathcal{R}$ if the elements accessed by $\mathcal{R}'$ are a subset of the elements accessed by $\mathcal{R}$. For instance, all of the n-complementary regions of $\mathcal{R}$ are subregions of $\mathcal{R}$.

To formulate the general problem of subregions, suppose that a region $\mathcal{R}'$ has access descriptor $\mathcal{A}^{\sigma'_1,\cdots,\sigma'_m}_{\delta'_1,\cdots,\delta'_m} + l'$ and another region $\mathcal{R}$ has access descriptor $\mathcal{A}^{\sigma_1,\cdots,\sigma_n}_{\delta_1,\cdots,\delta_n} + l$. Then, the problem:

"Is $\mathcal{R}'$ a subregion of $\mathcal{R}$?"

can be rephrased equivalently as an integer programming problem

> *"Is there an n-tuple of integers, $(x_1, x_2, \cdots, x_n)$, which satisfies the equality $\sum_{i=1}^{n} \sigma_i x_i - \sum_{j=1}^{m} \sigma'_j y_j = l' - l$ for any permutation of integers, $(y_1, y_2, \cdots, y_m)$, subject to the constraints, $0 \le x_i \le \frac{\delta_i}{\sigma_i}$, $1 \le i \le n$, and $0 \le y_j \le \frac{\delta'_j}{\sigma'_j}$, $1 \le j \le m$?"*

For example, suppose that $\mathcal{R}$ is represented by $\mathcal{A}^{1,15}_{7,60} + 1$ and $\mathcal{R}'$ by $\mathcal{A}^{4,30,1}_{4,30,2} + 24$. In order to determine if $\mathcal{R}'$ is a subregion of $\mathcal{R}$, we construct the equation

$$x_1 + 15x_2 - 4y_1 - 30y_2 - y_3 = 23$$

with five constraints

$$0 \le x_1 \le 7,\ 0 \le x_2 \le 4,\ 0 \le y_1 \le 1,\ 0 \le y_2 \le 1 \text{ and } 0 \le y_3 \le 2.$$

In this case, we can prove that $\mathcal{R}'$ is not a subregion of $\mathcal{R}$ by showing that there is no solution (x_1, x_2), subject to the constraints, satisfying the equation when $y_1 = y_2 = y_3 = 0$. If $\mathcal{R}''$ has the same access descriptor as $\mathcal{R}'$ but with a lower bound of 32, that is, $\mathcal{A}^{4,30,1}_{4,30,2} + 32$, then we have a different equation

$$x_1 + 15x_2 - 4y_1 - 30y_2 - y_3 = 31,$$

subject to the same constraints given above. From this equation, we can see that $\mathcal{R}''$ would become a subregion of $\mathcal{R}$ since the equation now holds on any values of (y_1, y_2, y_3).

Similar to other operations on access regions, such as intersection and subtraction, it is NP-hard to find a solution for this general access region problem for subregions. However, the properties of regularity and similarity of usual access patterns in real programs, supported by the discussion about similar access patterns in Section 2.2.2, lead us to solve this problem in the form of the integer programming problem.

From the discussion with the example from Figure 2.4, we argued that the similarity of access patterns depends on the similarity of strides and spans involved in those accesses. The similarity of strides and spans does not necessarily mean that the strides and spans must be exactly the same, but that they must have functional relationships between them, such as one being a multiple of the other. Back to the example from Figure 2.12, if we have an access region with descriptor $\mathcal{A}^2_{20} + 0$, then we can see that many elements accessed by the region are 'regularly' overlapping with those from the region with descriptor $\mathcal{A}^4_{20} + 2$ in Figure 2.12, but not with the regions with descriptor $\mathcal{A}^5_{15} + 1$. This is, in fact, mainly because stride 4 is a multiple of stride 2, although two regions have different strides. From this example, we may say,

> "*Two regions are similar if each stride of one region can be represented as a multiple of the other region's stride*".

As another example for this case, let's consider the two regions $\mathcal{R}$ and $\mathcal{R}'$ above again. In the example, we see that strides 1, 4, and 30 of $\mathcal{R}'$ can be represented by multiples of stride 1 and 15 of $\mathcal{R}$. After considering their spans, we may couple the strides with each other to rebuild the equation as

$$(x_1 - 4y_1 - y_3) + 15(x_2 - 2y_2) = 23.$$

In this new equation, we only have two terms, each of which corresponds to each index for $\mathcal{R}$, on the left hand side, thus simplifying the problem. Here, we attempt to prove that

$$\mathcal{R}' \text{ is not a subregion of } \mathcal{R}$$

by calculating the possible boundaries of values of both terms, which are

$$-6 \leq x_1 - 4y_1 - y_3 \leq 7 \text{ and } -2 \leq x_2 - 2y_2 \leq 4.$$

If we move the first term on the left-hand side to the right-hand side, we get

$$15(x_2 - 2y_2) = 23 - (x_1 - 4y_1 - y_3)$$

from which we learn that the right-hand side must be a multiple of 15 in order to match the left-hand side. To test this, we produce boundary conditions for the right-hand side

$$16 \leq 23 - (x_1 - 4y_1 - y_3) \leq 29$$

It is clear that the right-hand side cannot be equal to the left-hand side because the term $23 - (x_1 - 4y_1 - y_3)$ on the right-hand side cannot be a multiple of 15, thereby proving $\mathcal{R}'$ is not a subregion of $\mathcal{R}$.

Now, let's compare another region, $\mathcal{R}''$, which was given above, with $\mathcal{R}$. Since $\mathcal{R}''$ has the same stride/span pairs as $\mathcal{R}'$, their strides also can be represented by multiples of strides $\mathcal{R}$, which results in an equation

$$(x_1 - 4y_1 - y_3) + 15(x_2 - 2y_2) = 31.$$

This equation is again converted to

$$(x_1 - 4y_1 - y_3 - 1) + 15(x_2 - 2y_2 - 2) = 0$$

where the constant term 31 is split into 30 (a multiple of 15) and 1 (the remainder), and absorbed into the two terms on the left-hand side. Now, instead of attempting to solve the whole equation, we divide the original problem into two subproblems:

$$x_1 - 4y_1 - y_3 = 1, \text{ and } x_2 - 2y_2 = 2$$

We can see that if x_1 and x_2 satisfy both the equations, then

$$\mathcal{R}'' \text{ is a subregion of } \mathcal{R},$$

even though the converse is not necessarily true. Based on this observation, let's first look at the equation $x_1 - 4y_1 - y_3 = 1$. First, we convert it to $x_1 = 1 + 4y_1 + y_3$ to show that x_1 satifies this equation for whatever values (y_2, y_3) have subject to the given contraints. Then, we calculate the boundaries of the terms x_1 and $1 + 4y_1 + y_3$, which are

$$0 \leq x_1 \leq 7 \text{ and } 1 \leq 1 + 4y_1 + y_3 \leq 7.$$

From this, we can show that $1 + 4y_1 + y_3$ is a subrange of x_1. Now, consider the next equation $x_2 - 2y_2 = 2$. In a similar way, we calculate the boundaries of the terms x_2 and $2 + 2y_2$, and show that

$$2 \leq 2 + 2y_2 \leq 4 \text{ is a subrange of } 0 \leq x_2 \leq 4.$$

Consequently, we have proven that $\mathcal{R}''$ is a subregion of $\mathcal{R}$ by showing that x_1 and x_2 satisfy both the equations for any permutations of integers, (y_1, y_2, y_3).

As can be seen in these two examples, we may simplify the original problem in real programs, taking advantage of the property of similarity of access patterns. In addition, we have found in the study of real programs that loops which are more deeply-nested than three or four are uncommon. Although the numbers of stride/span pairs do not always depend directly on the nesting of loops, this fact helps keep the numbers relatively small in access region problems, which convinces us that most problems of our interest will have only low order polynomial time complexities.

2.4 Basic Operations on Access Regions

Many components of a parallelizing compiler rely on the analysis of array access regions. Dependence analysis checks whether the access patterns of arrays overlap. Array privatization, one of the most important dependence-elimination techniques, is based on region *intersection* and *subtraction* operations. A precise *aggregation* operation on access regions is important for generating the data communication needed for distributed memory machines.

The complete implementation of these operations has not been done as yet. For the experiments reported in this dissertation, we developed a few primitive algorithms for these operations. These algorithms are not general enough to handle all the cases occurring in real programs. The full versions of the algorithms are still under development.

One important prerequisite study we should conduct to develop the full version of algorithms is thorough analyses of access patterns in a broad range of scientific applications. For this purpose, the study on the access patterns of real programs based on benchmarks from the Perfect and SPEC95fp benchmark suites would be required. Based on this study, the complete algorithms to be used in the procedure for the operations on Access Regions shown in Figure 2.11 can be implemented. Those complete algorithms will define the input conditions and region operation algorithms for each target operation.

For now, therefore, we will put aside the full description of these algorithms and, throughout this section, we will discuss only the basic issues of input conditions and algorithms for three basic operations (intersection, subtraction, aggregation) which were used for the analysis of array access regions in our experiments.

2.4.1 Intersection

The intersection algorithm for two access regions can return either the exact region(s) of intersection, or a more simple YES/NO-style answer by checking whether the exact result is empty. Theorem 2.3 shows that even generating the simple YES/NO answer has worst-case exponential time complexity.

However, we found that, especially for simple and similar regions, we can solve the problem with worst-case polynomial time algorithms. For instance, consider two regions, represented by triplet notation: `V(1:8:1)` and `V(5:15:1)`. Intersecting these two regions does not require expensive general algorithms; instead, only traditional interval algorithms are requred to produce the result, `V(8:15:1)`.

Similarly, for two access regions represented by access descriptors $\mathcal{A}^4_{16} + 2$ and $\mathcal{A}^4_{16} + 4$ in Figure 2.12, we get an empty region as the result by simply checking the strides and spans. For these kinds of simple and similar regions, the polynomial time region operation algorithms discussed in Section 2.3 have been developed for intersection. We call these algorithms *region intersection* algorithms.

The input conditions for intersection, as mentioned in the beginning of Section 2.4, are not yet fully defined. However, one input condition we currently have for the intersection operation is

"Are all the input regions single-stride?"

where a single-stride region is one whose access descriptor has only a single stride. To describe the region intersection algorithm for single-stride regions, suppose we are given two input access regions $\mathcal{R}_1$ and $\mathcal{R}_2$ for array `Y` which are represented by descriptors $\mathcal{A}^2_{18}(+0)$ and $\mathcal{A}^3_{12} + 3$, respectively. The regions are shown in in Figure 2.15.

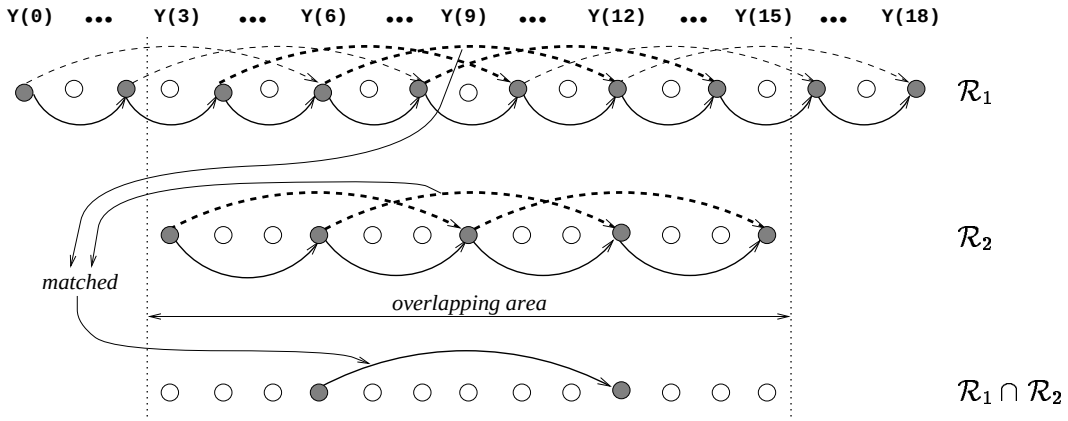

Figure 2.15: $\mathcal{R}_1 \cap \mathcal{R}_2 = (\mathcal{A}^2_{18}, \cdots) \cap (\mathcal{A}^3_{12} + 3, \cdots) \implies (\mathcal{A}^6_6 + 6, \cdots)$

To perform intersection of the input regions $(\mathcal{A}^2_{18} \cap (\mathcal{A}^3_{12} + 3))$, the region intersection algorithm first determines the area of $\mathcal{R}_1$ which overlaps $\mathcal{R}_2$, called the *overlapping area*. If $\mathcal{R}_1$ and $\mathcal{R}_2$ have lower bounds l_1 and l_2, then we can compute the upper bounds u_1 and u_2 of the regions by adding spans to the lower bounds. By comparing the lower and upper bounds, we can determine the overlapping area. In this example, the overlapping area ranges from `Y(3)` to `Y(15)` in the array `Y`.

Next, the region intersection algorithm finds a stride σ_{lcm} which is the least common multiple (LCM) of the strides of both $\mathcal{R}_1$ and $\mathcal{R}_2$. Then, it forms a set of complementary regions with that stride for each original region. In this example, σ_{lcm} is 6 and, thus, the complementary regions with stride 6 are 3-complementary regions of $\mathcal{R}_1$, represented by three access descriptors $\mathcal{A}^6_{18}$, $\mathcal{A}^6_{12} + 2$, and $\mathcal{A}^6_{12} + 4$, repectively. Similarly, 2-complementary regions of $\mathcal{R}_2$ can be represented by two descriptors which are $\mathcal{A}^6_{12} + 3$ and $\mathcal{A}^6_6 + 6$. These complementary regions are marked with dashed lines in Figure 2.15.

Then, it must determine which of the complementary regions in those sets access the same elements within their overlapping area `Y(3:15)`. This operation requires finding *synchronized* regions of the complementary regions whose elements belong solely to the area.

Definition 2.5 *If two regions $\mathcal{R}$ and $\mathcal{R}'$ have the same n strides, say $\sigma_i, 1 \leq i \leq n$, and $LCM(\sigma_1, \sigma_2, \cdots, \sigma_n)$ divides $|l_1 - l_2|$, the distance between the regions, then they are defined to be* **synchronized**.

For example, the general form of all synchronized regions of $\mathcal{A}^6_{12}+2$ is $\mathcal{A}^6_{\delta*}+2+6d$ where d and δ* can be any integer. Among these, there is only one synchronized region which accesses the overlapping area in Figure 2.15: $\mathcal{A}^6_6+8$. Likewise, we can compute two other synchronized regions of the original complementary regions of $\mathcal{R}_1$: $\mathcal{A}^6_6+4$, $\mathcal{A}^6_6+6$. We make a set $\mathcal{S}_1$ composed of these three resulting regions. Similarly, we compute the synchronized regions of the original complementary regions of $\mathcal{R}_2$, $\mathcal{A}^6_{12}+3$,$\mathcal{A}^6_6+6$, and make a set $\mathcal{S}_2$ composed of these regions. The regions in $\mathcal{S}_1$ and $\mathcal{S}_2$ are highlighted among the dashed lines in Figure 2.15. The result of intersection of $\mathcal{R}_1$ and $\mathcal{R}_2$ is

$$\mathcal{S}_1 \cap \mathcal{S}_2 = \{\mathcal{A}^6_6+8, \mathcal{A}^6_6+4, \mathcal{A}^6_6+6\} \cap \{\mathcal{A}^6_{12}+3, \mathcal{A}^6_6+6\} = \{\mathcal{A}^6_6+6\}.$$

We can generalize in a very natural way to the region intersection algorithms for multi-stride regions by providing several other input conditions, such as:

"Do all the stride/span pairs of input regions match with the exception of at most one pair?".

One example of the input regions satisfying the above condition would be $\mathcal{A}^{2,100}_{18,800}$ and $\mathcal{A}^{3,100}_{12,800}$+3. The stride/span pair (100,800) from each region matches, but the other, (2,18) and (3,12), do not. In this example, we extend the ideas used to process the single-stride case, such as complementary regions and synchronized regions. In order to calculate the intersection of the two regions, we reduce by ignoring the matched pairs of the multi-stride problem to a single-

stride problem, which is the intersection of $\mathcal{A}^2_{18}$ and $\mathcal{A}^3_{12}+3$. Now, this problem can be handled in the same way described above in Figure 2.15, which produces $\{\mathcal{A}^6_6+6\}$. We expand this result to $\{\mathcal{A}^{6,100}_{6,800}+6\}$ for the final result.

In the full version of the intersection operation on Access Regions, more input conditions will be needed. The corresponding region intersection algorithms will be defined in a way similar to that mentioned in this section.

2.4.2 Subtraction

The *region subtraction* algorithm is a slightly modified version of the region intersection algorithm. Like the intersection operation, the general problem of the subtraction operation on access regions requres worst-case exponential time algorithms to solve. However, based on our access analysis on real programs, most access regions for those programs do not need such expensive algorithms. Therefore, we established various input conditions for subtraction which are almost the same as for intersection. One of these conditions is

"Are the input regions all single-stride?"

Suppose two regions $\mathcal{R}_1$ and $\mathcal{R}_2$ are single-stride. When we compute the subtraction operation $\mathcal{R}_1 - \mathcal{R}_2$, similar to intersection in Section 2.4.1, we first identify the overlapping and non-overlapping areas between regions $\mathcal{R}_1$ and $\mathcal{R}_2$ using their lower bounds and upper bounds. The elements of $\mathcal{R}_1$ in the non-overlapping area must be part of the result of intersection. If there is an area of overlap, then we must calculate the elements which are accessed by both regions, and remove them from $\mathcal{R}_1$. The remaining elements in $\mathcal{R}_1$ must be part of the result.

Just as in the intersection algorithm, the region subtraction algorithm converts any two single-stride regions to sets of regions with the same stride by finding their complementary

regions with the appropriate stride. Then, it finds a stride σ_{lcm} which is the least common multiple (LCM) of the strides of both $\mathcal{R}_1$ and $\mathcal{R}_2$, and forms a set of complementary regions with that stride for each original region. Next, it must find which of the complementary regions in those sets access the same elements within the overlapping area. This is done by finding a pair of complementary regions, one from each set, which are synchronized.

The final result consists of all those complementary regions of $\mathcal{R}_1$ which were not synchronized with any complementary regions of $\mathcal{R}_2$, plus the parts of $\mathcal{R}_1$ from the non-overlapping areas, which were calculated earlier.

For example, let's consider again the two regions from Figure 2.15. This time, the task is to subtract $\mathcal{R}_2$ from $\mathcal{R}_1$, as displayed in Figure 2.16.

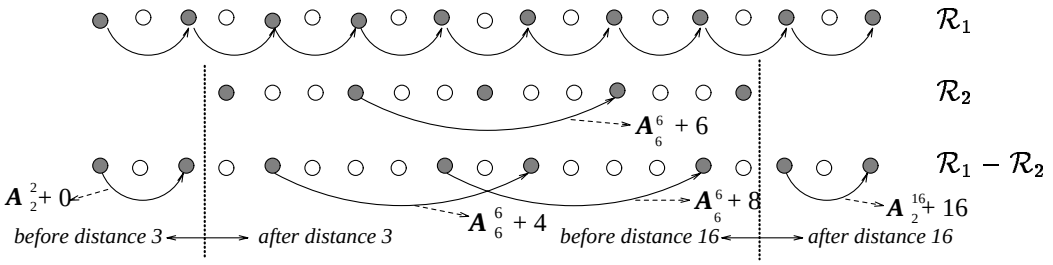

Figure 2.16: $\mathcal{R}_1 - \mathcal{R}_2 \implies (\mathcal{A}_2^2, \cdots), (\mathcal{A}_6^6 + 4, \cdots), (\mathcal{A}_6^6 + 8, \cdots), (\mathcal{A}_2^2 + 16, \cdots)$

First we find the non-overlapping area in $\mathcal{R}_1$, which is

$$\mathcal{S}_{non-overlap} = \{\mathcal{A}^2_{\lfloor \frac{(3-1)}{2} \rfloor \cdot 2}, \mathcal{A}^2_{\lfloor \frac{(18-(12+3))}{2} \rfloor \cdot 2} + 12 + 3 + 1\}$$

where the new spans are calculated in a way similar to that shown in Theorem 2.7. Next, we find the $LCM(2,3)$, which is 6 in this example. Then, we compute the 3-complementary regions of $\mathcal{R}_1$, which are

$$\mathcal{S}_1 = \{\mathcal{A}^6_{\delta*}, \mathcal{A}^6_{\delta*} + 2, \mathcal{A}^6_{\delta*} + 4\}$$

where δ^* is a span not yet computed. Similarly, we find the 2-complementary regions of $\mathcal{R}_2$, which are

$$\mathcal{S}_2 = \{\mathcal{A}^6_{\delta*} + 3, \mathcal{A}^6_{\delta*} + 6\}.$$

Comparing $\mathcal{S}_1$ and $\mathcal{S}_2$, and identifying that $\mathcal{A}^6_{\delta*}$ from $\mathcal{R}_1$ and $\mathcal{A}^6_{\delta*}+6$ from $\mathcal{R}_2$ are synchronized, we remove $\mathcal{A}^6_{\delta*}$ from $\mathcal{S}_1$ to produce

$$\mathcal{S}_{overlap} = \mathcal{S}_1 - \{\mathcal{A}^6_{\delta*}\} = \{\mathcal{A}^6_{\delta*} + 2, \mathcal{A}^6_{\delta*} + 4\}.$$

The region $\mathcal{A}^6_{\delta*} + 2$ has a lower bound less than 3, placing it outside the overlapped area; therefore, we recompute its lower bound to place it inside the overlapped area, with the result $\mathcal{A}^6_{\delta*} + 8$. Now, we compute the new spans for the regions in $\mathcal{S}_{overlap}$:

$$\mathcal{A}^6_{\lfloor\frac{12+3-4}{6}\rfloor\cdot 6} + 8 = \mathcal{A}^6_6 + 8 \quad \text{and} \quad \mathcal{A}^6_{\lfloor\frac{12+3-8}{6}\rfloor\cdot 6} + 4 = \mathcal{A}^6_6 + 4.$$

They are included to make the complete result

$$\mathcal{S}_{result} = \mathcal{S}_{non-overlap} \cup \mathcal{S}_{overlap} = \{\mathcal{A}^6_6 + 8, \mathcal{A}^6_6 + 4, \mathcal{A}^2_2, \mathcal{A}^2_2 + 16\}.$$

Similar to the cases with intersection of multiple-stride regions, we can handle subtraction of multiple-stride regions by enforcing some input conditions such as those discussed in Section 2.4.1. To illustrate the multi-stride case, consider the following loop from the ARC2D benchmark in Figure 2.17.

```
real X(P,Q,R)
...
do J = 1, N, 1
   do K = 1, M, 1
      X(J,K,2) = ···
      X(J,K,1) = ···
   enddo
   ··· X(J,M,1) ···
   ··· X(J,M,2) ···
enddo
```

Figure 2.17: Simplified code example in ARC2D from the Perfect benchmarks

Let $\mathcal{R}_{inner}$ be the access region made by the references `X(J,K,1)` and `X(J,K,2)`, and $\mathcal{R}_{outer}$ be the one made by the references `X(J,M,1)` and `X(J,M,2)`. In order to calculate $\mathcal{R}_{inner} - \mathcal{R}_{outer}$, the subscripting expressions are first linearized and aggregated, to produce the regions

$$\mathcal{R}_{inner} = (\mathcal{A}^{1,P,PQ}_{N-1,(M-1)P,PQ} + 1, \cdots) \text{ and } \mathcal{R}_{outer} = (\mathcal{A}^{1,P,PQ}_{N-1,0,PQ} + (M-1)P + 1, \cdots).$$

Notice that, in this example, the stride/span pairs are the same between the two regions except for the middle one in each region. By ignoring the other pairs, therefore, we can reduce the multi-stride problem to a single-stride problem, which is $\mathcal{R}'_{inner} - \mathcal{R}'_{outer}$ where

$$\mathcal{R}'_{inner} = (\mathcal{A}^{P}_{(M-1)P} + 1, \cdots) \text{ and } \mathcal{R}'_{outer} = (\mathcal{A}^{P}_{0} + (M-1)P + 1, \cdots).$$

The region subtraction algorithm calculates this reduced problem to generate the result

$$\mathcal{S}'_{result} = \mathcal{S}'_{non-overlap} \cup \mathcal{S}'_{overlap} = \{\mathcal{A}^{P}_{(M-2)P} + 1\}.$$

By expanding $\mathcal{S}'_{result}$ for the original multi-stride problem, we have the final result

$$\mathcal{S}_{result} = \{\mathcal{A}^{1,P,PQ}_{N-1,(M-2)P,PQ} + 1\}.$$

2.4.3 Aggregation

The task of aggregation is to identify the access patterns of input regions and to combine regions with *similar* structure. In Section 2.3, we discussed how conjunctive regions and complementary regions are combined. The first step of the *region aggregation* algorithm is to partition the input regions into groups based on the similarity of their strides and spans. Here, we define the similar regions as conjunctive regions or complementary regions. We cluster all conjunctive or complementary regions into the same groups.

After combining the regions that belong to the same group into an aggregated region, the region aggregation algorithm identifies subregions among these aggregated regions, and absorbs the subregions into their super-regions. The final result is a set of the regions remaining after this step.

To illustrate the aggregation operation, consider the subroutine `foo` in Figure 2.12. The access patterns for `X` within `foo` and their corresponding abstract forms are shown in the figure.

Given these input regions, the region aggregation algorithm uses the following procedure to produce a single aggregated region, represented by descriptor $\mathcal{A}^1_{19} + 1$, for the summarized MUST READ region for `X` in `foo`:

1. Following Theorem 2.1, $\mathcal{A}^5_{15} + 1$ is expanded to $\mathcal{A}^{\sigma^*,5}_{0,15} + 1$ where σ^* can be any number.
2. σ^* in $\mathcal{A}^{\sigma^*,5}_{0,15} + 1$ is set to 1 to be compatible with others, $\mathcal{A}^{1,5}_{2,15} + 2$ and $\mathcal{A}^{1,5}_{2,15} + 3$.
3. Three conjunctive regions with descriptors $\mathcal{A}^{1,5}_{0,15} + 1$, $\mathcal{A}^{1,5}_{2,15} + 2$ and $\mathcal{A}^{1,5}_{2,15} + 3$ are now aggregated to a region with $\mathcal{A}^{1,5}_{4,15} + 1$, as discussed in Section 2.3.1.
4. $\mathcal{A}^{1,5}_{4,15} + 1$ is coalesced to $\mathcal{A}^1_{19} + 1$ by the function `coalesce_region`.
5. Two complementary regions with $\mathcal{A}^4_{16} + 2$ and $\mathcal{A}^4_{16} + 4$ are combined to the aggregated region with $\mathcal{A}^2_{18} + 2$ because they represent 2-complementary regions of the aggregated region.
6. $\mathcal{A}^2_{18} + 2$ is finally absorbed into $\mathcal{A}^1_{19} + 1$ because $\mathcal{A}^2_{18} + 2$ represents a subregion of what is represented by $\mathcal{A}^1_{19} + 1$.

2.5 Implementation in Polaris

A *Region Processor* is the module we are implementing in Polaris for supporting the basic operations for Access Region manipulation, which are described in Section 2.4. Each region operation within the Region Processor is structured as a decision tree, as shown in Figure 2.18. Each decision is a simple one, such as "is the lower bound value of $\mathcal{R}_1$ larger than the upper bound value of $\mathcal{R}_2$". A compiler implementor can easily extend the decision tree structure to improve accuracy by simply adding more branches to the tree.

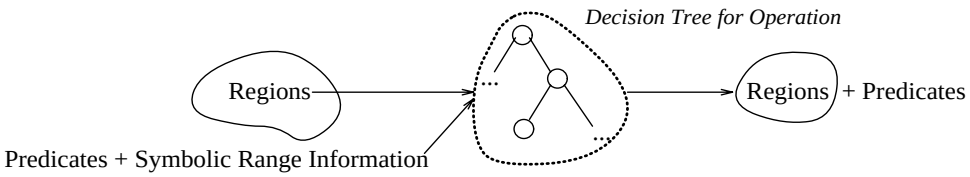

Figure 2.18: Region processing in Polaris

We decided to design the region operations for regions with similar structures. However, if the regions do not meet the *similarity* constraints, our representation allows us to simplify the regions and still use our operations, or else choose different techniques which might be a better fit [67, 60].

Since the access regions being operated on will often involve unknown values, the Region Processor is designed to proceed with the operations by making favorable assumptions, and to return the expressions representing those assumptions as conditions under which the result is correct. The condition expressions could be evaluated at runtime, when perfect information is available, to choose between alternative transformations.

The work of the Region Processor is supported by two important features of the Polaris compiler. First, the program is represented in Gated Single Assignment (GSA) form [70]. The GSA form makes it easy to determine which definition of a variable is used at any point in the program, and the conditions under which a certain definition is used. Second, the symbolic manipulation modules in Polaris, such as range propagation and the range dictionary, make the value ranges for variables available at any point in the program. These features provide a mechanism which can determine relationships between variables even when their exact values are unknown. This rich environment was crucial to the success of the Range Test [14], and can enable many of the symbolic operations of the Region Processor.

The predicate for an access region $\mathcal{R}$ may be thought of as the pertinent information found in the gating information from the GSA form and in the value range constraints. These conditions and values provide the symbolic manipulation context for making decisions within the region processor. Thus, implicit in the descriptions of the operations in Section 2.4 is the use of the predicates, which provide the ability to reason symbolically about the relationships between variables.

2.6 Usage of Access Region Analysis

The array privatizer, dependence analyzer, and communications generation modules can all make use of the Region Processor. The advantages of this are many. First, the Region Processor should greatly simplify each of these modules by removing all region-handling code from them and letting them concentrate on strategies for using the results of the region processing and the conditions produced by it. Second, it promotes a demand-driven style of compilation, as opposed to a pass-based style. The use of a consistent region representation and framework makes it possible to pass Access Regions between several compiler modules, which are called as they are needed. Third, the conditions generated by the Region Processor make it possible to parallelize loops at run-time instead of serializing them for lack of information.

Each compiler module uses the region aggregation algorithm to combine the array accesses of interest in a program section as shown in Figure 2.19. The program section could be a loop-nest, a subroutine, or any part of the input program. The module then can use the Region Processor to perform any of the other operations on the aggregated regions. If the Region Processor lacks the information it needs to perform the operation, it returns a condition under which the result is correct.

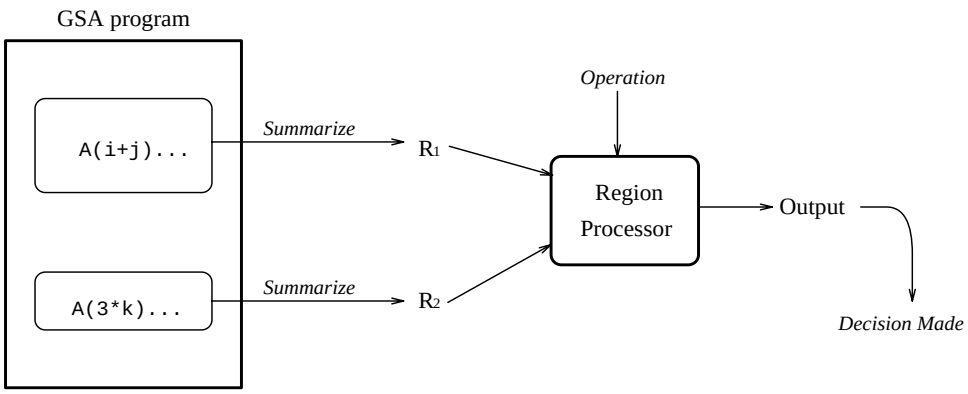

Figure 2.19: Use of the Region Processor

This framework also could be used as a basis for interprocedural analysis. The regions of each array which are accessed in a subroutine could be summarized in the form of an Access Region and then used in the analysis just like any other Access Region.

2.6.1 Dependence Analysis

Using the powerful expressiveness of Access Regions and the Region Processor support, we can reformulate the definition of dependences in terms of regions and use that reformulation to parallelize programs. In [36], we formally propose a new dependence test, called the *Region Test*. Since we detailed the test in that previous work, we will only briefly mention the basic concepts of the Region Test in this section.

The Region Test is designed to solve dependence problems at variable granularity in that it can test for loop-based dependence between individual array references, dependence for whole loops, or dependence between arbitrarily large code sections for a particular array. For each granularity of dependence problems, we consider three types of dependence between array accesses: *flow*, *anti*, and *output* [74]. First, for intraprocedural loop-based dependence problems in a loop

```
do I = l, u, s
    ··· X(f(I)) ···
enddo
```

whose index starts at the value l and strides in steps of s to an upper bound of u, we reformulate the definitions for the three types of cross-iteration dependences in terms of array regions in the form of Access Regions. The basis for the reformulation is that a cross-iteration dependence exists if and only if the portion of an array accessed prior to an arbitrary iteration overlaps with the portion accessed in that iteration or later. To state this in terms of values of the loop indices, let t be an arbitray iteration of the I-loop between l and u. Then, we say that there is a cross-iteration dependence if and only if the portion of an array accessed in iterations from l to $t - s$ overlaps with the portion accessed in interations from t to u.

We write the region-based definition of a cross-iteration flow dependence for an array X in such a loop as follows:

$$\texttt{FLOW} : [\mathbf{W}(\texttt{X})_{l:t-s:s} \cap \mathbf{R}(\texttt{X})_{t:u:s}]_{t=l+s:u:s}.$$

This refers to aggregating the write ($\mathbf{W}$) accesses to the array over the iterations between l and $t - s$, and then intersecting that region with the aggregated read ($\mathbf{R}$) accesses of X in the iterations between t and u. The interation t can be any iteration in the range from $l + s$ to u. No flow dependence exists if the intersection is empty; otherwise, a flow dependence exists. We can define an anti dependence in a similar way. In this case, the read comes first, so we denote it as

$$\texttt{ANTI} : [\mathbf{R}(\texttt{X})_{l:t-s:s} \cap \mathbf{W}(\texttt{X})_{t:u:s}]_{t=l+s:u:s}.$$

Similarly, for output dependence, we express it as

$$\texttt{OUTPUT} : [\mathbf{W}(\texttt{X})_{l:t-s:s} \cap \mathbf{W}(\texttt{X})_{t:u:s}]_{t=l+s:u:s}.$$

This intraprocedural loop-based dependence technique allows us to solve the dependence problems presented in Section 2.1. For the loop from TFFT2, shown in Figure 2.2, we can employ the function `coalesce_region` to summarize the regions accessed at each loop level. The region of `X` written in the first `J` loop may be calculated by first aggregating the writes, and then coalescing the regions to produce the abstract form:

$$\mathbf{W}(\mathtt{X})_{0:t-1:1} = \mathbf{W}(\mathtt{X})_{t:2^M-1:1} = (\mathcal{A}^1_{2^M} + 1, \mathrm{MUST}, \mathrm{WRITE}, \mathbf{T}).$$

Similarly, the region of `X` read in the second `J` loop becomes:

$$\mathbf{R}(\mathtt{X})_{0:t-1:1} = \mathbf{R}(\mathtt{X})_{t:2^M-1:1} = (\mathcal{A}^1_{2^M} + 1, \mathrm{MUST}, \mathrm{READ}, \mathbf{T}).$$

By subtracting the READ region from the WRITE region, we get an empty region, showing that the read region is equal to the write region regardless of the value of t. This implies that there exist anti and output dependences in the `I`-loop. Therefore, following the array privatization algorithm [69], we can remove these dependences by privatizing `X`. Since `X` has no cross-iteration flow dependence in the `I`-loop, it no longer blocks the loop from being parallelized.

The loop in Figure 2.2 is the most important in TFFT2; thus, good speedups were not possible without parallelizing this loop. To measure the effectiveness of the Region Test, we tested the performance of the TFFT2 benchmark on the Cray T3D [23] both with and without applying the Region Test. The results, shown in Table 2.1, demonstrate that the Region Test allows us to significantly reduce the parallel execution times.

PEs	Parallelized without Region Test (sec)	Parallelized with Region Test (sec)	Speedup improvement
4	764	188	4.06
8	634	114	5.56
16	563	69.0	8.16
32	540	47.0	11.5
64	522	36.0	14.5

Table 2.1: Comparison of parallel execution times

The Access Region representation also helps us to summarize access patterns in order to avoid inlining for parallelization problems involving loops with subroutine calls. Figure 2.20 shows a code example similar to some loops in TURB3D. The array U undergoes reshaping as it is passed to the subroutine, but this is not a problem since all array references are linearized by the Access Region representation anyway.

```
subroutine caller
common U(N,M,L)
 ...
do I = 1, M
   call foo(U(1,I,1))
enddo
 ...
return
  ⋮
subroutine foo(X)
common X(*)
 ...
do J = 0, L-1
   do K = 1, N
      X(J*N*M+K) = ···
   enddo
enddo
 ...
return
```

Figure 2.20: Simplified code example from TURB3D program

The access descriptor for the region accessed in the subroutine foo is $\mathcal{A}^{1,NM}_{N-1,(L-1)NM} + 1$. This same pattern happens regardless of the argument passed to the subroutine, making this a useful summary of the access. Arguments passed to the subroutine supply different starting points for that access pattern. In the Figure, the I-loop containing the call to foo produces an additional stride (N) for the access pattern, making the write accesses for prior iterations of the outer loop

$$\mathbf{W}(\mathtt{U})_{1:t-1:1} = (\mathcal{A}^{1,NM,N}_{N-1,(L-1)NM,(t-2)N} + 1, \cdots),$$

and the write accesses for later iterations

$$\mathbf{W}(\mathtt{U})_{t:M:1} = (\mathcal{A}^{1,NM,N}_{N-1,(L-1)NM,(M-t+1)N} + (t-1)N + 1, \cdots).$$

Since intersection $\mathbf{W}(\mathtt{U})_{1:t-1:1} \cap \mathbf{W}(\mathtt{U})_{t:M:1}$ is empty, we can prove that there is no dependence in the I-loop, thus making the loop parallel.

Many compilers give up when confronted with unknown values. Run-time parallelization techniques have been proposed [61, 63] for these cases, but, until now, these have required the potentially high overhead involved in checking individual array accesses for dependence at run-time. We suggest a different approach where the conditions, which we call *safe conditions*, can be extracted from the code to test at run-time.

Safe conditions are those under which it is safe to parallelize a section of code. Given a loop with safe conditions, we can produce a *two-version loop* which will check the safe condition at run-time and choose between a parallel version and a serial version of the loop as follows:

```
if (safe-conditions) then
        Parallelized Loop
else
        Serialized Loop
endif
```

We can apply this simple run-time test with safe conditions to the example in Figure 2.1. Here, we first need the condition extraction function of the Region Processor. The use of predicates of the Access Region inside the Region Processor gives the Region Test a way to extract the safe conditions from the code. In the example, the aggregated write region for FX in the loop is

$$(\mathcal{A}^{1,N}_{2,2(M-1)} + 1, \text{MUST}, \text{WRITE}, \mathbf{T}).$$

In order to determine the dependences involved, we must show that the second stride (N) is greater than the first span (2), which allows the access to FX on each iteration to stride beyond the array elements accessed on the previous iteration. Since N is unknown, the Region Processor would generate the condition $N > 2$ as a safe condition. Now, using the condition, we can then produce code which will check the condition at run time, choosing between a parallel version and a serial version of the loop.

We have observed that this predicate-based run-time test is useful for various programs, such as MDG and OCEAN from the Perfect benchmarks.

2.6.2 Communication Analysis

Single-sided communication protocols [24, 27, 45, 48] in the form of PUT/GET primitives have been rapidly gaining wide acceptance. A great advantage of PUT/GET primitives is that their use of asynchronous data communication works well with the shared-memory programming paradigm, which is also assumed by Polaris.

PUT/GET operations are useful for removing anti and output dependences, as illustrated in [28]. By using the function `coalesce_region` and the region aggregation algorithm in the loop shown in Figure 2.21, the Region Processor calculates the region of *upwards exposed uses* [1] of array V as

$$\mathcal{A}^1_{MN+N-((t-1)N+1)} + (t-1)N + 1$$

for each iteration $I = t$. The write region for the same iteration is

$$\mathcal{A}^1_{N-1} + (t-1)N + 1.$$

The write region for all the following iterations from $I = t+1$ to M is

$$\mathcal{A}^1_{MN+N-(tN+1)} + tN + 1.$$

The region intersection algorithm would report an overlap between

$$\mathcal{A}^1_{MN+N-((t-1)N+1)} + (t-1)N + 1 \text{ and } \mathcal{A}^1_{MN+N-(tN+1)} + tN + 1,$$

implying an anti dependence. To make the loop nest parallel, we can eliminate the dependence by privatizing the array V and generating a GET for the upwards exposed use region.

```
do I=1,M
   do K=1,N
      do L=I,1+M-I
         ··· = V(K+(I-1)*N+L*N)
      enddo
      V(K+(I-1)*N) = V(K+(I-1)*N) ···
   enddo
enddo
```

Figure 2.21: Code from MDG, with the induction variables substituted

We also use PUT/GETs to implement the *shared data copying scheme* [54, 55] in SPMD parallel codes for distributed memory multiprocessors. In the scheme, we use shared memory as a repository of values for use in private memory. Before a parallel loop starts, the processors copy all data that is used in the loop from shared memory into private memory. After the loop execution completes, the processors copy the results back to shared memory so that all the processors have access to the results. By doing so, we can localize most of the data that are used by the processors in the computations. In Chapter 3, we will give the details of this scheme and how this is used to optimize communication in the code generation for distributed memory multiprocessors.

In the data copying scheme, gathering precise array access information into a flexible representation is essential for supporting efficient copy(PUT/GET) operations. Our recent experiments with benchmarks showed that our implementation of the scheme, based on our new representation, has been successful, as presented in Chapter 4.

CHAPTER 3

AUTOMATIC CODE TRANSFORMATION

3.1 Target Parallel Programming and Machine Models

Starting with a conventional Fortran77 program, the Polaris code transformation procedure shown in Figure 1.4 generates a parallel version for distributed memory multiprocessors. The parallel version code is a shared-memory program which has the following characteristics:

- Single Program Multiple Data (SPMD) paradigm,
- explicit synchronization through barriers and locks,
- explicit data declaration as `private` or `shared`, and
- asynchronous communication with PUT/GET primitives.

The parallel program is in SPMD form and follows the master/slave (or the master/worker) model [18] in which one of the parallel processors, *the master*, executes all sequential regions, and the other processors, *the slaves*, participate only in the computations of parallel regions.

Barriers are inserted around parallel regions, such as parallel loops, and calls to subprograms containing parallel regions, and are used to explicitly control the flow of execution of masters and slaves. That is, the slaves wait at barriers while the master is in a sequential region. When the master hits a barrier preceding a parallel region, the slaves are released to join the computation of the region. The slaves return to the barrier after they complete the parallel region. Locks are useful for establishing critical sections where global operations using shared variables, such as reduction operations, are performed.

The program data are explicitly declared as either `private` or `shared`. A private variable is replicated to every local memory. In contrast, only one object for each shared variable exists in the system and is accessible to all processors. Shared arrays can be given `BLOCK` or `CYCLIC` distribution, similar to those in other parallel programming models [19, 64].

PUT/GET operations will be used to implement prefetching and poststoring schemes for the control of data movement between shared memory and private memory in the data localization phase shown in Figure 1.4. One reason we have chosen to use PUT/GET operations is that they work well with the shared-memory programming paradigm in that they allow the processors to asynchronously access any data object in the system whether the object is private or shared. Another reason is that PUT/GET operations have been rapidly gaining wide acceptance. Several portable shared-memory programming models [27, 45, 48] supporting PUT/GET operations already have been implemented on ordinary message-passing machines, such as the IBM SP-1/2 [37], Intel Paragon [16], and TMC CM-5 [34]. Furthermore, several existing and newly

proposed large-scale machines directly support these primitives in hardware [23, 24, 25, 32], which reduces the effect of the increased communication overhead resulting from the data copy operations.

Our techniques can be applied to any distributed memory system that supports all these characteristics by either hardware or software. However, for our techniques to be more effective, we specifically target the machines that provide special H/W mechanisms in order to support fast synchronization and low-latency asynchronous remote memory access for global address space operations.

3.2 Polaris Modules for Code Transformation

In this section, we describe the compiler modules used to implement each phase of the Polaris code transformation procedure shown in Figure 1.4. Among these modules, the modules that implement parallelism detection, data privatization, and data localization rely on the analysis of array access patterns in a program.

As discussed earlier, array access patterns must be expressed in some standard representation for which Polaris uses the triplet notation. However, we have found that the triplet notation does not always accurately capture complex access patterns that are commonly encountered in real programs and, thus, prevents our complier modules from carrying out their techniques in some important cases. For these cases, the Access Region representation has proven to be effective because of the increased accuracy of the access pattern analysis which in turn improved the effectiveness of the Polaris code transformation procedure.

In the work presented here, we make use of both the traditional techniques based on the triplet notation and the new techniques based on the Access Region.

3.2.1 Parallelism Detection

The techniques implemented in Polaris to detect parallelism include: dependence analysis, inlining, induction variable substitution, reduction recognition, and privatization [14, 31, 59, 69]. Since these techniques are well documented in other places, we will not discuss them any further in this dissertation.

The loops that are identified as parallel by these techniques are marked with `parallel` directives. For instance, the I-loop in Figure 3.1 has been identified to have no cross-loop dependence and, thus, is marked with a `parallel` directive for later modules searching for parallel loops.

```
cdir$ parallel (I)
      do I = 1, N
         X(I) = ···
      enddo
      print *, X(N), N
       ···
```

Figure 3.1: Code example with a parallel loop in Polaris

The dependence analysis techniques in Polaris have been developed to be effective mainly for the important loop patterns that commonly occur in scientific programs. This strategy has worked well on medium- and small-scale UMA shared-memory multiprocessors, where a few important loops in a program dominate the overall performance [13]. In large-scale multiprocessors, however, more accuracy has been found to be necessary because small unimportant loops often become important to determine the multiprocessor speedups, as illustrated in Table 3.1 where the *serial coverage* is the percentage of execution time consumed by serial loops.

As illustrated in the table, the serial loop execution time can be ignored for eight or fewer processors; but, in the spirit of Amdahl's law, the serial loop execution time becomes significant

Processors	Parallel Loops (sec)	Serial Loops (sec)	Serial Coverage (%)
2	120	3.2	2.7
8	31	2.8	8.2
64	6.1	2.7	31

Table 3.1: Loop execution times using the MDG benchmark on the Cray T3D

as the number of processors increases. We, therefore, found it necessary to parallelize very small loops to generate high quality parallel programs for distributed memory multiprocessors, thus motivating us to develop the *Region Test* discussed in the previous chapter.

The Region test is similar to the Range Test [12], the most powerful dependence test used by Polaris, in the sense that both are based on array access region analysis; but, the experiments showed that the Region Test can break many dependences that the Range Test conservatively assume present mainly because of the accuracy of the Access Region representation. Capitalizing on the powerful expressiveness of the Access Region, the Region Test is especially useful for parallelizing loops with very complicated access patterns. Consequently, the test contributes significantly to scalable speedups, as demonstrated in the experimental results (see Chapter 4).

3.2.2 Work Partitioning

We transform the original program annotated with `parallel` directives into the SPMD parallel form described in Section 3.1, where all parallel loop iterations are statically assigned to the processors. We use the same loop scheduling scheme for the distributed memory architectures as Polaris currently uses for UMA architectures: *cyclic* schedules for triangular loops, and *block* schedules for square loops [57]. According to our experiments, these conventional static scheduling schemes provide relatively good load balancing between processors at low loop scheduling costs in most cases.

Figure 3.2 shows the results after the code in Figure 3.1 is transformed by the work partitioning module. $\mathcal{P}$ is the number of processors and *my_pid* is the processor ID number between 0 and $\mathcal{P}$-1.

```
        ILOW = my_pid+1
        IHIGH = N
        do I = ILOW, IHIGH, P
           X(I) = ···
        enddo
        if (slave) goto wait
        print *, X(N), N
wait:   call barrier()
         ···
```

Figure 3.2: SPMD code with the **cyclic** scheduled parallel loop from Figure 3.1

Notice that the `print` statement, as is usually the case with most I/O statements, is in a sequential region and, therefore, the slaves skip the statement by jumping to a barrier and wait there for the master.

For the case of loops containing *reductions* [59, 74] it is necessary to modify the simple strategy used by Polaris for a parallel loop without reductions, as shown in Figure 3.3.

```
cdir$ parallel (J)
      do J = 1, M
         do I = 1, N
            ···
            A(I) = A(I) + ···
            ···
         enddo
      enddo
```

Figure 3.3: A parallel loop with reduction

The original array `A` in Figure 3.3 will be declared shared in the SPDM parallel code, and a private array `A'` with the same size and type as `A` will substitute for `A` in the reduction loop. The transformed loop, which is shown in Figure 3.4, consists of three parts: the loop preamble, the loop body, and the loop postamble. In the preamble, `A'` is initialized by all processors before

the loop execution. After the loop body execution completes, the partial results stored in A′ are gathered into A in the postamble.

```
cdir$ private A′, ···
cdir$ shared A(distribution), ···
        ⋮
cdir$ loop preamble
      JLOW = M/P*my_pid+1
      JHIGH = M/P*(my_pid+1)
      A′(1:N) = 0.0
cdir$ loop body
      do J = JLOW, JHIGH
         do I = 1, N
             ···
            A′(I) = A′(I) + ···
             ···
         enddo
      enddo
cdir$ loop postamble
      call set_lock(lock)
      A(1:N) = A(1:N)+A′(1:N)
      call clear_lock(lock)
```

Figure 3.4: SPMD code with the **block** scheduled parallel loop from Figure 3.3

This parallel version of the loop has the disadvantage that the postamble is executed serially because it is in a critical section. This works well for a few processors, but a different strategy is needed for a large number of processors. In our current implementation, A′ within each processor is (conceptually) divided into $\mathcal{P}$ sections. The postamble consists of two phases. First, the i-th section of all processors, $1 \leq i \leq \mathcal{P}$, is copied into the i-th processor. Then all processors add in parallel the $\mathcal{P}$ sections copied into them. As expected, this approach of parallelizing the postamble has an important impact on performance. This is illustrated in Table 3.2, which contains the measurement of loop INTERF_do1000 from the benchmark MDG.

If several loops in a multiply-nested loop nest are parallel, then our current loop scheduling scheme parallelizes only one loop (usually the outermost loop) in the nest. Whereas this scheme

Processors	Preamble (sec)	Loop Body (sec)	Serial Postamble (sec)	Parallel Postamble (sec)
2	0.014	260	0.39	0.10
64	0.017	11	2.7	0.13

Table 3.2: Comparison of reduction parallel loops execution times

simplifies the loop scheduling algorithm, it has the drawback that sometimes we may not fully utilize the degree of parallelism in a multiply-nested parallel loop nest, particularly when the number of iterations of the outermost loop is small relative to the number of processors.

To overcome this drawback, the traditional loop interchanging technique is recommended to move more practical inner loops to the outer level so that an inner loop can be parallelized. A more general scheme for this problem will be a multi-level loop distribution onto processors where the iteration space of multiple loops in a loop nest is hierarchically allocated to processors. Although we do not consider this scheme in the work of this dissertation, this is another possible technique to take full advantage of multiply-nested parallel loops.

3.2.3 Data Privatization

Given a partition of the computation, the Polaris privatizer analyzes the data regions accessed by each processor and declares data as private if the data is always accessed by the same processor. The benefits of data privatization are many. One is that it removes some types of *output* and *anti* dependences, as discussed in [28]. For instance, the Region Test would determine that the loop `PREDIC_do1000` in Figure 3.5 has only anti dependences; hence, this loop can be parallelized if the dependences are eliminated by *renaming* the array `X` as a private array and copying the upwards exposed use [1] regions of `X` to the private array with PUT/GET primitives. The parallel version of this loop is shown in Section 3.2.4.

```
do I = 1, M
   do K = 1, N
      do L = 1, 1+M-I
         ··· = X(K+(I-1)*N+L*N)
      enddo
      X(K+(I-1)*N) = X(K+(I-1)*N) ···
   enddo
enddo
```

Figure 3.5: Loop from the MDG benchmark, with induction variables substituted

Data privatization is important to distributed memory architectures because it increases not only parallelism, but also the chance that the processors fetch their data from local memories, thus reducing the overall communication overhead. Furthermore, in some non-cache coherent multiprocessor systems [23, 25], shared data may not be cached, thereby causing performance degradation whenever the computation uses shared data. Data privatization presents additional benefits in these machines by providing more chances for processors to use data caches in their computations.

Data privatization in Polaris consists of two passes: *loop* and *procedure* privatizers. The loop privatizer [69] tries to find parts of a data object that are written prior to being read in each iteration of a loop. In order to do so, it identifies the read region whose access is covered by the write region on the same iteration by aggregating all writes and reads to the data object and intersecting the two regions. The intersection is the portion of the object which is privatizable. The scope of the loop privatizer is confined to a loop nest. For example, in Figure 3.6, the loop privatizer identifies `TEMP` as privatizable, but not array `W` which is read in the `I`-loop and written in another loop within the subroutine goo.

The procedure privatization is a variation of the loop privatization. Its purpose is to try to find parts of a data object written in the previous program section prior to being used in the current section. A program section can be any code segment within a program, including loops

```
subroutine foo(V,L)
real W(LEN,2), V(*) ···
 ···
call goo(V,W,L)
 ···
ILOW = L/P*my_pid+1
IHIGH = L/P*(my_pid+1)
do I = ILOW, IHIGH
   TEMP = W(I,1) + W(I,2)
   V(I) = TEMP ···
enddo
  ⋮
subroutine goo(A,B,L)
 ···
JLOW = L/P*my_pid+1
JHIGH = L/P*(my_pid+1)
do J = JLOW, JHIGH
   B(J,1) = ···
   B(J,2) = ···
enddo
  ···
```

Figure 3.6: Simplified code from the SU2COR benchmark after work partitioning

and subroutines. In Figure 3.6, the privatizer first aggregates the region `W(ILOW:IHIGH,1:2)` read in the I-loop and the region `W(JLOW:JHIGH,1:2)` written in goo. Then, it determines that the write region covers the read region in all processors by proving that `JLOW` $\leq$ `ILOW` and `JHIGH` $\geq$ `IHIGH`, from which `W` is identified as privatizable. It is clear from the code that each processor accesses at most `LEN/`$\mathcal{P}$ elements in the first dimension of `W` in the SPMD code; hence, the privatizer redeclares `W` to have `W(LEN/`$\mathcal{P}$`,2)` and changes the subscript expressions of `W` in `foo`. Notice that the formal parameter `B` in goo also must be declared as private in order to match the private attribute of the actual parameter `W` in `foo`.

Figure 3.7 shows the code after data privatization is applied. We now have five privatized variables in `foo`. Notice that the formal parameter `V` is not privatized. One reason for this is

```
      subroutine foo(V,L)
      real W(LEN/P,2), V(*) ···
cdir$ private TEMP, W, I, ILOW, IHIGH
       ···
      call goo(V,W,L)
       ···
      ILOW = L/P*my_pid+1
      IHIGH = L/P*(my_pid+1)
      do I = ILOW, IHIGH
         TEMP = W(I-ILOW+1,1) + W(I-ILOW+1,2)
         V(I) = TEMP ···
      enddo
        ⋮
      subroutine goo(A,B,L)
       ···
      JLOW = L/P*my_pid+1
      JHIGH = L/P*(my_pid+1)
      do J = JLOW, JHIGH
         B(J-JLOW+1,1) = ···
         B(J-JLOW+1,2) = ···
      enddo
        ···
```

Figure 3.7: Code after applying data privatization to the code from Figure 3.6

that the actual parameter corresponding to V in the subroutine which calls foo could be shared. We will discuss how to deal with these *non-privatized* arrays in the next section.

3.2.4 Data Distribution and Localization

We declare all non-privatized arrays as shared and BLOCK-distribute all their dimensions. Then, we apply the *shared data copying scheme* [54] to localize most of the accesses to these shared arrays that are made by the processors. In the scheme, shared memory is used as a repository of values for private memory, as shown in Figure 3.8. In current distributed memory systems, the shared memory is usually a logical collection of the shared address portions of all local memories.

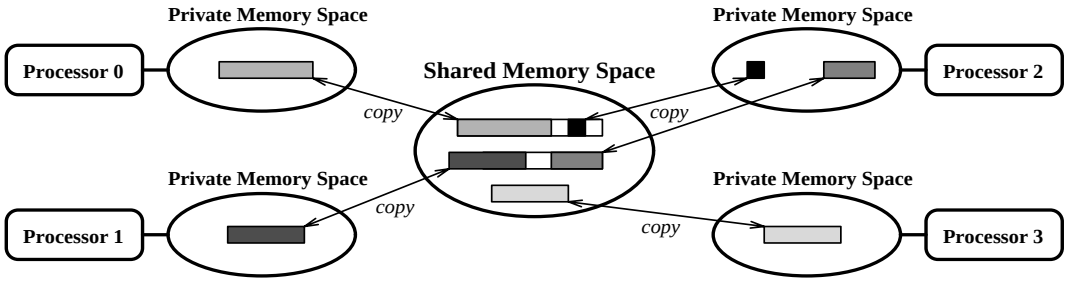

Figure 3.8: Conceptual view of the shared data copying scheme for four processors

In order to apply the scheme, we first change references to shared arrays to the corresponding private arrays in each loop. Before a loop starts, the processors copy all portions of shared arrays that are read in the loop into private memory. After the loop execution completes, the processors copy the updated results back to shared arrays so that all the processors have access to the new results. By doing so, most of the work is done on private data objects. This scheme brings us the following benefits:

1. We can take advantage of prefetching and poststoring strategies. Unlike cache memory, the data copied into private memory is fully software controllable; that is, the data would never be flushed out until explicitly done so by the program. Hence, as long as the local memory space is available, a processor can prefetch data anytime before it is needed and poststore it sometime after the computation.

2. We can reduce communication overhead by copying data blocks instead of single data items. This scheme is particularly useful when the data distribution requirements of a program are dynamic, as in the case of array `X` in Figure 1.3. Instead of total data redistribution of the elements of `X`, the processors can prefetch and poststore the exact portions of `X` that they access in each loop.

3. Similar to data privatization, the scheme provides a benefit of helping processors to better utilize caches in non-cache coherent machines [23, 25].

In order to bring about all these benefits, this scheme should carry out efficient copying operations based on a very precise access region analysis. In Chapter 2, we discussed the access region analysis used in our code transformation algorithm. Using access region analysis, we identify the array regions to be copied and direct where, in a loop, these regions are to be copied using PUT/GETs. For example, in Figure 3.7, access region analysis would help us identify that the region `V(ILOW:IHIGH)` is written. This region would be represented by access descriptor

$$\mathcal{A}^{1}_{\texttt{IHIGH-ILOW}} + \texttt{ILOW}$$

in the Access Region representation. This region information is used to generate the appropriate PUT/GET primitives. Our primitives are specified as follows:

`polaris_put/get(`*shared-address*`,` *private-address*`,` *shared-stride*`,` *private-stride*`,` *length*`)`

where the routine `polaris_put` transfers *length* words of data from *private-address* in private memory space to *shared-address* in shared memory space. The *shared-stride* and *private-stride* are, respectively, the strides of the shared and private arrays being transfered. The routine `polaris_get` works the same way as `polaris_put` except that the source and destination of data movement are reversed.

Figure 3.9 shows the code that results after applying the scheme to the subroutine `foo` in Figure 3.7. In the subroutine, the non-privatized array `V` is declared as shared with `BLOCK` distribution. Our experience with scientific codes reveals the following facts:

1. Block distribution generally works well with the data copying scheme because many loops access arrays in a contiguous way.

```
      subroutine foo(V,L)
      real W(LEN/P,2), V(*), v(*) ···
cdir$ private TEMP, W, I, ILOW, IHIGH, v
cdir$ shared V(BLOCK)
       ···
      call goo(V,W,L)
       ···
      ILOW = L/P*my_pid+1
      IHIGH = L/P*(my_pid+1)
      call alloc(v(1:IHIGH-ILOW+1))
      do I = ILOW, IHIGH
         TEMP = W(I-ILOW+1,1) + W(I-ILOW+1,2)
         v(I-ILOW+1) = TEMP ···
      enddo
      call polaris_put(V(ILOW),v(1), 1, 1, IHIGH-ILOW+1)
      call dealloc(v)
        ···
```

Figure 3.9: Subroutine `foo` in Figure 3.7 after data localization

2. Data access patterns of real programs are often dynamic and, consequently, it is impractical to find a single data distribution for those access patterns.

3. Block distribution facilitates the calculation of the target location of the data to be copied within calls to either the routine `polaris_put` or `polaris_get`. Block distribution always guarantees no more than $\mathcal{P}$ PUT/GET calls per data copy for a $\mathcal{P}$-processor partition.

These observations support our choice of the block distribution policy for shared arrays in our code transformation.

The array region accessed by the reference `V(I)` within the `I`-loop shown in Figure 3.6 is represented by access descriptor $\mathcal{A}^1_{\texttt{IHIGH-ILOW}} + \texttt{ILOW}$, as discussed in Chapter 2. Using the access region information for array `V`, the shared data copying scheme generates a `polaris_put` call with shared address `V(ILOW)` and private address `v(1)`, where `v` is a private array for the shared array `V`, as shown in Figure 3.9.

In our transformation, private arrays are dynamically allocated and deallocated with the functions `alloc` and `dealloc`. When they are allocated, their lower bounds are normalized to one; thus, array reference `v(1)` indicates the first address of the private array v in Figure 3.9. The strides and the length of data for transfer in this `polaris_put` call are derived from the stride/span pair in access descriptor $\mathcal{A}^1_{\texttt{IHIGH-ILOW}} + \texttt{ILOW}$. Notice here that the I-loop no longer contains remote memory access.

```
cdir$ private x, ···
cdir$ shared X(BLOCK), ···
       ···
      call alloc(x(1:(M-ILOW+2)*N))
      call polaris_get(X(1+(ILOW-1)*N),x(1), 1, 1, (M-ILOW+2)*N)
      do I = ILOW, IHIGH
         do K = 1, N
            do L = 1, 1+M-I
               ··· = x(K+(I-ILOW)*N+L*N)
            enddo
            x(K+(I-ILOW)*N) = x(K+(I-ILOW)*N) ···
         enddo
      enddo
      call polaris_put(X(1+(ILOW-1)*N),x(1), 1, 1, (IHIGH-ILOW+1)*N)
      call dealloc(x)
        ···
```

Figure 3.10: Loop from Figure 3.5, parallelized after anti dependence has been removed

Figure 3.10 shows the parallelized version for the loop `PREDIC_do1000` from Figure 3.5 which contains anti dependence. As discussed in Section 3.2.3, the PUT/GET primitives also are of use to eliminate anti and output dependences. As mentioned earlier, generating `polaris_put/get` calls requires access region information about array `X` within the I-loop in Figure 3.5. For this purpose, we compute the access descriptor

$$\mathcal{A}^1_{\texttt{(M-ILOW+2)N-1}} + 1 + (\texttt{ILOW} - 1)\texttt{N}\ ^{\dagger}$$

[†]This corresponds to `X(1+(ILOW-1)*N : N+M*N : 1)` in triplet notation.

to represent the upwards exposed region of array X in the I-loop, and the access descriptor

$$\mathcal{A}^1_{(\texttt{IHIGH}-\texttt{ILOW}+1)\texttt{N}-1} + 1 + (\texttt{ILOW} - 1)\texttt{N}^{\ddagger}$$

to represent the downwards exposed region of X. In the parallelized loop from Figure 3.10, the processors that execute the block-scheduled loop iterations from ILOW to IHIGH *get* the upwards exposed region and *put* the downwards exposed region between the shared array X and the privately-owned array x, in a similar way shown in the example with Figure 3.9.

One important issue in the shared data copying scheme is to avoid consuming an excessive amount of space for private data. Consider, for example, the loop in Figure 3.11.

```
real X(100,100)
   ...
 <-- copy-level 1
do K = 1, L
    <-- copy-level 2
   do J = 1, N
       <-- copy-level 3
      do I = 1, M
         ... = X(I,J)
      enddo
   enddo
enddo
```

Figure 3.11: Choices of copy-levels for a GET operation on X

In the example, the *copy-level* is a loop nest level at which the elements of the array X are copied by using the Polaris PUT/GET primitives. In the example, if we perform GET operations at copy-level 1 or 2, we have to allocate $O(N \cdot M)$ words on each processor. We can save more private memory space by placing the GET primitive at copy-level 3 since each processor now needs only $O(M)$ words. If we are concerned only with memory space, copy-level 3 seems to be a better place for the copying. However, there is an obvious trade-off between

[‡]This corresponds to X(1+(ILOW-1)*N : IHIGH*N : 1) in triplet notation.

time and space. Copying at copy-level 1 may reduce communication costs because a processor needs to make only $O(1)$ `polaris_get` calls for a large data block; on the opposite side, copying at copy-level 3 requires a processor to make $O(L \cdot N)$ `polaris_get` calls for smaller fragments of the block.

To handle the issue of determining the copy-level in our transformation, we employ a heuristic algorithm presented in Figure 3.12.

Definition 3.1 *The* **dimension** *of access region $\mathcal{R}$ is the number of stride/span pairs of $\mathcal{R}$.*

In the algorithm, `dim(`$\mathcal{R}$`)` denotes the dimension of the region $\mathcal{R}$. For instance, if region $\mathcal{R}$ has access descriptor $\mathcal{A}^{a,b,c}_{d,e,f}$, then dim($\mathcal{R}$) is three. The parameter P is a program which is generated by the data privatization module described in Section 3.2.3. The current Polaris PUT/GET generation strategy is applied only to parallel loops. The purpose of the routine **gather_copy_levels** is to return a list, called *copylevel-list*, each element of which is a pair of a loop and an array; here, the loop corresponds to the copy-level for the array.

To collect the pair [copy-level,array] in the list *copylevel-list*, all the parallel loops in P are searched for arrays read or written in them. If array $\mathcal{X}$ is written in a parallel loop nest $\mathcal{L}$, then $\mathcal{L}$ is the copy-level for $\mathcal{X}$ and, thus, a `polaris_put` call for the array is placed right after the loop. If $\mathcal{X}$ is also read in $\mathcal{L}$, then a `polaris_get` call is placed right before $\mathcal{L}$ in addition to the `polaris_out` call. As can be seen, if an array is written in a parallel loop nest, then, whether it is read or not, the outermost loop $\mathcal{L}$ of the loop nest will become the copy-level of the array. This is because the access region of an array, written in a Polaris parallel loop, is disjoint between iterations of the loop, making the total access region in a partitioned parallel loop for each processor always smaller than the whole access region of the original array.

```
gather_copy_levels(program P)
   copylevel-list ← ∅
   for each parallel loop nest L in P do
      for each array X used in L do
         determine_copy_level(L, X, copylevel-list)
   return copylevel-list
end

determine_copy_level(loop L, array X, copylevel-list)
   compute access region R for X in L
   if R is read-only then
      if dim(R) ≤ D or L has no inner loops then
         copylevel-list ← [L, X]
      else
         for each inner loop L' of L do
            determine_copy_level(L', X, copylevel-list)
      endif
   else      // R is write-only or read-write
      copylevel-list ← [L, X]
   endif
end
```

Figure 3.12: Heuristic algorithm for the `polaris_put/get` generation

If an array is read only in a loop nest, then we choose as the copy-level for the array the loop(s) in which the dimension of access region for the array is less than or equal to D, a given constant. The value of D is determined when Polaris starts the code transformation. In our experiments, D was set to 1 or 2. However, we generally prefer D to be 1 because it usually allows block copying of consecutive data with a single stride without increasing the private memory space required for the data copying scheme. Note that the algorithm traverses the loop nest $\mathcal{L}$ working from the outermost loop inward, to search for the copy-level for array $\mathcal{X}$. This strategy has been found to be very useful in many cases because it often removes redundant copying in a natural way. For instance, consider the loop nest in Figure 3.11. If the loop nest is parallel, then the subroutine call

determine_copy_level(loop `K`, array `X`, *copylevel-list*)

will be invoked. The access region for array `X` in the `K`-loop will be computed first and, if `X(I,J)` is the only reference in the loop, the result will be $\mathcal{A}^{1,100}_{M-1,(N-1)100}$.

Suppose D is set to 2. Then, D is equal to $\texttt{dim}(\mathcal{R})$, the dimension of the access region of `X` in loop `K`, thereby making the algorithm stop searching other loops inside $\mathcal{L}$ and return copy-level 1 as the copy-level for array `X`. In fact, when D is 2, the dimension of the access region of `X` in the inner loop `J` is also equal to D because the array reference `X(I,J)` is invariant in loop `K`. Thus, copy-level 2 also can be the copy-level for `X`. For $D = 2$, however, copy-level 1 is obviously a better choice than copy-level 2 because both copy-levels consume the same amount of private memory space, yet copy-level 1 needs only $O(1)$ GET operations while copy-level 2 needs $O(L)$ operations which are redundant. By searching the loop nest inward from the outermost loop `K`, we could choose copy-level 1 instead of copy-level 2, and thereby avoid redundant copy operations in this case.

Now, suppose that D is set to 1. The dimensions of the access region of `X` in both loops `K` and `J` are larger than D and, therefore, the subroutine **determine_copy_level** would be called with regard to the innermost loop `I` for testing. The access region for `X` in loop `I` differs from those for the outer loops: $\mathcal{A}^1_{M-1}$. Finally, the dimension of this region is equal to D. Thus, the `I`-loop will be identified as the copy-level for `X`, which corresponds to copy-level 3 in Figure 3.11. As a consequence, the pair [loop `I`, `X`] will be added to the list *copylevel-list*.

Once the routine **gather_copy_levels** returns, the Polaris PUT/GET generator will be invoked to place `polaris_put/get` calls in the program P, according to the information stored in the list *copylevel-list*.

Although we used this naive algorithm to determine the copy-level in our experiments reported in Chapter 4, this algorithm can be improved by using several factors, such as the dimension of the array `X`, and the value of N and M in Figure 3.11. For instance, given $D = 1$, whereas the current Polaris implementation chooses copy-level 3 for array `X` in Figure 3.11, the decision may be changed depending on the value of M. In general, copy-level 3 is useful only when M is large. Therefore, the original algorithm may be modified to choose copy-level 1 if M is found to be too small, thereby reducing copying overhead without significantly consuming private memory.

Another factor affecting the determination of the copy-level is whether the *data pipelining* technique [68, 35] can be exploited in data copying operations. If the implementation of `polaris_put/get` routines is non-blocking, then we may exploit data pipelining. Thus, we choose copy-level 1 in this example regardless of the value of M because data pipelining, by overlapping data copying with loop computation, can offset the data copying overhead necessary to initiate a copying operation.

However, in order to apply data pipelining, the compiler transformation algorithms become more complicated and error-prone. Due to this technical problem, in real cases, we may want to perform copy operations at outer copy levels. In principle, this strategy may consume more space for private data; however in practice, there are several advantages to copying at the outermost levels. First, it is generally possible to stripmine the outermost loop iteration space in a parallel loop nest and to distribute it to processors, thereby linearly decreasing the amount of private memory needed for each processor as the number of processors increases. Also, in order to apply this methodology to distributed memory multiprocessors with high memory latencies, the minimization of communication times becomes very critical. And, the trends in multiprocessor systems indicate that processor speed will grow much faster than memory access and network speeds. Thus, remote access times could be an important runtime factor that needs to be taken into account. On the other hand, because the cost of memory is dramatically reduced, saving space is less important than optimizing time in general cases. Generally, the memory required to allocate private data can be managed using dynamic allocation functions, such as `alloc` and `dealloc/free`. Thus, as soon as the data has been updated in the shared area (e.g., the array access area for `X` in Figure 3.11), this private memory area can be liberated.

CHAPTER 4

EXPERIMENTS AND PERFORMANCE ANALYSIS

To measure the effectiveness of our transformation techniques, the Polaris code generators have been implemented for various types of distributed memory systems, such as the Convex Exemplar [33], and networks of workstations. In this section, we present our recent experiments on one of these systems: the Cray T3D [23].

Section 4.1 will first briefly describe the T3D. Section 4.2 will present several machine-dependent issues addressed by the Polaris code transformation for the T3D. Section 4.3 will briefly discuss the characteristics of the benchmark programs used in our experiments. Section 4.4 will present the results of the preliminary experiments conducted in this work. In the experiments, we used a few basic code transformation algorithms only. This section will describe how we analyzed the parallel performance based on the basic techniques to identify several optimization techniques which would improve the preliminary performance. Section 4.5

will show how much our optimization techniques could improve the results of the preliminary experiments and analyze the parallel performance of the techniques.

4.1 The Cray T3D

In this section, we very briefly describe the hardware characteristics of the Cray T3D and parallel programming models supported in the machine. More detailed descriptions of the machine can be found in other documents[22, 23, 24, 50].

4.1.1 Hardware Characteristics

While the lack of cache coherence in the Cray T3D helps to make the machine affordable and scalable, it also introduces some difficulties in the development of efficient programs [10]. The experimental results presented in this chapter give us hope that these programming difficulties can be overcome with the effective compiler techniques introduced in Chapter 3.

The Cray T3D was designed mainly for large-scale parallel scientific applications. It consists of up to 1024 processing nodes, each containing 2 processing elements(PE) and a local memory. The PEs are 150 MHz DEC Alpha 21064 microprocessors. The interconnection network is a 3-D torus network with high throughput and low latency. Remote memory latency on the T3D ranges from 90 to 130 cycles [5]. Local memory latency is 22 cycles.

The local memory is logically partitioned into private and shared address spaces. The shared memory in the T3D is no more than the collection of the shared address portions of all local memories. Every shared memory access is manipulated by off-chip components before sending the request to the appropriate module. This off-chip manipulation requires 20-30 cycles. The

T3D has a first level on-chip cache which is used for data only in the private address space of the local memory.

The T3D also contains a special tree-like network for global barrier synchronization. We use the T3D barrier to enforce synchronization in the master/slave model, as described in Section 3.1. Barriers do not have a significant impact on overall program execution time in the T3D due to this efficient hardware implementation of barriers in the machine. Table 4.1 shows the performance of the T3D barrier.

PEs	Barrier Time (μsec)
2	1.73
4	1.73
8	1.76
16	1.80
32	1.81
64	1.86
128	1.88
256	1.90

Table 4.1: Barrier performance on the T3D: times are the average of 5000 executions

Furthermore, in our experiments, barriers are executed infrequently. The execution time increases by less than 1% due to barriers in all the programs we studied. For instance, in FLO52 from the Perfect benchmarks, the dynamic counts of barrier calls are about 50,000. The total overhead due to barriers is approximately 0.1 sec on a 64-processor partition, which is less than 0.2 % of the overall execution time of FLO52.

Various communication primitives, including single-sided communication primitives, are supported in the T3D hardware. Therefore, the T3D can compensate for its lack of hardware cache coherence through its hardware mechanisms that facilitate efficient software control of local memory coherency and PUT/GET operations through library primitives [24].

4.1.2 Shared Memory Programming

Like most other distributed memory machines, the Cray T3D supports the message-passing programming models through libraries, such as MPI and PVM. However, the hardware features discussed in the previous subsection differentiate the T3D from other true message passing machines by efficiently supporting shared memory programming on top of its physically distributed memory structures.

Users may develop their parallel programs in a shared memory programming paradigm using the Cray Fortran, called **CRAFT**, which is an extension to Fortran 77 that includes several data parallel programming features from Fortran 90 as well as directives to control parallelism and data placement. CRAFT provides all the necessary characteristics that we described in Section 3.1. It supports the SPMD model operating on a shared address space, and provides a direct interface to the PUT/GET library primitives as well as to several explicit synchronization mechanisms.

In our experiments, each CRAFT process was allocated to a separate physical processor. Data objects can be declared as shared or private. Shared data can be distributed across memory using directives similar to those made popular by High Performance Fortran, Vienna Fortran, and other similar languages [19, 64, 68]. CRAFT uses '`:block`' for block distribution and '`:block(`N`)`' for block-cyclic distribution where N is an integer.

The `do shared` directive of CRAFT is used to mark parallel loops. To illustrate its semantics, consider the loop in Figure 4.1. Its i-th iteration is executed by the PE that owns `A(`i`)` in its local memory. Since the elements of `A` are local to the PE accessing them, the compiler enables the caching of `A`. The elements of `B`, on the other hand, are not cached because some of them may be accessed remotely. Therefore, due to the **on** clause shown in Figure 4.1, CRAFT

users can partition the computation according to the data distribution and thereby enable the caching of some shared data structures.

```
cdir$ shared A(:block(1)), B(:block)
cdir$ do shared (I) on A(I)
      do I = 1, N
         A(I) = B(I)
      enddo
```

Figure 4.1: Code example with a parallel loop in CRAFT

In this example, the computation happens to be distributed according to the owner computes rule, as is done in HPF; however, other distributions also could have been used. Table 4.2 summarizes the differences between CRAFT and HPF, as discussed in [50].

	CRAFT	HPF
Memory Classes	shared/private	implicitly all shared
Data Distribution	W:block(N)	block/cyclic/align
Computation Distribution	programmer-controlled	owner-computes rule
Redistribute statement	NO	YES
Explicit Communication and Synchronization	YES	NO

Table 4.2: Comparison of CRAFT and HPF

Although the machine-supported shared memory model facilitates programming on the T3D, inefficiencies could arise for the following reasons if the program is not carefully designed:

- Unlike cache coherent machines, a remote memory access to a single array element does not take advantage of spatial locality.
- Shared data objects are not cached, even when they are accessed from local memory. Therefore, shared data objects have to be fetched from the remote location every time the program references them. This significantly increases average memory latency and network contention.

To avoid these inefficiencies, we found it necessary to explicitly control caching and data transfer. For this, we have implemented the shared data copying scheme, which was described in Section 3.2.4, based on the PUT/GET library of the T3D. As built on very low level hardware primitives, Since this PUT/GET library in the T3D is built on very low level hardware primitives, it is implemented to be more efficient than other message-passing models currently implemented in the machine. In fact, in experiments conducted on a 16-processor partition [44], the latency of a PUT operation in the T3D was measured at 2 μsec and the peak throughput was measured at 116.8 MB/s, while the equivalent figures for PVM send/receive operations are 63 μsec and 26 MB/s, respectively.

4.2 Code Transformation for the Cray T3D

The code transformation algorithms for the T3D are almost identical to that shown in Figure 1.4 except that there are several additional algorithms to address machine-dependent problems occurring in the code transformation for the T3D.

MPP Fortran extensions, such as CRAFT and HPF[64], help the user attain high performance through distribution of data, while maintaining compatibility with conventional Fortran 77 or Fortran 90. However, it is very difficult to achieve total compatibility. Therefore, CRAFT diverts from several conventional language features in the name of performance. Although there are numerous restrictions imposed by CRAFT, some of them include:

- Fortran's sequence and storage association rules [2] do not apply to shared data.
- Shared data may not be in `EQUIVALENCE` or blank `COMMON`.
- Shared data may not be of type `CHARACTER`.

- Shared data may not be used as file specifiers.
- The dimension size of shared arrays must be a power of two[†].
- Shared formal parameters may not be associated with private actual parameters, and their size and shape must match those of the corresponding actual parameters.
- Shared data cannot be passed as parameters directly to PUT/GET primitives provided by the T3D PUT/GET library [24].

We have developed translation algorithms to resolve the semantical incompatibilities between CRAFT and Fortran 77 which occur when we translate Fortran to CRAFT as a result of all these restrictions. In the code transformation procedure shown in Figure 1.4, we did not explicitly mention these algorithms. This is because the incompatibility problems are interspersed throughout the whole procedure and, thus, they are not dealt with in any single phase within the procedure.

Of these algorithms, we will discuss several important ones in subsequent subsections although there are additional algorithms actually implemented in Polaris to address the incompatibility problems.

We could eliminate many important incompatibility problems using these algorithms, but not all of the problems. We sometimes could not generate correct programs for a few programs due to some incompatiblities or to some T3D hardware problems. For example:

- Too frequently invoked barrier operations sometimes resulted in the failure to guarantee the integrity of shared data.

[†]In the Cray T3E, this restriction is no longer imposed.

- The processor memory is sometimes too small to accommodate the program.

These problems are often unavoidable unless we change the algorithms of the input programs themselves or develop more powerful techniques to eliminate the restrictions causing incompatibility problems.

4.2.1 Renaming

Aliasing always has been an important issue in program analysis in general, and in automatic parallelization in particular [74]. Aliasing also is one of the most difficult problems in automatic parallelization for the T3D. Consider, for example, the following segment extracted from one of the SPEC95fp benchmarks in Figure 4.2. CRAFT cannot distribute a shared data object if it is aliased to other objects of different shapes or types. In the code above, the shared variables within `SCRA` cannot be distributed because of the aliasing of `A` and `I`.

```
subroutine X1
real A, B, C, D, E, F
common /SCRA/ A, B, C, D, E, F
   ⋮
subroutine X2
real G(5)
integer I
common /SCRA/ I, G
   ⋮
```

Figure 4.2: Aliasing in the simplified code from HYDRO2D

In order to address this problem, we interprocedurally check the lifetimes associated with each variable and apply renaming. In the previous example, it can be proven that the lifetime of the values of `SCRA` in `X1` and in `X2` are disjoint. As a result, we could rename either occurrence of the common block without affecting the outcome of the program.

4.2.2 Linearization

Variable linearization and renaming are the most common techniques used to solve problems related to storage association rules. For example, linearization is needed to inline a subroutine call when an actual parameter differs in shape from the corresponding formal parameter.

We use linearization to deal with array equivalences because of the restriction that the size of all shared array dimensions must be a power of 2. When two arrays of different shapes are equivalenced, we linearize the array before dimension expansion. Linearization also helps to save memory. For example, a shared array `A(9,9,9)` is expanded to `A(16,16,16)` without linearization, and `A(1024)` with linearization. Linearization, in this case, saves 3K words. However, we do not want to apply linearization to all multidimensional arrays because linearization makes the program less readable and program analysis more difficult due to the complex subscript expressions.

4.2.3 Array Reshaping and Procedure Cloning

Figure 4.3 shows the code, where we assume that `A` and `B` are shared arrays, to illustrate one important difference between Fortran 77 and CRAFT. In this code, if interpreted as Fortran 77, `B(2,2)` is aliased with `A(6,7)`; but, if interpreted as CRAFT, it is aliased with `A(2,8)`. This is because, in CRAFT, aliasing between shared arrays takes place at the submatrix level whereas, in Fortran, a linear storage sequence is often assumed for parameter aliasing. Many real Fortran codes rely on such association rules.

We address this problem by changing the subscript expressions within the subroutine to conform to the CRAFT semantics. *Procedure cloning* is applied whenever the same routine is

```
real A(8,8)
call foo(A(1,7))
   ⋮
subroutine foo(B)
real B(4,4)
... B(2,2)
   ⋮
```

Figure 4.3: Sequence and storage association rules for the dummy array B

called with different submatrices as actual parameters. To illustrate this, suppose we are given a code in Figure 4.4, a generalized version of the code shown in Figure 4.3.

```
program main
real A(L1 : U1, ..., Ln : Un)
call foo(A(f1, ..., fn))
   ⋮
subroutine foo(B)
real B(L'1 : U'1, ..., L'm : U'm)
... B(g1, ..., gm)
   ⋮
```

Figure 4.4: Code before procedure cloning

Figure 4.5 shows the results after procedure cloning is applied to the code in Figure 4.4. Notice that in the resulting code the origin of the parameter array is passed to the subroutine rather than to the address of one of its elements. The actual and formal parameter arrays are forced to have the same size and shape. Let X be the offset of A($f_1, \ldots, f_n$) from the first address of A, and X' be that of B($g_1, \ldots, g_m$). Then, the array index h_k is defined as $h_k = X_k$ mod $N_k + L_k$ where $N_k = U_k + L_k + 1$, $X_1 = X + X'$ and $X_i = \lfloor X_{i-1}/N_{i-1} \rfloor$ for $i > 1$.

Our experiments show that cloning does not increase the size of the original programs by more than a factor of 2. This is the case because actual and formal parameters usually have the same shape, size, and dimension.

```
program main
real A(L1 : U1, ..., Ln : Un)
call foo_clone(A)
   ⋮
subroutine foo_clone(B)
real B(L1 : U1, ..., Ln : Un)
... B(h1, ..., hn)
   ⋮
```

Figure 4.5: Code after procedure cloning

4.3 Benchmark Programs

Table 4.3 briefly describes eleven programs that were used in our experiments. These programs come from either the SPEC95fp or the Perfect benchmarks. Detailed documents on these programs are available in [11, 62]. The table also shows the sequential execution times of each program on the Cray T3D.

We used all these programs in the preliminary experiments presented in Section 4.4. However, in the following experiments with advanced techniques, presented in Section 4.5, we used only six of them (BDNA, MDG, TRFD, SWIM, TFFT2, and TOMCATV) to produce the results. One reason for using only these six is that three of the other five programs not included (ARC2D,APPSP,FLO52) are not suitable for large-scale parallel machines like the T3D mainly because their data set sizes are too small, thereby limiting the loop-level parallelism that Polaris can utilize. One solution would be to increase the data sizes; however, this was not permitted in the experiments with these benchmarks. Another solution would be to exploit more parallelism at levels other than loops by enhancing Polaris. This was beyond the scope of this thesis. For the remaining two programs (HYDRO2D,SU2COR), we found that Polaris' current implementation of the techniques is not effective enough to obtain successful speedups. This is because we have not yet fully implemented all of the techniques presented in Chapter 3.

Program Name	Brief Description	Benchmark Source	Code Lines	Execution (sec)
ARC2D	Implicit finite-difference code for fluid flow	Perfect	4694	243
BDNA	Molecular dynamics simulation of biomolecules	Perfect	4887	82
FLO52	2-D analysis of transonic flow past an airfoil	Perfect	2368	54
MDG	Molecular dynamics model for water molecules	Perfect	1430	287
TRFD	Kernel for quantum mechanics calculations	Perfect	634	35
APPSP	Gaussian elimination system solver	SPEC	4439	1436
HYDRO2D	Navier Strokes solver to simulate galactical jets	SPEC	4289	1377
SU2COR	Quantum mechanics with Monte Carlo simulation	SPEC	2333	989
SWIM	Finite Difference solver of shallow water equations	SPEC	429	2377
TFFT2	Collection of FFT routines from NASA codes	SPEC	642	433
TOMCATV	generator of 2-D meshes around geometric domains	SPEC	190	1633

Table 4.3: Benchmark programs used in the experiments

In addition, we tried to include some other programs, not listed in Table 4.3, for our experminets. These failed because some of the problems discussed in Section 4.2 made Polaris unable to generate correct parallel programs for them.

4.4 Preliminary Experiments

Generating a correct CRAFT parallel program from an ordinary sequential program is a very complex task itself, even without considering the common optimization issues that exist for many other distributed memory machines, such as data distribution, data movement, and parallelism detection. This is primarily because of the restrictions imposed by CRAFT, as

discussed in Section 4.2. For this reason, in the beginning of our work, we focused on only the development of the basic compiler algorithms to generate correct SPMD parallel programs that may be validated when running on the T3D.

Therefore, the experiments using the parallel programs produced by the basic algorithms were preliminary because we handled few optimization issues in the code transformation based on these algorithms. As a consequence, the preliminary experimental results were unsatisfactory for most of the benchmarks. However, the performance analysis on the results helped identify several advanced techniques that would improve the original code quality. We will first discuss the preliminary experiments in this section, and continue our discussion with the experiments based on those advanced techniques in Section 4.5.

4.4.1 Code Transformation with Basic Techniques

Figure 4.6 shows the basic code transformation procedure to generate SPMD parallel code for the preliminary experiments. This procedure differs from that in Figure 1.4 primarily in that the data localization phase is not included here. Another difference is that the parallelism detection phase in this procedure does not use the Region Test to find parallel loops in a given program.

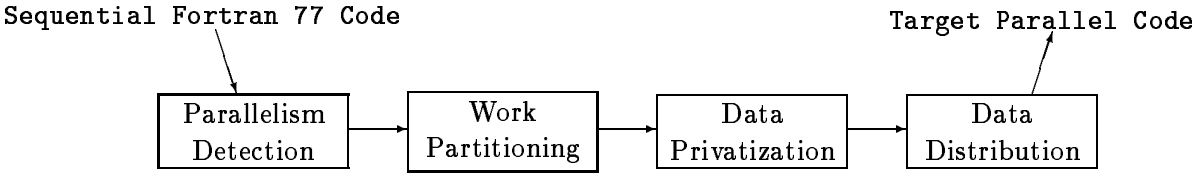

Figure 4.6: Basic code transformation for the T3D in Polaris

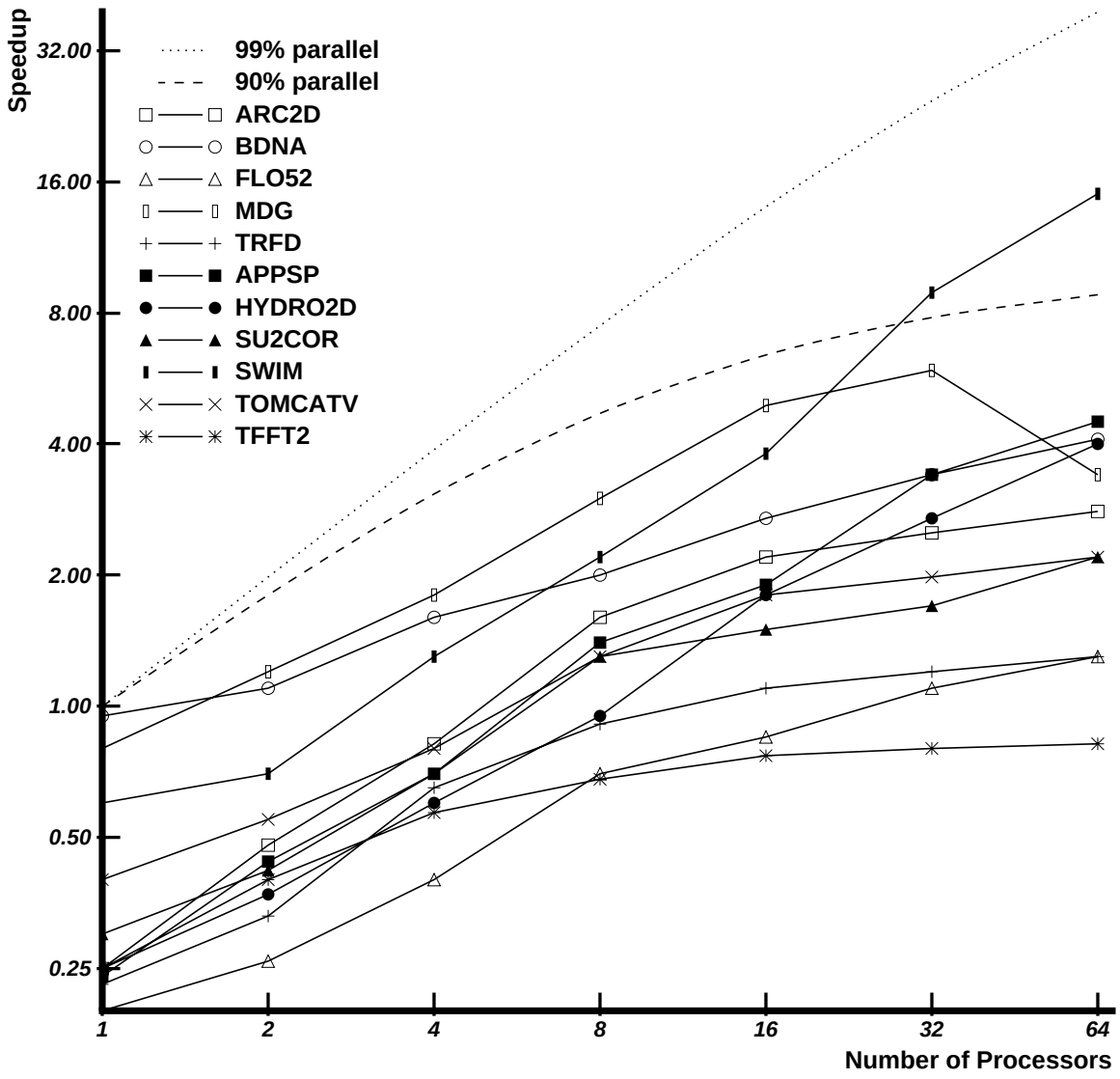

Figure 4.7: Results of the preliminary experiments with 11 benchmarks

4.4.2 Experimental Results

Figure 4.7 presents the speedups obtained by Polaris on the T3D for five programs from the Perfect benchmarks, and six programs from the SPEC95fp collection. We report speedups for processor numbers that are powers of two between 1 and 64. During the sequential execution of an original program, we measured the percentage of overall running time of the program for each loop, and added the percentages of each parallelizable loop to obtain the total sum. We call this total sum *parallel coverage* [15]. Two dotted lines in Figure 4.7 plot the ideal speedups for programs with parallel coverage of 99% and 90% respectively. The ideal speedup, S, is calculated using Amdahl's equality: $S = \frac{1}{\frac{c}{\mathcal{P}}+1-c}$, where $\mathcal{P}$ is the number of processors and c is the parallel coverage.

After all eleven analyzed programs were transformed by Polaris, their parallel coverage was between 90 and 99%. Therefore, the speedup curves for these programs should, under ideal conditions, lie between the two dotted lines. Real program speedups are much lower than the ideal for several reasons, including synchronization and scheduling overhead, communication costs, and usage of shared data that are not cached in the non-cache coherent machines like the T3D. These speedups should improve once we implement in Polaris optimizations to deal with these issues.

The effect of using shared data that are not cached, which we call the *cache bypassing penalty*, is reflected in the speedups for one PE, where the speedup is below one in all cases, as shown in Figure 4.7. The largest values are 0.9 for BDNA and 0.8 for MDG. This occurs because important arrays in both programs are privatized by the reduction recognition technique [59, 61] and, thus, have very few accesses to shared data. In contrast, FLO52 and TRFD have the lowest speedups for one PE because they repeatedly access large sections of shared arrays in the loops.

In addition to the cache bypassing penalty, as we increase the number of processors, other factors become significant, such as communication, the amount of parallelism, and the number of iterations of parallel loops. Such factors tend to decrease the efficiency as the number of processors grows. In this paper, *efficiency* is defined as the ratio between speedup and the ideal speedup made possible by the parallel coverage of the program. For example, the efficiency of TRFD, whose parallel coverage is 90%, goes from 0.23 to 0.13 as the number of processors grows from 1 to 64.

4.4.3 Performance Analysis

In this section, we analyze the behavior of three of the eleven programs presented in Figure 4.7: SWIM, MDG, and TRFD.

SWIM is a finite difference solver of the shallow water equation on a 512x512 grid. Its serial execution time on the T3D is 2378 seconds. SWIM performs most of its operations on fourteen 513x513 arrays. In the parallelized version, these arrays are expanded to 1024x1024 shared arrays because of the CRAFT restrictions discussed in Section 4.2. Fortunately, for 64 processors, the total amount of extra memory required is relatively small: 0.1M words per processor.

SWIM shows the best speedups in Figure 4.7. The main reason for these speedups is that Polaris parallelizes all the outermost loops, which results in a parallel coverage of almost 100%. Most parallel loops in SWIM are doubly-nested loops with 512x512 iterations. This is large enough to saturate 64 processors. Furthermore, each processor accesses different regions of the shared arrays in parallel loops, resulting in few memory access collisions during the execution of the parallel loops. Above all, our simple block

distribution policy happens to match computation distribution in SWIM, thus reducing a large amount of remote memory access overhead. As a result, we see that the speedup grows with the number of processors.

Despite the good characteristics of SWIM, we notice that the efficiency for a single processor is still 0.55. As mentioned earlier, this number mainly reflects the impact of the cache-bypassing penalty in the T3D. In other words, if caching is allowed for shared data, we would get a speedup of $1.8(= 1/0.55)$. In Figure 4.8, the upper limits of this predicted speedup line is plotted. This predicted line of course gives a glimpse of only the speedups we may achieve by optimizing the cache-bypassing penalty; therefore, the real speedups can surpass this prediced line by applying other techniques. However, it is clear that one of the major optimization efforts in SWIM should focus on increasing the fraction of cacheable data.

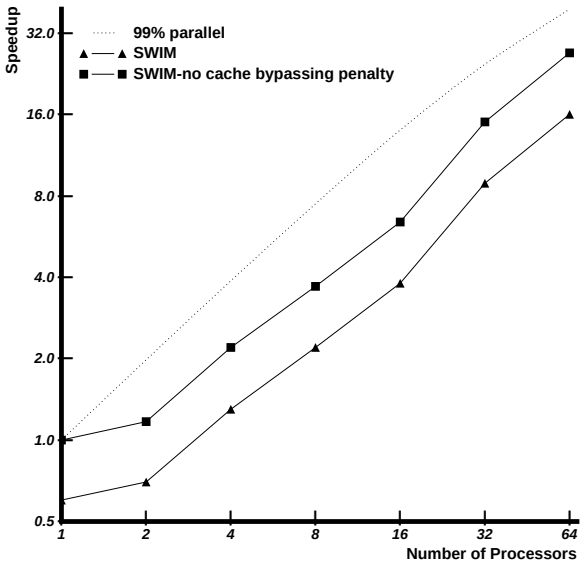

Figure 4.8: Speedup Analysis for SWIM

MDG is a molecular dynamics model of water. Its sequential execution time on the T3D is 330 seconds. The most important loop in MDG is `INTERF_do1000`, which accounts for 92% of the sequential execution time. This loop is parallelized because of several advanced techniques applied by Polaris, including inlining, array privatization, induction variable recognition, and histogram reduction recognition. The parallel version of `INTERF_do1000` has many privatized reduction arrays instead of shared arrays.

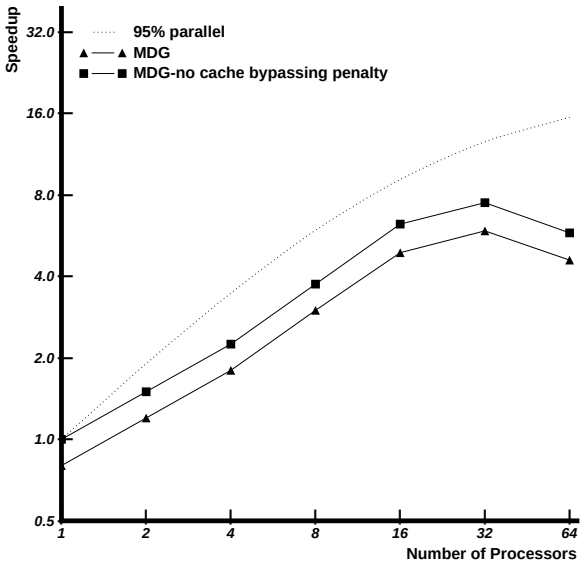

Figure 4.9: Speedup Analysis for MDG

Figure 4.9 shows that the speedup will not grow significantly even after eliminating all the cache bypassing penalty for the parallel loops. Thus, the cache bypassing penalty is not as influential in MDG as it is in SWIM.

The main cause of the drop in speedup is a doubly-nested sequential loop, `INTRAF_do1000`, which accounts for just 1% of sequential execution time. This loop accesses shared data and, when the number of processors grows, so does the execution time of this loop, as shown in Table 4.4. This shows that, in some cases, communication costs in a serial

loop grow much faster than the same costs in a parallel loop. Hence, we need to reduce these costs in sequential loops even when their execution time is relatively small. In fact, the loop `INTRAF_do1000` is parallel and, thus, the speedup will improve when Polaris is extended to parallelize this loop.

PEs	INTERF_do1000 (sec)	INTRAF_do1000 (sec)
2	260	4.6
4	140	5.8
8	76	8.4
16	41	14
32	21	26
64	11	59

Table 4.4: INTERF_do1000 and INTRAF_do1000 on the T3D

TRFD is a small kernel for quantum mechanics calculations. It has two major loops, `OLDA_do100` and `OLDA_do300`, both of which correspond to 90% of the sequential execution time. All the other parallel loops in TRFD take up less than 1% of the sequential execution time.

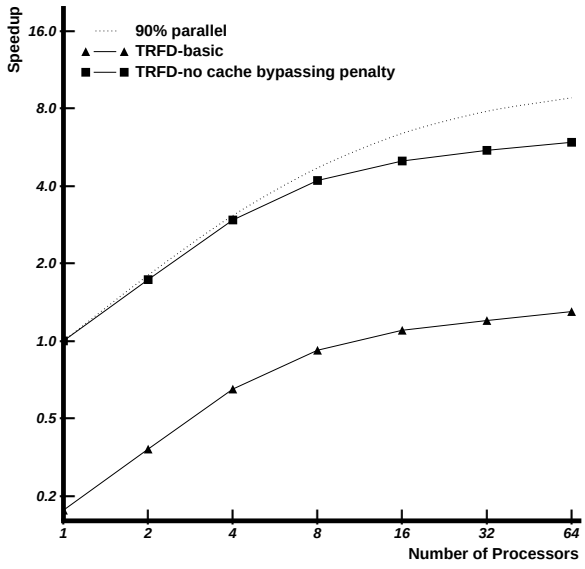

Figure 4.10: Speedup Analysis for TRFD

Judging from the case on a single processor in Figure 4.7, TRFD suffers from bypassing the cache for shared data more than SWIM and MDG. Removing such a penalty would drastically boost the performance by a factor of 4.6, as projected by the ideal speedup shown in Figure 4.10.

4.5 Experiments with Advanced Techniques

The analysis on the results of the preliminary experiments helps us to identify the main factors influencing parallel performance of these programs. These factors are:

1. the parallel coverage;
2. the parallel loop structure in a program;
3. the amount of shared data used in local computations; and,
4. the amount of data being moved in the system.

Based on this performance analysis, we could develop and implement the complete code transformation algorithms with various advanced techniques, which were discussed in Chapter 2 and 3. To increase the parallel coverage, we could use the Region Test, discussed in Section 3.2.1. Section 3.2.2 discussed the issues of the parallel loop structure. The Access Region and the shared data copying scheme are the techniques that handle the issues related to the last two factors.

Using the algorithms presented in Chapter 3, we transformed the original benchmark programs again to parallel form, and conducted the experiments by running them on the T3D. In this section, we discuss these experiments and performance analysis on them.

4.5.1 Experimental Results and Analysis

Figure 4.11 shows the experimental results. It is usually difficult to accurately quantify the effects of each compiler technique on the factors presented above because performance results from the combination of various techniques in most cases. For instance, the speedups in Table 2.1 of Section 3.2.1 are not possible without the Region Test's identification of the major loops in TFFT2 as parallel; yet, the Region Test could not detect those parallel loops without the Polaris privatizer's removal of anti and output dependences from the loops. The removal of these anti and output dependences is attributed to the PUT/GET primitives, and, the primitives, in turn, are essential to the shared data copying scheme.

Nevertheless, in Table 4.5, we try to single out the impact of each technique we used in the transformation. In the table, the rows correspond to the techniques and the columns to the benchmarks shown in Figure 4.11. If a technique is important for a program, the corresponding entry is marked.

	BDNA	MDG	TRFD	SWIM	TOMCATV	TFFT2
PD_1	×	×	×	×	×	
PD_2	×				×	
PD_3		×	×			×
LDP	×	×	×			×
LT					×	
DL	×	×	×	×	×	×

Table 4.5: Impact of the techniques on the benchmarks

PD_1, PD_2, and PD_3 stand for the parallelism detection techniques. We assume that PD_1 is applied first; then, PD_2 is applied to loops not parallelized by PD_1; and, finally, PD_3 is applied to the remaining serial loops. PD_1 includes the traditional dependence tests, such as the GCD test and the Range Test. The Polaris implementation of PD_1 cannot parallelize a

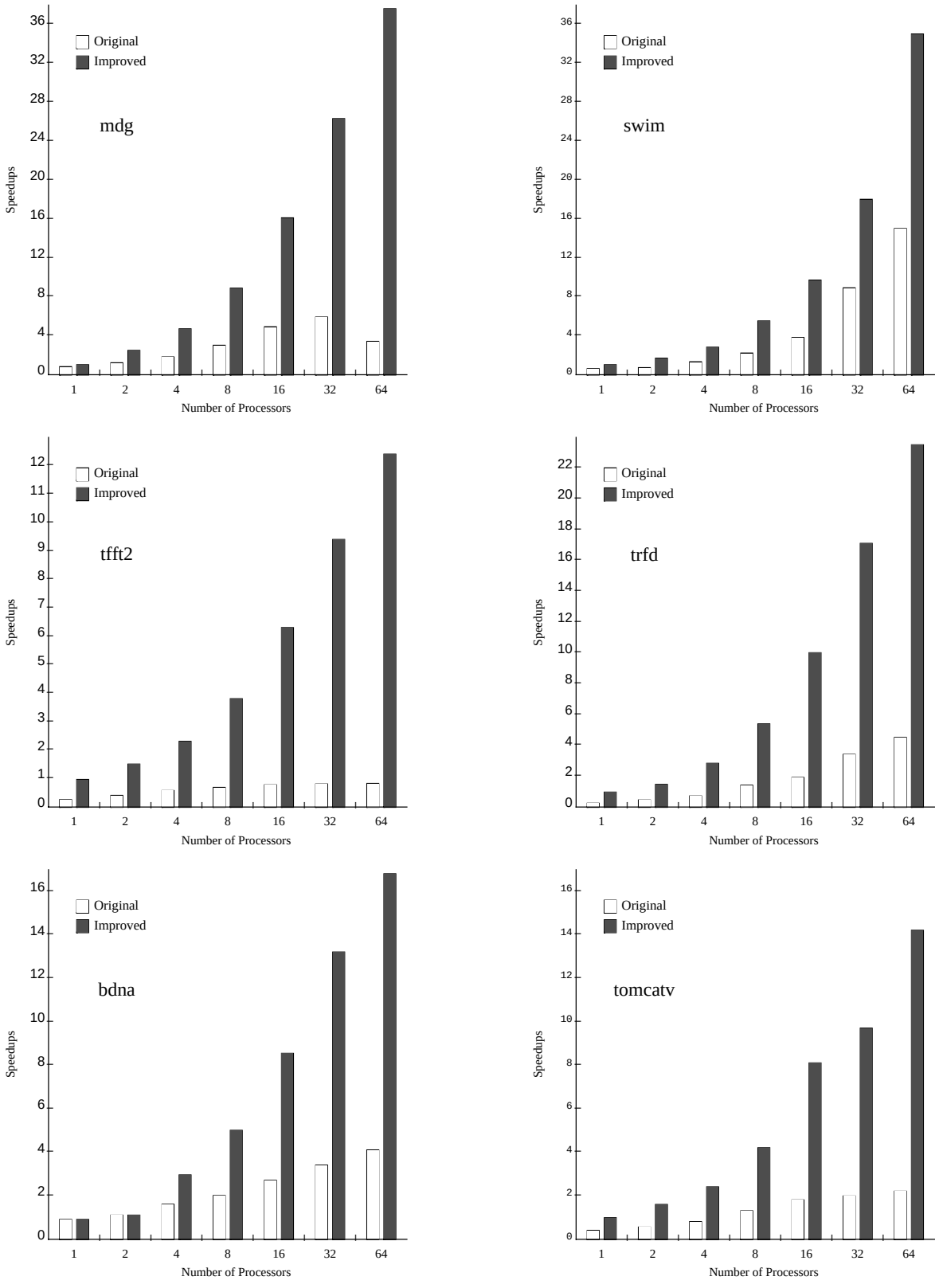

Figure 4.11: Improved parallel speedups for 6 benchmarks on the T3D

loop with I/O operations. But, some programs need to read large files, causing the I/O time to take a significant portion of the total execution time. Consequently, we developed a simple test to parallelize the I/O operations by making each processor read a small portion of the whole data file into shared memory space. We call this PD_2. PD_3 corresponds to the Region Test. As already mentioned in Section 3.2.1, the Region Test parallelizes the important loops in TFFT2 that PD_1 failed to do, and also enables us to deal with small loops in MDG and TRFD so that we can obtain good speedups on a larger number of processors. In the cases of BDNA, SWIM, and TOMCATV, PD_1 parallelizes most of the loops, thus leaving few loops to the Region Test.

LDP stands for the loop data privatization. In the experiments, we used only the loop privatization technique because we do not yet implement the procedure privatization module. According to our analysis, there are several benchmarks whose performance depends heavily on the procedure privatization technique, such as TOMCATV, SU2COR, and HYDRO2D, all from the SPEC95fp benchmarks. Specifically for SU2COR and HYDRO2D, we have found that we cannot obtain any outstanding speedups until we implement the procedure privatization module. The superlinear speedups in MDG are ascribed to LDP, which privatizes most of the important arrays in the program, as well as to PD_1 and PD_3, which parallelize almost all the loops in it.

LT stands for the conventional loop transformation techniques, including *loop fusion*, *loop distribution* and *loop interchange* [40, 73]. These techniques are useful in our approach because they help neighboring loop nests to reuse data stored in local memory. In TOMCATV, LT doubles the speedups.

processors	BDNA	MDG	TRFD	SWIM	TOMCATV	TFFT2
2	1.0	1.3	4.0	2.2	1.7	2.0
8	1.4	1.3	4.6	2.4	1.2	1.8
32	1.5	1.2	5.8	2.1	1.4	1.4

Table 4.6: Performance improvement due to data localization

The entries in Table 4.6 represent the value $\frac{speedup\ with\ DL}{speedup\ without\ DL}$, where DL stands for the data localization discussed in Section 3.2.4. The values in the table reflect the effectiveness of DL for each program. The table indicates that DL is effective across all the programs, especially for TRFD. In the next subsection, we will discuss in more detai the effectiveness of the data localization techniquel with three programs: SWIM, TOMCATV, and MDG.

4.5.2 Effectiveness of Data Localization

Figure 4.12 shows the speedups for three programs, SWIM, TOMCATV, and MDG, which were already shown in Figure 4.7. As mentioned earlier, these speedups were obtained after applying only the basic transformation techniques. As expected, the speedups are low in all cases due to the two factors discussed above. We can measure the impacts of the cache bypassing penalty on performance by looking at the speedups for one processor in the figure. As can be seen, the speedup is below one in all cases. We see TOMCATV is most heavily affected by the cache bypassing penalty. The dotted lines in the figure are the predicted speedups for each program, discussed in Section 4.4.3, that we might achieve after eliminating the penalty.

Figure 4.13 shows the speedups for these programs which were obtained after applying LD_1, the data localization technique, to the results in Figure 4.12. This figure illustrates how we improve the speedups to reach the predicted lines by using the data localization technique and the shared data copying scheme. As can be seen in the figure, Polaris now automatically gets

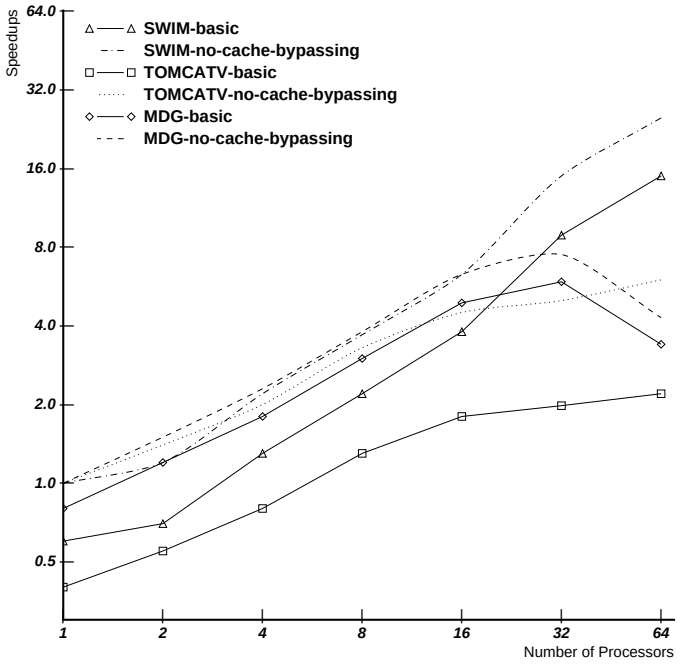

Figure 4.12: Before applying the data copying scheme

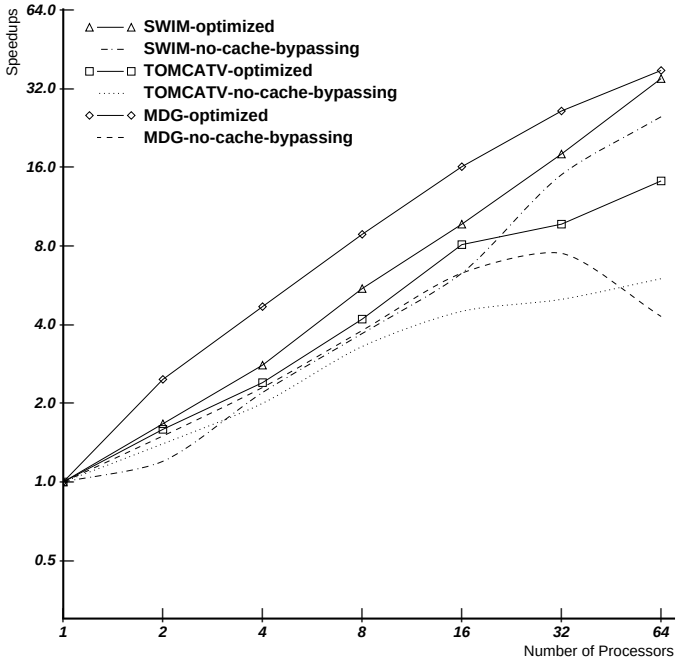

Figure 4.13: After applying the data copying scheme

rid of most of the cache bypassing penalty by using the shared data copying scheme. The figure also shows that the communication cost reduction through block data copying brings even better performance to the actual speedups than the predicted ones.

Although the greatest improvement in Figure 4.13 came from this data copying scheme, we also applied several other optimization techniques in the experiments. In MDG, we used the Region Test to parallelize several loops that the current Polaris frontend identifies as serial but that can be automatically parallelized. By doing so, we removed the performance drop in MDG for more than 32 processors. As discussed in the previous subsection, we applied loop fusion to remove most of the data copying overhead in TOMCATV. These additional loop transformation and parallelization techniques are very important for the data copying scheme.

CHAPTER 5

CONCLUSIONS AND FUTURE WORK

5.1 Achievements

In this dissertation, we have discussed our efforts to develop compiling techniques which automatically transform a sequential program into a shared-memory style parallel program for distributed memory multiprocessors.

In these efforts, we first tried to understand the fundamental issues related to code generation for distributed memory multiprocessors. To do so, we studied former work by others who have attempted to address these issues, which was discussed in Chapter 1. Also, we conducted the preliminary experiments, shown in Section 4.4, to comprehend the behaviors of real programs on distributed memory multiprocessors.

In this dissertation, we chose the Cray T3D, a commercial distributed memory machine, as our target machine for the experiments. The analysis of the experimental results helped us to

identify several performance factors that heavily affect the speedups on the machines. Furthermore, the analysis helped us to determine optimization techniques to handle these performance factors.

Based on these previous studies, we proposed a new approach for the code transformation which was presented in Chapters 2 and 3. Our code transformation capitalizes on the traditional compiler techniques already implemented in Polaris [13, 52]. In addition, it uses several newly developed techniques for a distributed memory architecture, such as the Access Region, the Region Test, procedure data privatization, and data localization.

The Access Region is a new way to represent the access patterns of arrays within a program which is more precise than traditional triplet notation. We have defined operations for manipulating this representation which are suitable for supporting the region processing needs of several compiler modules. We discussed using a general facility for processing this representation as a basis for privatization analysis, dependence analysis, communication generation, and interprocedural analysis within a parallelizing compiler.

The Region Test is a new dependence test built upon the Access Regions representation. We have shown that it combines the strengths of several existing dependence tests. We have argued that since the Access Region representation uses an abstraction of the program text, it can summarize the effect of arbitrary program sections and, therefore, can be used for interprocedural summarization and dependence testing as well as for loop-based parallelization. We presented a reformulation of the dependence problem in terms of regions, and showed how a combination of high-level summarization, strong symbolic region manipulation routines, and a precise representation can allow the parallelization of program sections that no other existing technique can. We also showed that a by-product of our reliance on symbolic manipulation is

the ability to extract the safe conditions for parallelization and to generate run-time dependence tests for some loops from real programs.

The procedure privatizer is a general version of the traditional loop privatizer. It can privatize variables used in any code segment within a program, including loops and subroutines, by finding parts of a data object written in the previous program section prior to being used in the current section.

The core technique for data localization is the shared data copying scheme. The scheme removed most of the communication needed for the access to shared data, thereby reducing communication overhead. The shared data copying scheme is a main beneficiary of the Access Region representation because accurate data access region analysis is the most vital component to the success of the scheme.

Combining all these new techniques with the traditional complier techniques, we parallelized sequential codes from the Perfect and SPEC95fp benchmarks, and experimented with the parallelized codes on the T3D. The experimental results, shown in Section 4.5, revealed that our approach is reasonably effective on real applications by addressing most of the performance factors we identified from the preliminary experiments.

5.2 Additional Techniques to Further Improve Performance

So far, we have presented various compiler techniques in Polaris for code transformation on distributed-memory multiprocessors and the experimental evidence to reveal their effectiveness. However, our work also revealed that there is much room for improvement. In particular, our current implementation has no advanced techniques to handle the data distribution issues arisen in the code generation for distributed-memory multiprocessors. In this section, we present our

recent study on some strategies to further improve our code transformation, as reported in a paper by the author and others [47].

5.2.1 Redundant PUT/GET Elimination

One important optimization is the reduction of communication overhead in the shared data copying scheme, which is achieved by reducing the number of PUT/GET operations. The communication optimization algorithm currently implemented in Polaris is relatively simple and, as a consequence, the communication overhead is sometimes unnecessarily large and scalability is hindered.

One way to reduce the communication overhead in our approach is to reduce the number of GET/PUT operations. For this purpose, we have found that we still need additional cross-loop analysis. Although our data copying scheme greatly reduces the overall communication overhead by aggregating data to be copied in the experiment, cross-loop analysis allows us to eliminate the redundant data copying repeatedly performed between loop nests. One technique we have already used for this purpose is conventional loop transformation, such as *loop fusion*, *loop distribution*, and *loop interchange* [40, 73].

Another, yet more general, technique we consider now is *redundant PUT/GET elimination*. We illustrate this technique using the example in Figure 5.1. In the example, for the shared array `X`, our current scheme generates two private arrays, `x0` and `x1`, and two GET operations because `X` is accessed in two different loop nests. However, using access region analysis, we can identify that the region of `X` read by the `I`-loop completely covers the region by the following the `J`-loop and, thus, `x1` is redundant. In this case, we can eliminate `x1` and replace `x1((J-L)/2+1)` with `x0(J-L+1)`; as a result, only one GET operation is needed.

```
cdir$ private x0, x1, ···
cdir$ shared X(BLOCK), ···
       ···
      call polaris_get(X(L),x0(1), 1, 1, U-L+1)
      do I = L, U, 1
         ··· = x0(I-L+1)
      enddo
       ···
      call polaris_get(X(L),x1(1), 2, 1, (U-L)/2+1)
      do J = L, U, 2
         ··· = x1((J-L)/2+1)
      enddo
       ···
```

Figure 5.1: Code example for redundant PUT/GET elimination

We have found that these redundant `polaris_put` and `polaris_get` calls occur fairly frequently in the codes automatically generated by Polaris. The elimination improved the execution times and the scalability of studied programs.

5.2.2 Access Sensitive Data Distribution

Data distribution is another important issue in the code generation for distributed memory multiprocessors. As discussed in Chapter 3, the current Polaris data distribution strategy is *access insensitive* in that it simply chooses `BLOCK` distribution regardless of the data access patterns in a program. This naive distribution policy facilitates the shared data copying scheme to some extent, as discussed in Section 3.2.4. For very regular programs, however, it has often been suggested that more sophisticated data distribution policies improve performance [6, 8, 9, 19, 41, 56]. Therefore, we need to apply *access sensitive* data distribution strategies mainly to those regular programs. These strategies can be implemented in the data distribution stages of the transformation procedure discussed in Figure 1.4 to complement data privatization and localization techniques in Polaris.

To estimate the importance of the access sensitive data distribution in our current transformation framework, four benchmarks, SWIM, MDG, TRFD and MDG, were recently hand-optimized with these data distribution issues taken into account. That is, in the hand-optimized versions, appropriate data distribution policies for shared arrays were carefully chosen based on the access pattern analysis of the programs.

5.2.3 Manual Experiments with Additional Techniques

Figure 5.2 shows that the application of the techniques produced almost perfect speedups for the four programs on the Cray T3D. The speedups of the automatic versions were obtained from Figure 4.11. The speedups of the manual versions were obtained from the experiments using the hand-optimized parallel codes.

The figure shows that, in TRFD, access sensitive manual data distribution strategies substantially minimize the communication overhead resulting from the copying operations. In MDG, the manual version of MDG shows slightly better scalability in the speedup curves on more than 32 processors. This is made possible by the techniques for reduction of communication overhead due to manually chosen access sensitive data distribution strategies.

However, we were unable to develop a manual version for MDG based on conventional data distribution techniques that outperformed the automatic version. One reason for this is that the Polaris privatization module privatizes only the most important arrays, thereby eliminating the need to distribute data. A second reason is that MDG has irregular data access patterns; that is, it is composed of several different subroutines that have conflicting data distribution requirements for fast execution. Thus, a single static distribution cannot be determined to satisfy all the data access patterns in the program. Also, there are frequent requirements for

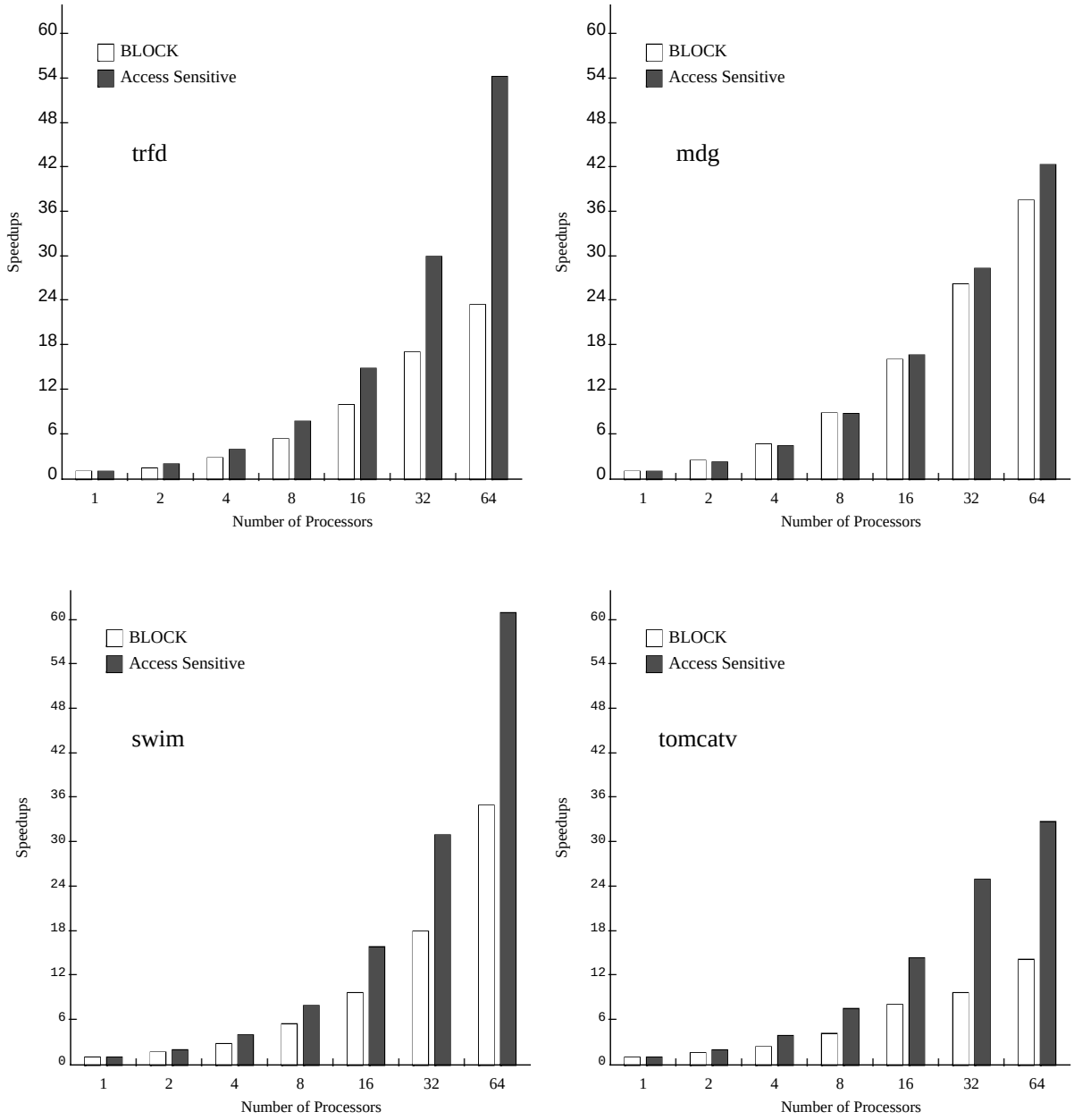

Figure 5.2: Impact of data distribution strategies on the T3D

reduction operations (i.e., to get positions for all atoms and to evaluate a processor's total contribution in the final force) which results in the need for expensive global communication. This global information sharing makes it more difficult to find good data distribution for MDG. As a result, we conclude that, in MDG, the access sensitive data distribution strategies do not help much.

The experiment with MDG supports the argument in Section 3.2.4 that the shared data copying scheme can be a better solution than sophisticated data distribution algorithms for the codes that need frequent global data sharing or that contain irregular data access patterns.

5.3 Conclusions

There are several requirements for the powerful parallelizing compilers that automatically transform sequential codes into parallel form for a diversity of distributed multiprocessor systems. One is that the cases occurring in real-world applications usually vary too much to be handled by any single or a few compiler techniques. As a matter of fact, the compiler that attempts to be effective for many real problems must be equipped with a variety of powerful techniques. However, exhaustedly applying a series of those techniques to a program would not be advantageous because it would take the compiler too long time to generate parallel code. Thus, the techniques implemented in the compiler are also required to be efficiently organized. All these requirements for powerful parallelizing compilers must be based on a thorough and careful study of various compiler techniques and the characteristics of a broad range of real applications.

The work presented in this dissertation is based on the Polaris compiler which quite successfully satisfies these requirements. Its powerful frontend modules, consisting of parallelism detection and elimination techniques, have been built upon more than a decade's experience of

automatic parallelization of Fortran 77 programs. The effectiveness and applicability of these techniques have been strongly proven through experiments on several commercial UMA shared memory multiprocessors using numerous well-recognized benchmark programs.

Another factor making the Polaris compiler effective for versatile programs is the design of the Polaris backend modules which follows a careful study of the target machines. This machine-dependent design has resulted in many unique and effective code generation techniques for each target machine.

In the work presented in this dissertation, we have followed the same philosophy as we have for other types of machines. To obtain excellent multiprocessor speedups, we utilized a diversity of existing compiler techniques already implemented in Polaris. Also, we developed the T3D-specific code generation backend modules, profiting from the fact that the T3D supports fast barrier and PUT/GET operations based on low latency network. The experimental results reported in this dissertation proved the effectiveness of our approach for at least the T3D.

To extend our transformation techniques to other classes of machines with remote memory latency higher than the T3D, in Section 5.2, we analyzed the performance shown in Figure 4.11 to identify the importance of the data distribution issue in our code transformation platform. The manual evaluation has convinced us that better data distribution strategies will help us to extend the Polaris compiler to parallelize a broader range of programs for other classes of distributed memory systems.

BIBLIOGRAPHY

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.

[2] American National Standards Institute, ANSI X3.9-1978 ISO 1539-1980. *Programming Language FORTRAN*, 1978.

[3] C. Amza, A.L. Cox, S. Dwarkadas, P. Keleher, H. Lu, R. Rajamony, W. Yu, and W. Zwaenepoel. TreadMarks: Shared Memory Computing on Networks of Workstations. *IEEE Computer*, 29(2):18–28, Feburary 1996.

[4] J. Anderson and M. Lam. Global Optimizations for Parallelism and Locality on Scalable Parallel Machines. *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, pages 112–125, June 1993.

[5] R. Arpaci, D. Culler, A. Krishnamurthy, S. Steinberg, and K. Yelick. Empirical Evaluation of the CRAY-T3D: A Compiler Perspective. *International Symposium on Computer Architecture*, pages 320–331, June 1995.

[6] E. Ayguade, J. Garcia, M. Girones, J. Labarta, J. Torres, and M. Valero. Detecting and Using Affinity in an Automatic Data Distribution Tool. In *Lecture Notes in Computer Science*, pages 61–75. Springer Verlag, New York, New York, August 1994.

[7] V. Balasundaram and K. Kennedy. A Techniques for Summarizing Data Access and its Use in Parallelism Enhancing Transformations. *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 1989.

[8] R. Barua, D. Kranz, and A. Agarwal. Communication-Minimal Partitioning of Parallel Loops and Data Arrays for Cache-Coherent Distributed-Memory Multiprocessors. In *Lecture Notes in Computer Science*. Springer Verlag, New York, New York, August 1996.

[9] D. Bau, I. Kodukula, V. Kotlyar, K. Pingali, and P. Stodghill. Solving Alignment Using Elementary Linear Algebra. In *Lecture Notes in Computer Science*, pages 61–75. Springer Verlag, New York, New York, August 1994.

[10] D. Bernstein, M. Breternitz, A. Gheith, and B. Mendelson. Solutions and Debugging for Data Consistency in Multiprocessors with Noncoherent Caches. *International Journal of Parallel Programming*, 23(1):83–103, 1995.

[11] M. Berry, D. Chen, P. Koss, D. Kuck, L. Pointer, S. Lo, Y. Pang, R. Roloff, A. Sameh, E. Clementi, S. Chin, D. Schneider, G. Fox, P. Messina, D. Walker, C. Hsiung,

J. Schwarzmeier, K. Lue, S. Orszag, F. Seidl, O. Johnson, G. Swanson, R. Goodrum, and J. Martin. The Perfect Club Benchmarks: Effective Performance Evalution of Supercomputers. *International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[12] W. Blume. *Symbolic Analysis Techniques for Effective Automatic Parallelization.* PhD thesis, Univ. of Illinois at Urbana-Champaign, Dept. of Computer Science, June 1995.

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

[14] W. Blume and R. Eigenmann. The Range Test: A Dependence Test for Symbolic, Nonlinear Expressions. *Proceedings of Supercomputing '94*, pages 528–537, November 1994.

[15] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Petersen, W. Pottenger, L. Rauchwerger, P. Tu, and S. Weatherford. Polaris: The Next Generation in Parallelizing Compilers. *Proceedings of 7th Workshop on Language and Compilers for Parallel Computing*, pages 10.1–10.18, August 1994.

[16] S. Bokhari. Communuication Overhead on Intel Paragon, IBM SP2 and Meiko CS-2. Technical Report 28, Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, 1995.

[17] M. Burke and R. Cytron. Interprocedural Dedependence Analysis and Parallelization. *Proceedings of the SIGPLAN Symposium on Compiler Construction*, pages 162–175, July 1986.

[18] N. Carriero and D. Gelernter. Case Studies in Asynchronous Data Parallelism. *International Journal of Parallel Programming*, 22(2):129–149, 1994.

[19] B. Chapman, P. Mehrota, H. Moritsch, and H. Zima. Dynamic Data Distributions in Vienna Fortran. *Proceedings of Supercomputing '93*, pages 284–293, November 1995.

[20] S. Chatterjee, J. Gilbert, F. Long, R. Schreiber, and S. Teng. Generating Local Address and Communication Sets for Data-Parallel Programs. *Journal of Parallel and Distributed Computing*, 26(1):72–84, April 1995.

[21] K. Cooper, M. Hall, and L. Torczon. An experiment with inline substitution. Technical Report COMP TR-90-128, Department of Computer Science, Rice University, August 1990.

[22] Cray Research Inc. *CRAY MPP Fortran Reference Manual*, 1993.

[23] Cray Research Inc. *The CRAY T3D System Architecture*, 1993.

[24] Cray Research Inc. *SHMEM Technical Note for Fortran*, 1994.

[25] Cray Research Inc. *The CRAY T3E-900 Scalable Parallel Processing System*, 1996.

[26] B. Creusillet and F. Irigoin. Exact vs. Approximate Array Region Analyses. In *Lecture Notes in Computer Science*. Springer Verlag, New York, New York, August 1996.

[27] D. Culler, A. Dusseau, S. Goldstein, A. Krishnamurthy, S. Lumetta, T. Eicken, and K. Yelick. Parallel Programming in Split-C. *Proceedings of Supercomputing '93*, pages 262–273, November 1993.

[28] R. Cytron and J. Ferrante. What's in a Name? *Proceedings of the 1987 International Conference on Parallel Processing*, pages 19–27, August 1987.

[29] K. Faigin. The Polaris Internal Representation. Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., 1994.

[30] G. Goff, K. Kennedy, and C. Tseng. Practical Dependence Testing. In *Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation*, pages 15–29, June 1991.

[31] J. Grout. Inline Expansion for the Polaris Research Compiler. Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., May 1995.

[32] K. Hayashi, T. Doi, T. Horie, Y. Koyanagi, O. Shiraki, N. Imamura, T. Shimizu, H. Ishihara, and T. Shindo. AP1000+: Architectural Support of PUT/GET Interface for Parallelizing Compiler. *Proceedings of 6th International Conference on Architechtural Support for Programming Language and Operating Systems*, pages 196–207, October 1994.

[33] Hewlett-Packard CONVEX division. *HP-CONVEX Exemplar S-Class and X-Class Systems Overview*, 1996.

[34] W. Hillis. *The Connection Machine*. The MIT Press, MA, 1985.

[35] S. Hiranandani, Kennedy K, and C. Tseng. Evaluating Compiler Optimizations for Fortran D. *Journal of Parallel and Distributed Computing*, pages 27–45, 1994.

[36] J. Hoefling, Y. Paek, and D. Padua. Region-based Parallelization Using the Region Test. Technical Report 1514, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing R&D, 1996.

[37] IBM Corp. *RS/6000 Scalable POWERparallel Systems(SP)*, 1996.

[38] W. Jalby and U. Meier. Optimizing Matrix Operations on a Parallel Multiprocessor with a Memory Hierarchy. *Proceedings of the 1986 International Conference on Parallel Processing, St. Charles, IL*, pages 429–432, August 1986.

[39] V. Karamcheti and A. Chien. A Comparison of Architectural Support for Messaging in the TMC CM-5 and the Cray T3D. *International Symposium on Computer Architecture*, June 1995.

[40] K. Kennedy and K. McKinley. Maximizing Loop Parallelism and Improving Data Locality via Loop Fusion and Distribution. *Proceedings of 6th Workshop on Language and Compilers for Parallel Computing*, pages 301–320, August 1993.

[41] K.Kennendy and U. Kremer. Automatic Data Layout for High Performance Fortran. *Proceedings of Supercomputing '95*, November 1995.

[42] D. Kuck. What Do Users of Parallel Computer Systems Really Need? *International Journal of Parallel Programming*, 22(1):99–127, 1996.

[43] Kuck and Associate Inc. *The KAP™ Preprocessor*, 1995.

[44] R. Marcelin. *Message Passing on the CRAY T3D.* Massively Parallel Computing Group, NERSC, 1995.

[45] Message Passing Interface Forum. *MPI-2: Extensions to the Message-Passing Interface*, January 12, 1996.

[46] J. Moreira and C. Polychronopoulos. On the Implementation and Effectiveness of Autoscheduling. Technical Report 1372, Univ of Illinois at Urbana-Champaign, Cntr for Supercomputing Res & Dev, June 1994.

[47] A. Navarro, Y. Paek, E. Zapata, and D. Padua. Compiler Techniques for Effective Communication on Distributed-Memory Multiprocessors. *Proceedings of the 1997 International Conference on Parallel Processing*, August 1997.

[48] J. Nielocha, R. Harrison, and R. Littlefield. Global Arrays: A Portable Shared-Memory Programming Model for Distributed Memory Computers. *Proceedings of Supercomputing '94*, pages 340–349, November 1994.

[49] Oak Ridge National Laboratory, Oak Ridge, TN. *PVM 3 User's Guide and Reference Manual.*

[50] W. Oed. *The Cray Reseach Massively Parallel Processor System CRAY T3D.* Cray Research Inc., 1993.

[51] D. Padua. Polaris: An Optimizing Compiler for Parallel Workstations and Scalable Multiprocessors. Technical Report 1475, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., January 1996.

[52] D. Padua, R. Eigenmann, and J. Hoeflinger. Automatic Program Restructuring for Parallel Computing and the Polaris Fortran Translator. *Proceedings of the Seventh SIAM Conference on Parallel Processing for Scientific Computing, San Francisco, CA*, pages 647–649, February 1995.

[53] Y. Paek, J. Hoefling, and D. Padua. Access Regions: Toward a Powerful Parallelizing Compiler. Technical Report 1508, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing R&D, 1996.

[54] Y. Paek and D. Padua. Compiling for Scalable Multiprocessors with Polaris. In *Parallel Processing Letters*. World Scientific Publishing, UK, 1997.

[55] Y. Paek and D. Padua. Automatic Parallelization for Non-cache Coherent Multiprocessors. In *Lecture Notes in Computer Science*. Springer Verlag, New York, New York, August 1996.

[56] D. Palermo and P. Banerjee. Automatic Selection of Dynamic Data Partitioning Schemes for Distributed-Memory Multicomputers. In *Lecture Notes in Computer Science*, pages 392–406. Springer Verlag, New York, New York, August 1995.

[57] C. Polychronopoulos. *Parallel Programming and Compilers.* Academic Publishers, MA, 1988.

[58] C. Polychronopoulos, M. Girkar, M. Haghighat, C. Lee, B Leung, and D. Schouten. The structure of parafrase-2: An advanced parallelizing compiler for c and fortran. In *Lecture Notes in Computer Science.* Springer Verlag, New York, New York, August 1990.

[59] W. Pottenger. Induction Variable Substitution and Reduction Recognition in the Polaris Parallelizing Compiler. Master's thesis, Univ of Illinois at Urbana-Champaign, Cntr for Supercomputing Res & Dev, December 1994.

[60] W. Pugh. A Practical Algorithm for Exact Array Dependence Analysis. *Communications of the ACM*, 35(8), August 1992.

[61] L. Rauchwerger and D. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation*, June 1995.

[62] J. Reilly. SPEC95 Products and Benchmarks. *SPEC Newsletter*, September 1995.

[63] J. Saitz, R. Mirchandaney, and K Crowley. Run-time Parallelization and Scheduling of Loops. *IEEE Transactions on Computers*, May 1991.

[64] R. Schreiber. High Performance Fortran 2.0. *Workshop on Challenges in Compiling for Scalable Parallel Systems* in conjuntion with *IEEE 8th Symposium of Parallel and Distributed Processing*, October 1996.

[65] SGI/Cray Research Inc. *Origin^{TM} and Onyx2^{TM} Programmer's Reference Manual*, 1996.

[66] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An Empirical Study of Fortran Programs for Parallelizing Compilers. *IEEE Transaction on Parallel and Distributed Systems*, 1(3):350–364, July 1990.

[67] P. Tang. Exact Side Effects for Interprocedural Dependence Analysis. *Communications of the ACM*, 35(8):102–114, August 1992.

[68] C. Tseng. *An Optimizing Fortran D Compiler for MIMD Distributed-Memory Machines.* PhD thesis, Rice University, January 1993.

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

[70] P. Tu and D. Padua. Gated SSA-Based Demand-Driven Symbolic Analysis for Parallelizing Compilers. *Proceedings of the 9th ACM International Conference on Supercomputing*, pages 414–423, July 1995.

[71] S. Wallace. Teraflops into Laptops. *IEEE Parallel & Distributed Technologies*, pages 8–9, Fall 1994.

[72] M. Wolf and M. Lam. A data locality optimizing algorithm. In *SIGPLAN NOTICES: Proceedings of the ACM SIGPLAN 91 Conference on Programming Language Design and Implementation, Toronto, Ontario, Canada, June 26-28*, pages 30–44. ACM Press, 1991.

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

[74] H. Zima. *Supercompilers for Parallel and Vector Computers.* ACM Press, 1992.

VITA

Yunheung Paek was born on July 25th, 1965 in Seoul, Korea. In 1984, he entered Seoul National University, and earned a Bachelor in Science degree in Computer Engineering in 1988 and a Master in Science degree in Computer Engineering in 1990. While he was a graduate student in Korea, he joined several research projects affiliated with the Korean government and industry. In August 1990, he joined the Korean Army. After his honorable discharge from service as a lieutenant in 1991, he started his graduate studies at the University of Illinois at Urbana-Champaign. From August 1991 to July 1994, he has been supported by the scholarship from the Korean Ministry of Education. Since August 1994, he has been a graduate research assistant at the Center of Supercomputing Research and Development. After graduation, he will continue to work at the center as a researcher for several months, and join New Jersey Institute of Technology as a tenure-track assistant professor from the Fall semester of 1997.