# SUIF EXPLORER: AN INTERACTIVE AND INTERPROCEDURAL PARALLELIZER

**Shih-Wei Liao**

**Technical Report No. CSL-TR-00-807**

**August 2000**

# SUIF EXPLORER: AN INTERACTIVE AND INTERPROCEDURAL PARALLELIZER

**Shih-Wei Liao**

**Technical Report: CSL-TR-00-807**

August 2000

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science, Stanford University
Stanford, California 94305 - 9040
pubs@shasta.stanford.edu

## Abstract

Shared-memory multiprocessors that use the latest microprocessors are becoming widely used both as compute servers and as desktop computers. But the difficulty in developing parallel software is a major obstacle to the effective use of the multiprocessors to solve a single task. To increase the productivity of multiprocessor programmers, we developed an interactive interprocedural parallelizer called SUIF Explorer. Our experience with SUIF Explorer also helps to identify missing interprocedural analyses that can significantly improve an automatic parallelizer.

As a parallel programming tool, the Explorer actively guides the programmers in the parallelization process using a set of advanced static and dynamic analyses and visualization techniques. Our interprocedural program analyses provide high-quality information that restricts the need for user assistance. The Explorer is also the first tool to apply slicing analysis to aid the programmer in uncovering program properties for interactive parallelization. These static and dynamic analyses minimize the number of lines of code requiring programmer assistance to produce parallel codes for real-world applications.

As a tool for finding missing compiler techniques, SUIF Explorer helps the compiler researchers design the next-generation parallelizer. Our experience with the Explorer shows that interprocedural array liveness analysis is an enabler of several important optimizations, such as privatization and array contraction. We developed and evaluated an efficient context-sensitive and flow-sensitive interprocedural array liveness algorithm and integrated it into the parallelizer. We use the liveness information to enable contraction of arrays that are not live at loop exits, which results in a smaller memory footprint and better cache utilization. The resulting codes run faster on both uni- and multi- processors.

Another key interprocedural analysis which we developed and evaluated is the array reduction analysis. Our reduction algorithm extends beyond previous approaches in its ability to locate reductions to array regions, even in the presence of arbitrarily complex data dependences. To exploit the multiprocessors effectively, the algorithm can locate *interprocedural reductions*, reduction operations that span multiple procedures. In summary, we successfully apply the Explorer to help the user develop parallel codes effectively and to help the compiler researcher develop the next-generation parallelizer.

**Key Words and Phrases:** compiler optimization, interprocedural parallelization, programming environment, slicing, visualization, array liveness, reduction, array contraction

To my parents, brothers, and sisters

# Acknowledgments

The satisfaction of completing this Ph.D. research belongs to all who journeyed with me. I would like to first thank my advisor Monica Lam — whose intellectual vigor, insight, enthusiasm, and encouragement made critical contribution to this thesis. Furthermore, she is also a great friend with valuable advice in other areas of life. Bill Reynolds graciously agreed to be my orals chairman. John Hennessy and Mendel Rosenblum provided invaluable comments as members of my thesis and orals committee. I am very privileged to receive guidance from each of them.

I deeply appreciate the privilege to work closely with many extraordinary individuals on various projects. The interprocedural parallelizer project was a joint effort with Mary Hall, Saman Amarasingh, and Brian Murphy. The SUIF Explorer project was done with Amer Diwan, Robert Bosch, and Anwar Ghuloum. Working with Jeff Gibson on the FlashPoint project has also been a great pleasure. I also worked with Jeff Oplinger and David Heine on the topic of speculative thread-level parallelism earlier. Jeff is a fun person to share the office with and David always plays important roles in many projects. Finally, Jennifer Anderson, Amy Lim, and Evan Torrie have greatly enriched my research experience in the area of memory performance. I benefited enormously from the knowledge, expertise, and commitment of all of my colleagues above. They are awesome friends as well.

I enjoyed and learned from the remaining people in the SUIF compiler team, especially Gerald Aigner, Gerald Cheong, Robert French, Walter Hsueh, Vladimir Livshits, Dror Maydan, Jason Nieh, Martin Rinard, Costa Sapuntzakis, Dan Scales, Brian Schmidt, Mike Smith, Steve Tjiang, Chau-Wen Tseng, Chris Unkel, Hansel Wan, Bob Wilson, and Chris

Wilson. Charlie Orgish and Thoi Nguyen are instrumental in supporting the computing environment. Deborah Harber, Jeff Oplinger, Dwight Sunada, and Clifton Watt proofread my dissertation and provided many invaluable comments.

My prayer partners always help straighten out my priorities whenever my work has become more important than the people around me. These fellow students are David Banjerdpongchai, Gary Chan, Ken Chau, Eddy Chee, Benjamin Chen, Alvin Chow, Brian Decker, Eric Han, Ian Hsu, Pete Hwang, Wee Eng Koh, Arthur Lee, Paksing Lee, TJ Leong, Erik Lin, Jon Liu, Lai-Chee Man, Andreas Tobler, Clifton Watt, Andrew Wee, Edwin Wong, Yongjian Wu, Liya Yu. The hours we spent praying have been my best time at Stanford. They understand my innermost joy and struggles so I am deeply grateful to them. Friends in Chinese Christian Fellowship at Stanford have also supported me.

I thank my spiritual mentors: Professor Paul Hensleigh, Pastor Steve Lawry and John Chan. Their advice often reminded me that there are greater things in life than a degree. Special thanks to John for teaching me Greek weekly. My biggest thanks on earth surely go to my parents, David and Shirley, my sister Lois, and my brother Sam. My close-knit family cares for me unconditionally, even though they did not know what they were getting into when I decided to pursue a Ph.D. degree.

Last but not least, I want to thank Jesus for loving me and giving meaning and purpose to my life. He teaches me to maintain a balanced perspective and restores my hope whenever it has faded. It is he who enables me to thank God by enjoying him forever. He is my hero and most faithful friend.

# Table of Contents

# List of Figures

# 1 Introduction

As modern microprocessors become complex and the marginal performance improvement diminishes, using multiprocessors is a cost-effective way to improve performance beyond that achieved by a single processor. Shared-memory multiprocessors built with latest commodity microprocessors are becoming widely used both as compute servers and as desktop computers. To deliver supercomputer-like performance to the end users, the multiple processors need to be used together to accelerate the execution of *single* applications. However, the difficulty of developing parallel software is a major obstacle to the widespread use of parallel machines[59]. Developing high-quality parallel programs is difficult as the gain-to-pain ratio in parallel programming is very low. The goal of this thesis work is to develop an effective parallel programming tool with advanced program analyses to increase the gain-to-pain ratio in parallel programming for scientific applications.

One approach to increase the productivity of parallel programming is to use an automatic parallelizer[18][19][51][62][66]. A state-of-the-art parallelizer can automatically locate large outer parallel loops that span many procedures and hundreds of lines of code[51][52]. This *coarse-grain parallelism* comes from regions of code with independent computations, where a significant amount of work can be performed without any synchronization. Extracting coarse-grain parallelism is more important on multiprocessors than on vector machines because the former incur higher synchronization and communication costs[5]. Although an advanced parallelizer can locate coarse-grain parallelism, it is *fragile* due to the following reasons:

- Even a compiler that included every conceivable parallelization technique would be inadequate, because compilers are fundamentally limited by the sequential semantics originally coded into a program. It sometimes requires application-specific knowledge to modify the algorithm to make it parallelizable.

- Detecting coarse-grain parallelism is much more complicated than finding inner loop parallelism. It has been shown that a comprehensive set of high-level interprocedural analyses and optimizations are needed to detect coarse-grain parallelism[53]. Thus, the parallelizer for multiprocessors is more likely to miss some analyses than its counterpart for the vector machines, especially since a compiler typically aims primarily at common optimizations.

- The likelihood of finding a dependence in a large loop that spans hundreds of lines of code is higher than that in a fine-grain loop. A single dependence in a large but otherwise parallel loop can ruin the program's parallel performance.

To go beyond automatic parallelization, we developed a parallel programming tool called SUIF Explorer[78]. The Explorer makes the analysis results available to the programmer using state-of-the-art visualization[22]. Our automatic analyses are sophisticated enough to provide high-quality information. The programmer can then modify the source code or add directives as he or she examines the compilation results. Furthermore, SUIF Explorer couples compiler analysis results with dynamic program profiles and provides guidance to help the programmer choose the proper program transformations. As a result, the parallelization process is less fragile and more productive.

Another goal of SUIF Explorer is to help compiler researchers design next-generation parallelizers. Previous researchers have shown the importance of having a comprehensive set of interprocedural analyses in creating code that runs on multiprocessors effectively[53]. Finding the key interprocedural analyses that are missing is important. Our experience in using SUIF Explorer helps to identify missing interprocedural analyses, which are important for developing next-generation parallelizers. In using the Explorer, the programmer may encounter some optimization opportunities that are specific to the application. On the other hand, the programmer may encounter some *common* optimization opportunities. We

identified two such optimization techniques, interprocedural *array liveness analysis* and *array reduction analysis*. We developed these two key analyses and integrated them into the parallelizer.

## 1.1. Thesis Overview

SUIF Explorer both helps the user develop parallel code and helps the compiler researcher develop next-generation parallelizers. Thus, the Explorer is both a parallel programming tool as well as a tool for finding missing compiler techniques. We first present the design and evaluation of the Explorer as a parallel programming tool. Next, we describe two compiler algorithms, array liveness and array reduction analyses, that were implemented and integrated into the parallelizer.

### 1.1.1. SUIF Explorer

SUIF Explorer is an interactive and interprocedural parallelizer. Experience with previous parallel programming tools suggests that more powerful compiler analyses are necessary to help the programmer find coarse-grain parallelism[54]. Motivated by our observation that the SUIF interprocedural compiler is now capable of producing sophisticated analysis results that are truly helpful to programmers, we want to make this information available to the user in the Explorer. Furthermore, we want to filter the information intelligently and guide the programmer through the process of debugging and improving an application's parallel performance. In addition to the high-quality parallelization information that distinguishes SUIF Explorer, the Explorer is the first tool to apply slicing analysis to aid the programmer in uncovering program properties for interactive parallelization. We show in Chapter 3 that slicing helps the programmer's investigation process in parallelization.

This thesis presents the design of the SUIF Explorer. The Explorer includes four basic components: (1) the compiler analyses, (2) the Execution Analyzers, a suite of tools for analyzing sequential and parallel executions, (3) the visualization system that interfaces with the programmer, and (4) the Parallelization Guru that captures knowledge specific to parallelization for the purpose of assisting the parallel programming. The Parallelization Guru runs the static and dynamic analysis modules, analyzes the data, and formulates a strategy to

improve the code. The Guru also interacts with the programmer via the visualization module. It focuses the programmer's attention on the critical lines of code and data structures in the program and leads the user down the same path that a compiler expert might take in modifying the program.

This thesis shows that the system is effective in assisting a programmer in finding coarse-grain parallelism in sequential programs. The Explorer minimizes the number of lines of code requiring manual examination using three techniques: advanced interprocedural parallelization, sophisticated dynamic execution analyzers, and program slicing.

We experimented with real-world applications such as `arc3d` from NASA Ames Research Center, `hydro` from Los Alamos National Laboratory, and `flo88` from the Center for Integrated Turbulence Simulations at Stanford. SUIF Explorer successfully minimizes the number of lines of code that need assistance, and results in good overall parallelization. The key to the Explorer's success lies in having sufficiently powerful analyses that can restrict the need for user assistance to a small number of lines of code. It is critical that the SUIF compiler parallelize many of the loops automatically and leave only a few unresolved dependences in the remaining sequential loops.

### 1.1.2. Interprocedural Array Liveness Analysis

Transformations on loops and array data layouts have proven to be very effective in improving cache performance. Such optimizations can only become more important as the gap between processor and memory speeds continues to widen. We show that liveness analysis is an enabler of these optimizations. The analysis can be used to eliminate the need to finalize a privatizable array, to separate live ranges of array variables so their layouts can be optimized independently, and to enable array contraction. We show that interprocedural array liveness is a key analysis that should be included in any modern parallelizing compiler.

We propose a context-sensitive and flow-sensitive interprocedural algorithm that analyzes the program in two phases: a bottom-up phase that summarizes the exposed uses in a code region starting with the innermost ones and a top-down phase that propagates the global

liveness information down starting with the outermost region. We use sets of systems of linear inequalities to represent array accesses. Our experimental results show that the algorithm is effective in finding many dead array variables at loop boundaries. The precision in the algorithm is important as we show that simpler versions that do not differentiate between array elements or ignore the control flow within regions in the top-down phase yield inferior results. Finally, as an enabler of several optimizations, the liveness information is effective in helping produce fast codes on both uniprocessors and multiprocessors.

### 1.1.3. Interprocedural Array Reduction Analysis

A reduction is the application of an associative operation (for instance, addition, multiplication, and finding minimums and maximums) to combine a data set. Reduction operations occur often in scientific programs. Because of the commutativity of a reduction operation, a reduction can be parallelized by having each processor compute a partial reduction locally and update the global result at the end. We present a powerful algorithm for finding reductions. The algorithm extends beyond previous approaches in its ability to locate reductions to array regions, even in the presence of arbitrarily complex data dependences. As an important example, the algorithm can locate reductions on indirect array references through index arrays.

The algorithm can locate *interprocedural reductions*, reduction operations that span multiple procedures. We have integrated the algorithm into a fully functional interprocedural parallelizer. We show that interprocedural reductions occur in some computationally-intensive loops in scientific programs. We measure the impact of reduction recognition on parallelization of a collection of programs and the overhead of parallel reduction.

## 1.2. Organization of the Thesis

The organization of this thesis is as follows. In Chapter 2, we introduce our system for interactive parallelization: SUIF Explorer. Chapter 3 discusses the application of slicing to interactive parallelization. The experimental results of using SUIF Explorer are presented in Chapter 4. We discuss both the applications and the algorithm for the array liveness anal-

ysis in Chapter 5. Chapter 6 describes interprocedural reduction analysis. We conclude in Chapter 7.

# 2 The SUIF Explorer

We developed an interactive parallelizer, called SUIF Explorer, to make the compiler analysis results available to the programmer in an interactive manner. The Explorer couples compiler analysis results with dynamic program profiles and provides guidance to help the programmer choose appropriate program transformations. As a result, the parallelization process is more robust and productive. In Section 2.1 we discuss the motivation for SUIF Explorer. Next, we present several distinguishing features of SUIF Explorer in Section 2.2. Section 2.3 presents an overview of the Explorer architecture and the components in the system. The various components of the Explorer are described from Sections 2.4 to 2.8. Section 2.9 discusses the related work, and Section 2.10 summarizes the chapter.

## 2.1. Motivation

There has been much progress in automatic parallelization in recent years[15][18][19][21][45][51][57][62][66][101][103]. In particular, interprocedural analysis for both scalar and array variables has proven powerful enough to allow compilers to parallelize loops involving hundreds of lines of code and spanning several procedures. This coarse-grain parallelism is critical to achieving effective parallelization on multiprocessors. However, automatic parallelization is fragile as a single data dependence reported by the compiler can ruin the program's parallel performance. An example of a coarse-grain parallel loop missed by the compiler is found in the `hydro` application from Los Alamos National Laboratory. An outline of the loop is shown in Figure 2-1. The boxes represent procedures, and the lines represent procedure invocations. Although the parallelizing compiler can automatically parallelize 22 inner loops, marked using dark gray shading, the compiler misses the 268-line outer parallel loop, marked using light gray shading, due to a few dependences that can be resolved with human intervention. In addition, automatic parallelization is limited by its

focus on common optimization opportunities and by its lack of application-specific knowledge.

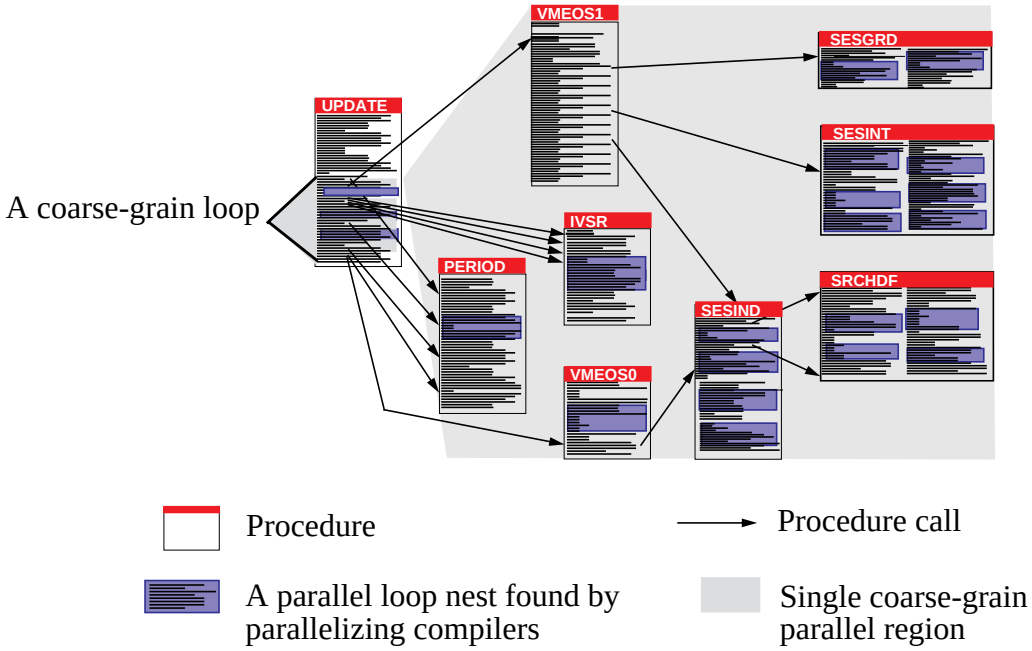


**Figure 2-1. Parallel regions from a code segment in `hydro`**

Finding coarse-grain parallelism in legacy codes *manually*, on the other hand, is difficult. Parallelizing coarse-grain loops by hand requires the programmer to fully understand many lines of code. It is generally not a matter of simply declaring a particular loop to be parallelizable, as it is often necessary to modify the data structures of the program, such as changing global arrays into private copies on each processor. Manual transformation errors can also cause race conditions which are hard to track down.

A solution that combines the advantages of both automatic and manual parallelization is to incorporate users' input into the automatic parallelization process. Several previous systems combines the knowledge of the compiler/run-time system and the user [1][11][28][60][61][71]. A common model is for the user to provide information in an *a priori* fashion, in the form of explicit directives to the compiler to perform or ignore some compiler analysis or optimization[60][61]. This requires that the application programmer have relatively deep knowledge of the compiler. Another model is for the compiler to make the analysis results available to the programmer in an interactive manner[1][11][28][71]. The user can then modify the source code or add directives as he or she examines the compilation result.

Experience with the earlier systems such as Parascope Editor[28] suggests the importance of coupling compiler analyses with dynamic program profilers[54]. This approach has been adopted in more recent systems such as dPablo browser[1], ForgeExplorer[11], and the KAP/Pro Toolset[71]. In addition, it is found that more powerful compiler analyses are necessary to help the programmer find coarse-grain parallelism, and that the programmer needs guidance in choosing the proper program transformations[54].

Our interactive and interprocedural parallelizer, SUIF Explorer, makes the analysis results available to the programmer in an interactive manner. The Explorer couples compiler analyses results with dynamic program profiles and provides guidance to help users choose the proper program transformations. As a result, the parallelization process is more robust and productive.

## 2.2. Features of SUIF Explorer

The SUIF Explorer is a new interactive parallelizer designed with the goal of enabling programmers without compiler expertise to parallelize a program with minimum effort. We achieve this through three distinguishing features:

1. *Deep program analysis*. Deep program knowledge is a prerequisite to an effective interactive parallelizer as it minimizes the mundane work that the programmer must perform. The SUIF Explorer is based on the SUIF parallelizing compiler[51][53][108],

which is a state-of-the-art interprocedural parallelizer. Its repertoire of analyses includes array privatization, which is one of the techniques critical to eliminating spurious dependences. The SUIF Explorer also includes a set of dynamic execution analyzers that can provide valuable run-time information to the programmer. It can pinpoint the loops that dominate the execution and thus deserve attention. It can also locate loops that are potentially parallelizable by detecting if they have loop-carried dependences in sample runs of the program.

2. *Guidance to improving program performance*. Instead of simply presenting all the raw data to the programmer, our system uniquely includes an active agent, known as the Parallelization Guru, which guides a novice user through the parallelization process. It integrates the static and dynamic information derived from the compiler and the execution analyzers, focuses the users on the important loops, and asks pertinent questions about the program that are critical to parallelization. The intention is that the programmer only needs to know about how the program works rather than the details of effective parallelization. The Guru communicates with the programmer using the Rivet visualization system which displays the information at different levels of detail using different visual metaphors.

3. *Assistance with user inputs via interprocedural slicing.* Our experience with the system is that the user often makes costly mistakes when attempting to parallelize the code. The SUIF Explorer uses the concept of program slicing[106] to make the user's parallelization process less error prone. A program slice of an expression is defined as a subset of statements that may potentially affect the value of the expression. By presenting the programmer with the program slice that affects the accesses in a data dependence relationship found by the compiler, the system focuses the programmer's attention on fewer lines of codes and thus reduces the likelihood of human error. The SUIF Explorer also checks the user assertions to ensure that they do not contradict the program analysis results.

## 2.3. The SUIF Explorer System

The SUIF Explorer System is designed to lead the user down the same path that a compiler expert might take in improving a parallel program. The main components of the system are shown in Figure 2-2. They are execution analysis, interprocedural compilation, interprocedural slicing, and visualization.

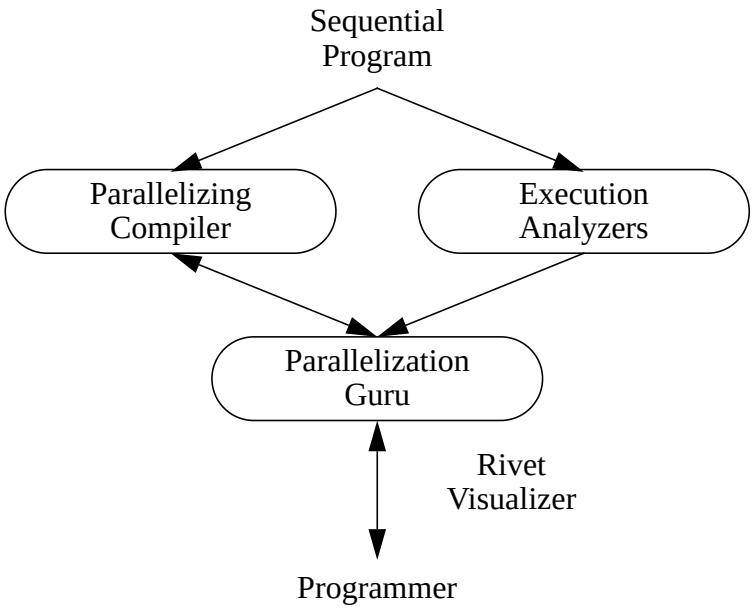


**Figure 2-2. Components of the SUIF Explorer**

### 2.3.1. Parallelization Process in SUIF Explorer

In parallelizing a program, SUIF Explorer first invokes the compiler to parallelize the code. Then, the Explorer instruments the parallelized code using the dynamic tools and gathers profile data of an execution. The Parallelization Guru module analyzes the static and dynamic information to identify target loops. The Guru uses *Execution Analyzers* to detect the long-running sequential loops that are potentially parallelizable. These loops have the greatest potential of improving program performance once they are parallelized. Next, the

Guru uses the SUIF parallel compiler to focus user's attention on only the statically dependent variables. Active guidance is provided for the user. Finally, the demand-driven slicing algorithm is invoked to help users decide the parallelizability without examining too many lines.

The Explorer interacts with the programmer via the visualization module. The hierarchy of visualizers is used to show key aspects of the code being parallelized to the user and let users control the information being displayed.

In the following, we first present an overview of the compiler and the execution analyzers, before presenting the details of the Parallelization Guru and the visualization.

## 2.4. Automatic Parallelization

The SUIF compiler consists of a large number of parallelization analyses designed to find coarse-grain parallelism in a program[51][52]. Our analysis of array accesses is based on the polyhedral theory of integer programs[7][56]. Array regions are represented as sets of systems of linear inequalities, and general mathematical algorithms are used to precisely capture the data accesses in a program. All of the parallelization analyses can be applied across whole programs, thus allowing information to be gathered across procedural boundaries. The list of analyses implemented is as follows:

- Symbolic analysis on scalar variables[51][58]: The symbolic analysis finds loop invariants and induction variables, determines affine relationships between variables, and performs constant propagation.

- Dependence analysis on scalar and array variables[14][85][115]: The analysis checks whether the parallel execution of a loop violates serial ordering constraints between any write operation and any other write or read operation to the same memory location.

- Detection of privatizable scalar and array variables[4]: The analysis checks whether the value used in each iteration comes from any previous iteration. If it does not, the loop-carried data dependence can be eliminated by giving each processor a private copy of the variable.

- Detection of reduction operations to both scalars and array variable[51]: A reduction (for instance, computation of a commutative and associative operation such as sum and product) can be parallelized by having each processor compute a partial reduction locally and update the global result at the end.

The parallelizer uses results of these analyses to parallelize the outermost loops in the program whenever possible. The programmer only has to concentrate on the sequential loops. Furthermore, even when a compiler fails to parallelize a loop, it can often determine that many of the data structures used in the loop are either parallelizable or privatizable. With interprocedural analysis, the compiler is able to eliminate many of the spurious data dependences that limited the usability of previous systems. The programmer can thus concentrate on the remaining difficult dependences.

## 2.5. Execution Analyzers

We have developed a set of tools that analyze different aspects of a program's execution that are relevant to parallelization. In the following, we describe two tools useful for locating important loops to parallelize: *Loop Profile Analyzer* and *Dynamic Dependence Analyzer*.

### 2.5.1. Loop Profile Analyzer

It is well known that most of a program's execution time is spent on a small percentage of the code. The Loop Profile Analyzer is a high-level tool that helps identify the important loops that need attention. It runs a program sequentially, and determines for each loop its total execution time and its average computation per invocation. This information indicates which loops dominate the execution time and whether the computation time is spread over many different invocations of the loop. The Loop Profile Analyzer uses the compiler to insert instrumentation code into the program to record timing information at the beginning

and end of each loop execution. The execution overhead of this code is rather small, and the analysis is very fast.

### 2.5.2. Dynamic Dependence Analyzer

Our second dynamic tool, the Dynamic Dependence Analyzer[88], computes the dependences that arise during an execution of the program and determines what loops may be parallelizable. The dynamic dependence analyzer works by instrumenting the read and write accesses of the program and keeping track of the most recent write operations for each memory location in the program. It is aware of the induction variables and reduction operations found by the compiler, and will ignore dependences on these variables. It also ignores anti-dependences and can detect parallelism that requires data to be privatized.

The analysis is very useful in locating potential parallelism in a program. The absence of any dynamic dependence in executing a loop is a hint that the loop may be indeed parallel. The analysis is expensive because it records all accesses. To speed up the instrumentation, we implement two optimizations. First, the instrumentation will skip accesses proven to be independent by compiler. Second, the instrumentation can skip batches of iterations because the analysis result is used only as a hint. As a result, the performance of the analyzer is acceptable.

## 2.6. The Parallelization Guru

The Parallelization Guru uses two quantitative metrics to guide the parallelization process:

- *Parallelism coverage* is defined as the percentage of total execution time spent in the parallel regions. According to Amdahl's law, the total speedup from parallelization is fundamentally limited by the fraction of time spent in sequential regions of the code: for example, if only half of the program execution is parallelized, the limit on the speedup factor is two. As a result, having a high parallelism coverage is critical.

- *Parallelism granularity* is defined as the average length of computation between synchronizations in the parallel regions. Due to overheads of synchronization and data communication, parallelizing fine-grain parallel loops on multiprocessors can actually

result in a *loss* of performance. Thus, a high parallelism coverage alone does not automatically lead to better parallel performance. If most of the computation is spent in fine-grain parallel loops, we try to parallelize the enclosing loops to increase the parallelism granularity.

The goal of the Guru is to increase the parallel performance by increasing both the parallelism coverage and parallelism granularity. It presents to the programmer the coverage and granularity of the automatically parallelized code, and updates the information as new loops are parallelized. It also presents to the programmer a list of loops to parallelize. The list contains all the sequential loops that have no I/O and that are not dynamically nested under a parallel loop; the loops are sorted in decreasing order of their execution time as measured by the Loop Profile Analyzer. Note that a sequential loop may be nested in another sequential loop, so parallelizing an outer loop may eliminate the need to parallelize an inner sequential loop. Attached to each loop is the information on whether they contain any loop-carried dynamic dependences found by the Dynamic Dependence Analyzer and the number of static data dependences found by the parallelizing compiler.

The Guru then interacts with the programmer, starting with the loop at the top of the list. If the loop has many dynamic and static dependences, the user may decide not to attempt the parallelization of that loop. The Guru looks up each static dependence in a loop and presents the program slice that needs to be examined to the programmer. The programmer then determines if the static dependence can be ignored or if an array can be privatized. He or she may also choose to rewrite the code to eliminate the dependences.

## 2.7. Visualization

The *Rivet visualization environment*[22] is a flexible tool for the efficient creation and use of visualizations for complex systems. Rivet enables the development of visualizations using well-known tools: visual metaphors are implemented in C++/OpenGL and are configured using Tcl/Tk. Tcl is also used to combine visual metaphors, enabling sophisticated visualizations to be built from simpler components.

Rivet provides three metaphors to show key aspects of the code being parallelized to the user and to let users control the information being displayed. This enables the Explorer to present information on large programs at multiple levels of detail. These metaphors include:

- *Hyperbolic graph browser.* A “focus-plus-context” graph drawing algorithm such as the hyperbolic graph viewer[86] is able to display much larger graphs than traditional layout techniques. Nodes that are the focus of the current view are large, and the nodes become smaller as they are located further from the focus. This browser can be used to display structures such as the function call graphs of large programs. This is especially useful for very large systems with hundreds of thousands of lines of code, providing a high-level understanding of the program structure and a roadmap for navigating through the system. Figure 2-3 shows the hyperbolic view of the call graph for the 98,000-line Gamess application from the SpecHPC benchmark. The size of the graph makes it impossible to navigate the call graph in the traditional 2-D space. Because the Rivet codeview described below is effective only up to tens of thousands lines of code, the hyperbolic viewer is used to help guide navigation for larger programs by distilling the program text down to the call graph.

- *Line-oriented program statistics*. Inspired by the SeeSoft system[35], the Codeview metaphor provides a “bird’s-eye” view of the source code. Each line of the source is displayed as a single line segment whose length is proportional to the textual length of the line. This view can be used to display attributes of the source code on a line-by-line basis, allowing the user to see information about thousands of lines of code at once. While the SeeSoft system shows mainly static information, Codeview shows both static information, such as loop nesting depth, and dynamic data, such as the amount of execution time spent on each line. Figure 2-4 provides a “bird’s-eye” view of the source code of the `mdg` application from the Perfect Club benchmarks[90], where each character in the code is represented by a pixel. The user can move the scroll bar in either viewer to control the code displayed in the source code viewer. The

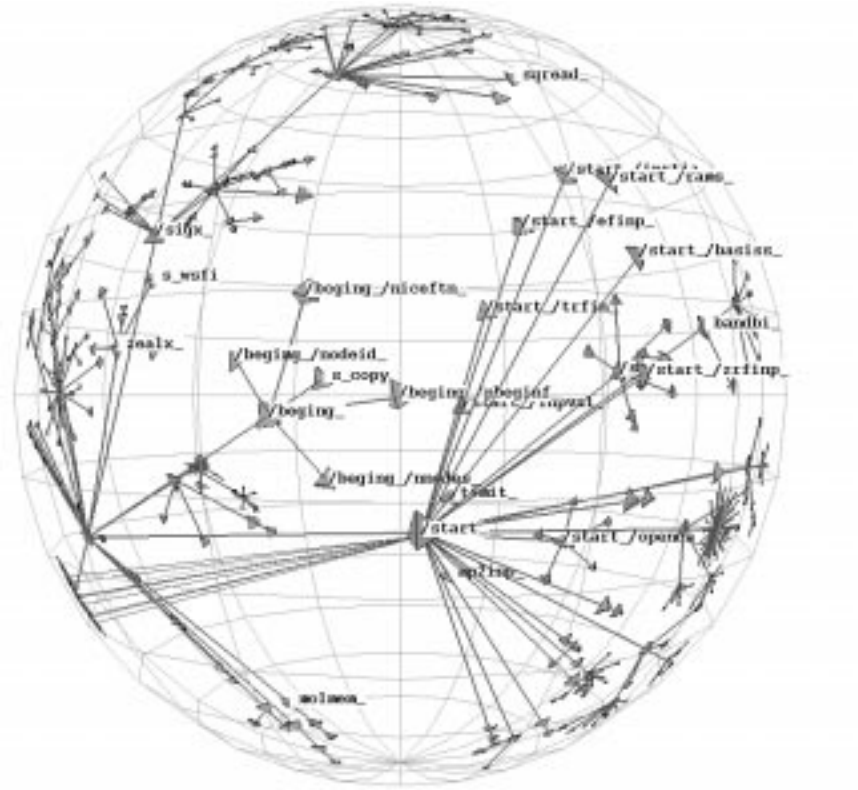


**Figure 2-3. Hyperbolic call graph viewer**

Explorer or the user can use a set of sliders to determine if loops should be filtered from the code view according to their loop depth, granularity and execution time. Filtered loops are shown in gray; unfiltered sequential loops are shown in black; unfiltered parallel loops are shown in white. A white focus bar in the Codeview indicates that the loop was selected as a good candidate for hand parallelization.

- *Source code viewer.* Rivet also includes an enhanced source code viewer which allows the code to be annotated with additional information through the use of color, font selection, and text. This traditional view is limited to displaying tens of lines of code at

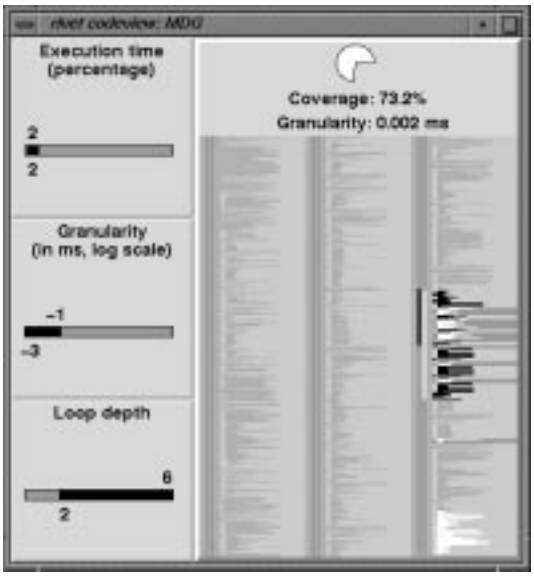


**Figure 2-4. Rivet codeview**

a time, but is very useful when linked with the visual metaphors above, enabling the user to see the source code associated with a region of the Codeview or a node of the call graph.

We utilized the extensibility and configurability of Rivet in developing these metaphors and in devising the interface between Rivet and the Explorer.

## 2.8. Assertion Checkers

Our experience is that the programmer is often mistaken when making assertions on the program, and these mistakes can result in long and difficult debugging sessions. To help make this process more robust, the Explorer includes an assertion checker which uses the available static and dynamic information to try to disprove the programmer's assertion. If the user asserts that two references are independent, the Explorer checks the information against the Dynamic Dependence Analyzer to determine if any true dependence has been

observed for the user-supplied input set. If the user asserts that a global array needs to be privatized in a procedure, the Explorer checks if a similar assertion is provided for all other called procedures that access the same array. If it is not, it issues a warning and privatizes the array for the programmer automatically.

After checking for consistency, the Explorer inserts the annotations in the program, which are then used by the compiler to re-parallelize the code. The programmer need not change the source directly, because the compiler will automatically change global arrays into private arrays where needed and invoke the necessary run-time routines.

## 2.9. Related Work

Four prior systems, the Parascope Editor (PED)[54], the dPablo browser[1], ForgeExplorer[11], and the KAP/Pro Toolset[71], support various degrees of interactive parallelization. PED displays data-dependence information to the user and provides a variety of source-to-source transformations that the user can pick to improve the code. Users have found the data dependence information to be too low-level, and they need guidance with program transforms. Finally, PED does not use sophisticated program analyses such as interprocedural array privatization and reduction and does not integrate dynamic performance data. The dPablo browser from University of Illinois extends the functionality of PED by coupling run-time measurements to the source code and providing visualizations of data-access patterns.

ForgeExplorer is a commercial system that, like PED and the dPablo browser, displays data-dependence information. ForgeExplorer also provides visualizations of UD-chains (Use-Definition chains), DU-chains (Definition-Use chains), control flow in the program, and all the references to a variable and its aliases. Unlike the SUIF Explorer, ForgeExplorer does not provide the slicing information. The program slice of an expression, as defined in Section 2.2, is the subset of statements that may affect the value of the expression. We will advocate the use of slicing for interactive parallelization in detail in Chapter 3.

The KAP/Pro Toolset contains tools to check the validity of users' OpenMP directives[71] and to detect communication leaks. The directives used in the SUIF Explorer are similar to

OpenMP directives. Also, the KAP/Pro Toolset provides users with dynamic dependence information which is similar to that generated by our dynamic dependence analyzer. In summary, the SUIF Explorer provides users with higher quality information than these prior systems because of its interprocedural analysis and higher-level tools such as program slicing.

Finally, few compilers provide a generic interface to access their high-level analysis results beyond data dependencies. The dPablo browser exports data distribution to users in addition to data dependencies. But it is limited to the source-to-source parallelizing transformations supported by the Fortran D compiler and requires users to come up with the transformations needed. The Paradyn system[114] from the University of Wisconsin provides visualization of memory profiles and indicates conflicts. It has a Performance Consultant which uses a Bottleneck Hypothesis Hierarchy to search for the performance bottleneck but stops short at guiding users as well. In comparison, SUIF Explorer formulates an optimization strategy and provides extensive compiler analysis to explain the performance characteristics in the context of the source code structure. The Explorer distinguishes from the previous work in that it is based on state-of-art interprocedural analysis and gives more meanings and reasons to the raw numbers.

## 2.10. Chapter Summary

We developed the SUIF Explorer, an interactive and interprocedural parallelizer, to increase the productivity of parallel programming and to exploit the multiprocessors effectively. This chapter presents the design of the SUIF Explorer and shows how the system can assist a programmer in finding coarse-grain parallelism in sequential programs. The Parallelization Guru in the Explorer couples compiler analysis results with dynamic program profiles and provides guidance to help the programmer choose the proper program transformations.

The Explorer minimizes the lines of code requiring manual examination by using three techniques: advanced interprocedural parallelization, sophisticated dynamic execution analyzers, and program slicing. The key to the Explorer's success lies in having sufficiently powerful analyses that can restrict the need for user assistance to a small number of

lines of code. It is critical that the SUIF compiler parallelizes many of the loops automatically and leaves only a few unresolved dependences in the remaining sequential loops. Chapter 4 will present the experimental results on the effectiveness of the Explorer in minimizing the number of lines of code requiring user inspection.

# 3 Slicing for Interactive Parallelization

SUIF Explorer is more effective than previous systems in minimizing the number of lines of code that require programmer assistance. In Chapter 2, we show that the interprocedural analyses in the SUIF system can parallelize many coarse-grain loops, thus minimizing the number of spurious dependences requiring attention. We also discuss how the Explorer uses dynamic execution analyzers to identify those important loops that are likely to be parallelizable. But even after the Guru narrows the parallelization question down to whether a pair of memory references in a loop is dependent, the programmer is still presented with a difficult problem. To answer the question, the programmer may have to examine the entire loop, and sometimes even code outside the loop, to identify all the statements that may affect the dependence. This examination is both labor intensive and error prone, especially if the loop bodies contain procedure calls.

To solve this problem, we apply the concept of program slicing to interactive parallelization. Slicing, first introduced by Weiser[106], is defined as a subset of the program statements that directly or indirectly contribute to the values assumed by a set of variables or memory locations at some program point. Slicing is used mostly for software engineering such as software maintenance, testing, and debugging. We are the first to apply demand-driven slicing to interactive parallelization[78]. As a result, the Explorer can automatically reduce the code to only a small number of statements that are related to parallelization and privatization. The experimental results from applying slicing to interactive parallelization are presented in Chapter 4.

The chapter is organized as follows. In the next section we will motivate the need for slicing with an example. Section 3.2 describes how slicing can be used for interactive parallelization. Section 3.3 examines the design of the slicing tool. Section 3.4 and 3.5 present our

program representation, interprocedural SSA form, and our context-sensitive interprocedural slicing algorithm, respectively. Section 3.6 describes slice pruning options which help programmers cope with large slices. Section 3.7 discusses related work.

## 3.1. Motivation with a Real-Life Example

The use of slicing was motivated by a real-life experience. A user tried to use the Explorer to parallelize his application that computes asset allocations in a portfolio. Figure 3-1 shows an excerpt from a 96-line loop that includes four procedure calls. The compiler could not parallelize this loop because of the dependences of array `XPS`. In his eagerness to speed up the application, the programmer declared that the `XPS` array is privatizable. The reasoning was that since line 2349 writes `XPS[1:NLS]` before it is read by line 2356, there is no loop-carried dependence if every processor gets its private copy of the array. This information enabled the compiler to parallelize the loop, but unfortunately the parallelized code did not run correctly. This was a costly mistake and it took the programmer

```
      XPS(...) = ...
      DO 2365 S=1,N
2320   IF ((S.NE.1).AND.(S.NE.5).AND.REE) GO TO 2355
       ...
       DO 2350 H=1,NLS
        ...
2349    XPS(H) = Y(H+1)
2350   CONTINUE
       ...
2355   DO 2360 JJ=1, NLS
2356    XP(S+(JJ-1)*N)=XPS(JJ)
        ...
2360   CONTINUE
        ...
2365 CONTINUE
```

**Figure 3-1. A code excerpt illustrating the usefulness of slicing**

many hours to realize that the array is not privatizable. He did not notice that statement 2320 can cause the control flow to bypass the statement 2349. As a result, the data read by line 2356 in some iterations were written by a previous iteration of the loop; therefore, the array cannot be privatized.

This example illustrates how easy it is to make a mistake when trying to determine if a data dependence exists in a code. The concept of a program slice helps alleviate this problem by distilling out the code that must be analyzed. As explained below, the program slicer will highlight exactly those lines shown in the excerpt. Then, the programmer can easily make the right inference when presented with these ten lines of code, rather than the 96-line loop in its entirety.

## 3.2. Slicing for Interactive Parallelization

We apply the program slicing to interactive parallelization, because slicing can reduce the code to only the statements relevant to the parallelization of the loop. We will first introduce various types of slices and then describe how to use them for interactive parallelization.

### 3.2.1. Defining Slices

The *program slice* of a reference is the set of operations that contribute to the value of the reference. The *data slice* of a reference is a subset of the program slice in which only the data dependence edges, and not the control dependence edges, are followed. We introduce the concept of *control slice*, which is all the operations that affect the conditions under which a reference is executed. Control slice is also a subset of the program slice. It is computed as the immediate control dependences of the reference plus the program slices of the expressions upon which the reference is control dependent.

### 3.2.2. Using Slices for Parallelization

As illustrated by the example in Section 3.1, proving data independence between a pair of array accesses requires us to know the *locations* that are read and written, and the *condition* under which the accesses are performed. The program slices of the array index expressions specify the locations accessed, and the control slices of the accesses specify when the

accesses are performed. When presented with these slices, the programmer can use the following procedure to determine if there is a dependence:

1. Analyze the program slices of the array indices to determine the ranges of the array indices read and written. If they do not intersect, or if the location read in an iteration is written only in that iteration, there is no loop-carried dependence that prevents parallelization.

2. Analyze the program and control slices of the accesses to see if the data read in an iteration were written earlier in that same iteration. If they were written earlier, the array is privatizable. In the example above, the write operation depends on the condition in line 2320 whereas the read operation does not. The read is not always preceded by a write in the same iteration, and therefore the array is not privatizable.

## 3.3. Requirements for the Interactive Slicing Tool

A common technique for computing program slices is to transitively follow all of the control and data dependence edges originating from the reference being sliced. But this notion of slicing, originally proposed by Weiser, is context-insensitive[106]. The imprecision is introduced when a parameter passed in one call to a procedure reaches the definition of a return variable to another call to the same procedure. But for slicing to become a practical tool for interactive parallelization, we need a more *precise* slice. Thus, we need a *context-sensitive* slice[63], a slice that is the union of all statements in all the paths that reach the reference while matching in- and out-parameter bindings.

We summarize our design goals as follows:

- Precise: Slices of interest must be significantly smaller than the number of statements in a program. This is especially important in an interactive tool because the user's time is precious. We do not want to overwhelm users with large slices. Therefore, the slicing algorithm should be context-sensitive.

- Demand-driven: To handle the interactive queries, the slicing tool needs to be able to slice the program on demand.

- Time and space efficient: For interactive uses, the slicing tool needs to run fast enough on real-world codes. Thus, we want to design an efficient program representation and a compact slice representation. We describe the former in Section 3.4 and the later, Section 3.5.4.

- Extensible: Our slicer should allow users to specify the traversal functions during the slice computation. Thus, users can control the type of slices they want to examine. For instance, users may want to specify different slice-pruning options as in Section 3.6.

## 3.4. Program Representation: Interprocedural SSA Form

We will first describe the program representation, before presenting the algorithm in the next section.

Our slicing algorithm operates on an interprocedural SSA form (ISSA), which differs from traditional SSA form (Static Single Assignments[30]) in two ways. First, it incorporates pointer alias information in the representation. Second, it contains additional nodes (similar in function to parameter-in and parameter-out nodes in Horwitz *et al*[63]) to connect the intraprocedural form of procedures into a global program graph. The rest of this section describes how we incorporate the alias information into our SSA form for C and Fortran programs[113], and then how we connect the intraprocedural graphs together.

### 3.4.1. Alias Information for C Programs

The pointer alias analysis we use is Steensgaard's flow-insensitive and context-insensitive alias analysis that executes in almost linear time[98]. The analysis partitions all the references into alias equivalence classes. To incorporate the alias information in our SSA form, we assign a new *alias variable* to each alias equivalence class. Then, we substitute every reference in the program with the alias variable representing the equivalence class to which the reference belongs. Assignments to alias variables, representing non-singleton equivalence classes, are treated as weak updates. That is, we introduce a $\phi$ node that combines the original content of the variable with the new value. The $\phi$ expression selects and returns one of its operands based on the path taken to reach the expression. This is similar to Cytron's approach[31], except that $\phi$ nodes are used instead of Cytron's *IsAlias* function.

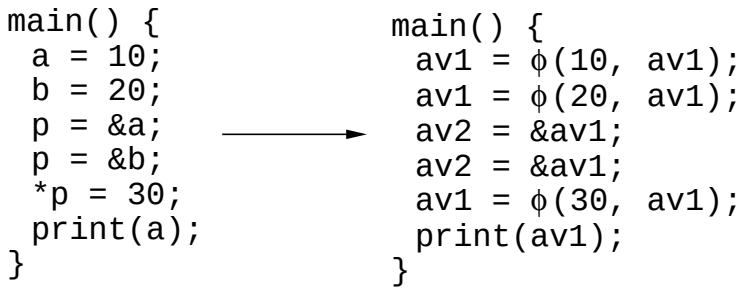


**Figure 3-2. Example of Interprocedural SSA Form**

Consider the example in Figure 3-2. In this example, the pointer alias analysis finds variables `a` and `b` to be aliased. Variable `av1` represents the alias equivalence class consisting of `a` and `b`, and `av2` represents `p`. Since `av1` represents two variables (`a` and `b`), all the right-hand-side expressions that assign to `av1` are ϕ functions, indicating that the variables represented may either retain their old values or be assigned new values.

Once the code is rewritten as shown above, alias variables behave just like regular program variables. For the rest of the paper, we will refer to alias variables as simply variables.

We extended Steensgaard's technique to improve its ability to locate strong updates in the program. We further partition each alias equivalence class so that direct reads and writes to individual scalar variables are placed in their own subclasses, and the rest are placed in the "alias" subclass. As we create the SSA form, direct writes to a scalar variable are treated as strong updates on that variable and weak updates for the alias subclass, whereas writes to the alias subclass are treated as weak updates for all the variables in the same equivalence class. This improved accuracy is helpful in keeping slices small.

### 3.4.2. Alias Information for FORTRAN Programs

Although Fortran 77 does not have pointers, aliases are possible due to the use of common blocks and passing parameters by reference. The former is handled separately by a simple pass that identifies all the overlapping common blocks. The Fortran standard says that even if parameters or global variables may be aliased, the value of a variable is undefined if it is modified by an assignment to an alias of that variable. Thus, we simply model Fortran's parameter passing convention by assigning the actual parameters to the formal parameters before a call while assigning the formals to the actuals after the call (known as copy-in/ copy-out).

Our algorithm does not distinguish between different elements in an array. We assume that any reference to an array element accesses the entire array and any store to an array element potentially modifies the entire array. We handle assignments to array elements in the same way we handle weak assignments in C. We did not find a need for more detailed array information so far in our experience with our system.

### 3.4.3. Building an Interprocedural SSA Graph

We build the interprocedural SSA graph by introducing new $\phi$ functions to capture the semantics of parameter passing. Once these functions are inserted, a standard SSA technique is used to find the interprocedural SSA graph.

Our first step is to apply an interprocedural analysis to find, for each procedure, all the variables that are modified or referenced by the procedure and its callees. They may be either global variables or variables accessed via pointer dereferences in the case of C programs. We handle these variables as if they were parameters of the procedure. All variables, that are potentially modified in the procedure, are treated as return values.

To capture the semantics of return values, we introduce a statement that assigns the formal return variable to the actual return variable after the call site. The edge that corresponds to this assignment is called a *return edge*.

To capture the semantics of passing a parameter into a procedure, we introduce an explicit assignment statement that initializes the parameter. If the procedure has multiple callers,

then the right hand side of the assignment is a $\phi$ function which combines all the corresponding actual parameters passed to that procedure, one from each potential caller. This $\phi$ expression is similar to that in traditional SSA form: it selects and returns one of its operands based on the path taken to get to the $\phi$ expression. For instance, if a procedure called `R` has one formal parameter, `f`, and `R` has two callers, `P` and `Q`, then the formal `f` is initialized with a $\phi$ function of two arguments, the actual parameter passed from `P` to `R` and the actual parameter passed from `Q` to `R`. The key to achieving context-sensitive slicing is: (1) to track the return edge that was traversed and (2) to select the argument of the $\phi$ function that corresponds to the return edge. Take the code segments in Figure 3-1(b) as an example. The context-sensitive slicing will have the $\phi$ expression in the procedure `R` return the argument `G3` when the calling context is `P`, which can be identified by the return edge information during the traversal of the ISSA graph.

For the statement that assigns the formal return variable to the actual return variable, the right hand side of this assignment will also be a $\phi$ node if the call site has multiple callees. Once these assignments are introduced, we compute the minimal SSA form for the whole program using the concept of iterated dominance frontiers[30]. In SSA form, there is exactly one assignment to each variable. SSA form accomplishes this by creating a new version of a variable (SSA variable) at each assignment to the variable while adding additional assignments to ensure that only one version of a variable reaches any use of the variable.

The code segments in Figure 3-3 shows a Fortran code before and after the ISSA transformation. The procedure `R` has one formal parameter `f.` The global variables `G` and `H` used in procedures `P` and `Q`, respectively, are also treated as parameters. A pair of assignments, one at the beginning of each procedure and one after each call site, are introduced to model the “copy-in-copy-out” parameter passing semantics for each parameter. In addition, we augment the ISSA graph with control dependence edges[30], which are necessary for calculating program and control slices.

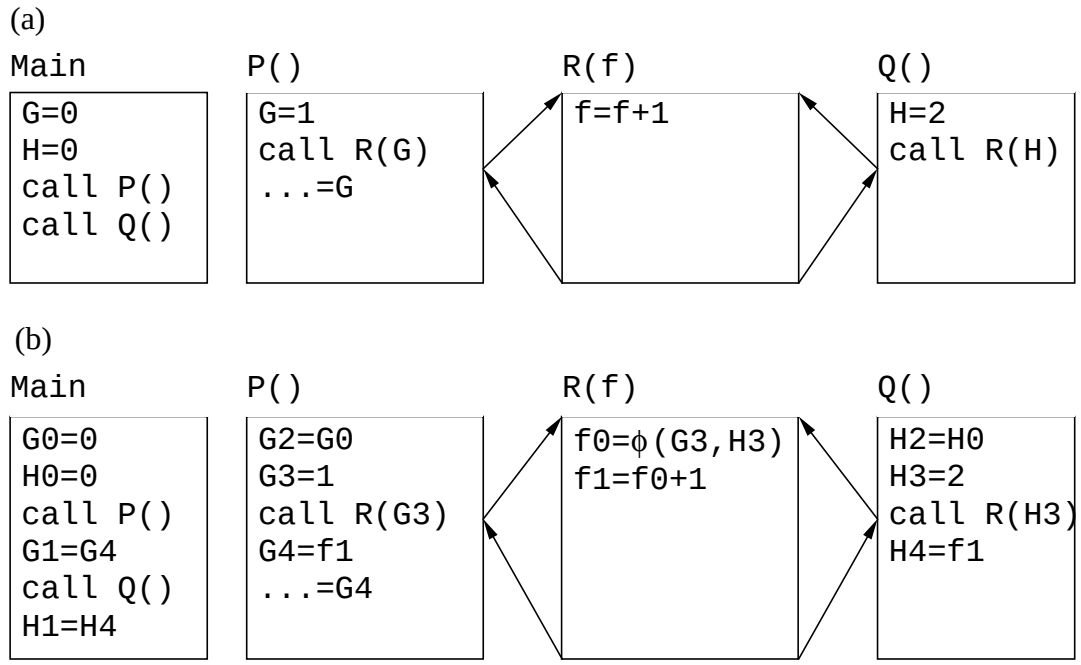


**Figure 3-3. (a) Code before and (b) code after the transformation to ISSA.**

## 3.5. The Slicing Algorithm

We developed a demand-driven, context-sensitive interprocedural slicing algorithm. It first builds a sparse interprocedural SSA representation[30] across the entire program, then on demand, computes the requested context-sensitive slice. To make the slice computation fast, we developed the concept of *slice summaries* which exploit redundancy in the calculation of slices and an efficient representation to make a *set union*, the most common operation in the slice computation, fast.

### 3.5.1. Demand-driven Slice Computation Based on ISSA

After building the ISSA graph for the entire program, our program slicer accepts requests for slices and computes them on demand. The slices computed are *context-sensitive*, and thus do not suffer from inaccuracy due to unrealizable paths. Consider again the example from Figure 3-3. Suppose we are interested in finding the program slice of the read operation of variable `G` in function `P`. A context-sensitive analysis will determine that the slice includes the procedure `R` and the assignment to `G` in procedure `P`. A context-insensitive analysis, on the other hand, will follow the control flow backwards from procedure `P` into procedure `R` via the return edge, and from procedure `R` reach both procedures `P` and `Q`, picking up the assignment to `W` in procedure `Q` as part of the slice. Context sensitivity is very important, because it retains small slices.

In Section 3.5.2, we introduce the concept of *slice summaries*, which are designed to exploit redundancy of slice computations at the procedural level. In Section 3.5.3 we describe our slicing algorithm. Finally, in Section 3.5.4, we describe the *hierarchical slice representation* used in the algorithm which exploits redundancy of slice computations at the statement level.

### 3.5.2. Slice Summaries

We observe that even context-sensitive slices, involving calls to the same procedure, share many common statements. Consider two program slices of a return value from the same procedure *p* at two different call sites. Each of the slices has two portions: the set of statements executed in one invocation of the procedure *p* that contributes to the return value, and the slices of the actual parameters passed into the procedure *p* that contribute to the return value. It is the latter part that is context sensitive. The former set, on the other hand, is identical in both cases and is independent of the calling context; also, it includes statements in the procedure *p* as well as statements in the callees of the procedure *p*. Thus, we only need to compute this set once and reuse it for different invocations to the same procedure. This optimization is just another example of *path compression* used in interval analysis[95].

Let $D$ be the definition of $r$, and $op(r)$ be its operands,

$$SS_r = \begin{cases} \langle \varnothing, r \rangle & \text{if } r \text{ is a formal parameter} \\ \langle S, \varnothing \rangle \cup \bigcup_{f \in F} SS_{GetActual} & \text{if } r \text{ is a return value, and where } \langle S, F \rangle = \bigcup_{v \in op(r)} SS_v \\ \langle \{D\}, \varnothing \rangle \cup \bigcup_{v \in op(r)} SS_v & \text{otherwise.} \end{cases} \quad \text{(EQ 1)}$$

**Figure 3-4. Equation for computing slice summaries**

We define the notion of a slice summary to capture this concept. A *slice summary, $SS_r$,* of a reference *r*, in a procedure *p* is a tuple of two elements <*S*, *F*>: *S* is the call subslice, which is the set of statements in the procedure *p* and its callees that contribute to the value of *r,* and *F* is the set of formal parameters of the procedure *p* that *r* depends on. Thus, *F* is the upwards-exposed uses with respect to *r*. The slice of the reference *r* is the union of its call subslice and the slices of all the actual parameters passed to the upwards-exposed uses in the procedure *p* with respect to *r.*

The slice summary of a reference in the program is calculated by using (EQ 1) in Figure 3-4. We define the *union* of two slice summaries to be the component-wise union of the two tuples. The function *GetActual* returns the actual parameter passed to a formal variable at a call site. If the reference is neither a formal nor a return value at a call site, its slice summary is the union of the slice summaries of all the operands in its definition with the addition of the definition of the reference in the call subslice. If the reference is a formal parameter, its slice summary simply consists of an empty call subslice and a singleton set containing itself as the upwards-exposed use component. If the reference represents a return value at a call site, its definition is a $\phi$ node whose operands are formal return variables, one for each potential callee. First. we find the slice summaries of all the formal

return variables in the callees. The slice summary of the reference consists of the union of the slice summaries of all the actual parameters passed to the upwards-exposed formal parameters of the callees, plus all the call subslices in the callees' slice summaries.

We compute the control and program slice summaries in a similar manner by adding control dependence edges to the above equations.

### 3.5.3. Algorithm to Find Slices

We are now ready to define slices formally in terms of slice summaries. Let $SS_r = <S, F>$ be the slice summary of a reference $r$, and $C_r$ be the set of call sites that invoke the procedure to which $r$ belongs,

$$Slice(r) = S \cup \bigcup_{c \in C_r, f \in F} Slice(GetActual(f,c))$$

That is, the slice of $r$ in procedure $p$ is recursively defined as the call subslice of $r$ plus the union of all the slices of the actual parameters supplied to an upwards-exposed formal variable of $r$ in any potential site that calls procedure $p$.

Here a slice is defined to include all the relevant statements in all the possible paths that lead to the reference of interest in the program. It is sometimes useful to find the slices with respect to some constraints on the execution path of interest. For example, in a debugging session, we may wish to ask for the slice given a particular call stack value. To capture this notion, we define a slice of a reference $r$ with respect to a calling context $C$, denoted as $Cslice(r, C)$, as follows. Let $C=[c_1, ..., c_n]$ be the call sites currently on the call stack, with $c_n$ being the one on the top of the call stack. Let $SS_r = <S, F>$ be the slice summary of a reference $r$,

$$Cslice(r, [c_1 \ldots c_n]) = S \cup \bigcup_{f \in F} Cslice(GetActual(f, c_n), [c_1 \ldots c_{n-1}])$$

That is, the computation of context-specific slices only tracks the slices up the chain of calls on the call stack.

Our demand-driven slice computation accepts a request for a slice, with or without a specific calling context, and uses a primarily recursive descent algorithm based on the equations above. The results of all slice summaries for every program point are memorized as they are computed.

Note that the equations above form a recurrence if recursive procedure calls or loops are present; therefore we need to find the fixed point solution to the equations. Our algorithm locates the recurrences by using a stack to keep track of the slice summaries being constructed. When a recurrence is detected—if a slice summary to be computed is already on the stack—the algorithm simply uses the approximate slice summary found thus far. If an approximate summary is used in the computation of another slice summary, we say that the latter depends on the former. All such dependence relationships are recorded. When the approximate slice summary is finalized, its dependent summaries are placed on a worklist. The algorithm finds the fixed point solution by iteratively removing a summary from the worklist, recomputing it, and adding new dependents on the worklist if the result changes. The fixed point computation terminates when the worklist is empty.

### 3.5.4. Hierarchical Slice Representation

Not only is there redundancy in the calculation of slices at the procedural level, there is also redundancy at the statement level. The slice of a reference in the program is often the same as the slice of another reference plus some additional definitions. Furthermore, the slice of a reference includes the slices of all its definition's operands. Thus, the reuse opportunities among different slices abound. This reuse is especially important if we want to slice many references rather than only a few references. A flat representation that enumerates the full slice of every reference does not exploit this commonality.

To capitalize on the redundancy between sets of statements in slices and slice summaries, we use a hierarchical set representation. We represent a set of statements by a collection of subsets of statements plus additional individual statements. Graphically, a set is represented as a node, labeled by the additional statements, and whose directed edges point to its subsets. Thus, a union operator between two nodes can be performed by simply creating a new node that points to the operands. Figure 3-5 shows an example of a hierarchical slice.

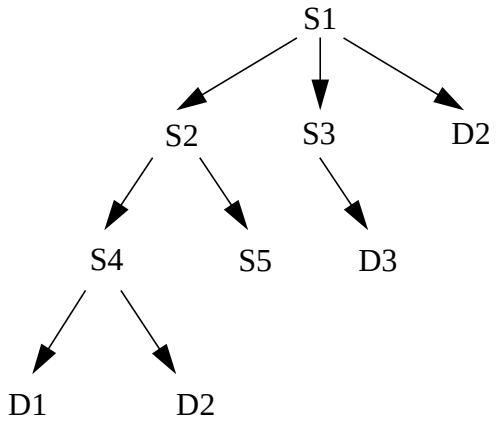


**Figure 3-5. Example of a hierarchy of slices**

In this slice, S1 includes subslices S2 and S3 and definition D4. Slice S2 contains subslices S4 and S5. Note that a slice may be empty. For example, S5 is empty in this hierarchy of slices.

The graph created in our slicing algorithm can be cyclic; two variables can depend on each other by mutually causing the slices to include each other. All elements in a strongly connected component have the same value. We remove the redundancy and simplify the graph by reducing all the strongly connected components to a single node. Edges, that used to point to nodes in the strongly connected components, now point to the new node.

## 3.6. Slice Pruning

Slices of interest must be significantly smaller than the number of statements in a program; otherwise, even a perfect slicing algorithm does not produce useful slices. We found that program and control slices of a reference can get quite large and have thusly created the notion of *slice pruning* to help programmers cope with large slices. The idea is to prune the

slice computation at those nodes that are unlikely to yield useful information for proving data independence. We call these *terminal nodes* and highlight them in the display to remind the programmer not to assume anything about the contents of those nodes. Slice pruning keeps the size of the displayed slice small; therefore the programmer can expand any of these terminal nodes if they so desire. We have found two useful forms of pruning:

- **Array-restricted slices.** Array contents are seldom useful for proving data independence, and slices of array accesses are usually large and imprecise. Thus, it is useful to prune the slice computation at array accesses.

- **Code-region-restricted slices**. The part of a program slice outside of a loop is often irrelevant to the parallelization of the loop. It is useful to prune the slice computation upon reaching nodes outside the loop.

Our slicing algorithm described in Section 3.5 can be easily parameterized to include these restrictions.

## 3.7. Related Work

Few parallelization systems provide a generic interface to access their high-level analysis results beyond data dependencies. The Parascope Editor (PED)[54], the dPablo browser[1], and ForgeExplorer[11] all display data dependence information. ForgeExplorer also provides visualizations of UD-chains, DU-chains, control flow in the program, and all the references to a variable and its aliases. However, none of these systems provides slicing information.

Since Mark Weiser introduced the concept of program slicing[106], much work has been done in the design of slicing methods, such as dynamic slicing and hybrid slicing. Dynamic slicing collects the set of statements bases on a particular run[2]. Hybrid slicing incorporates dynamic information to improve the precision of static slicing[48]. The SUIF Explorer uses static slicing. Furthermore, slicing has been found to be relevant to many software engineering activities, such as program construction, optimization, maintenance, testing, and debugging. But no previous slicing tools target interactive parallelization. Frank Tip provides a comprehensive survey of slicing and its applications[99].

Prior context-sensitive approaches to slicing[63][93] proceed in three steps. The first step builds a program dependence graph[43] which makes data and control dependences explicit. The second step makes one or more passes over the dependence graph to compute edges that summarize the effects of all the calls on the dependences. To be more concrete, it links return values from a call to the actual parameters of the call. This step may take multiple iterations if the program being sliced is recursive. The final step computes the slices from the modified graph in a demand-driven manner. Others have modified this basic approach to work on value dependence graphs[37] instead of program dependence graphs and to apply to object-oriented programs[74].

Our algorithm improves on past approaches in four ways. First, we compute both slices and slice summaries in a demand-driven fashion. Horwitz *et al.*[64] describe how to compute summary edges in a demand-driven fashion for a class of data-flow problems. However, their approach requires constructing an exploded supergraph which has *D* nodes for every node in a program-wide control flow graph, where *D* is the number of possible data flow facts. This graph would be prohibitively large for computing summary edges for slicing. Second, our slicing algorithm can compute slices with respect to a particular calling context. This is useful in debugging and in stepping through context-sensitive slices one level at a time. Third, we compute and memoize slice summaries for all nodes and not just return values. This avoids redundant work especially when we are computing many slices. Finally, we use a hierarchical representation of slices which contributes to our algorithm's efficiency in execution time and memory usage. Prior algorithms for slicing do not specify the slice representation they use.

## 3.8. Chapter Summary

Slicing has been used mostly for software engineering such as software maintenance, testing, and debugging. This chapter shows that the concept of slicing can be applied effectively to interactive parallelization. Slicing information minimizes the likelihood of human error due to two reasons. First, a slice contains only the set of statements that are relevant to resolving a dependence, and second, the question of parallelization and privatization can

be answered by examining the program slices and the control slices. Slicing captures the user's thinking process during interactive parallelization.

Our slicing tool meets a number of design goals. It is precise, demand-driven, time and space efficient, and extensible. Since the user's time is valuable, the SUIF Explorer must reduce the number of lines of code that need to be analyzed. For some applications, we found that even the context-sensitive slicing still produces too many lines of code in the slice. Thus, we proposed the concept of slice pruning to eliminate those statements that are unlikely to yield useful information for proving data independence. By leveraging knowledge of the problem domain of parallelization, the Explorer can further reduce the code to only a small number of statements that are related to parallelization and privatization.

# 4 Experimental Results of SUIF Explorer

This chapter presents the experimental results of using the SUIF Explorer to parallelize applications. First, we illustrate the user-assisted parallelization process by several case studies. The SUIF Explorer system is applied to four applications: `mdg` (a molecular dynamics model from the Perfect Club benchmark suite[91]), `hydro` (a 2-D Lagrangian hydrodynamics program from Los Alamos National Laboratory), `arc3d` (a 3-D Euler equations solver using an implicit method from NASA Ames Research Center), and `flo88` (a wing-body analysis solving transonic flow from the Center for Integrated Turbulence Simulation at Stanford). Second, to evaluate the effectiveness of the SUIF Explorer, we examine both the reduction in the size of the code requiring intervention and the amount of remaining analyses required in the user-assisted parallelization. We present two sets of experimental results:

- Size of code requiring intervention: We report the results of using the Explorer to minimize the lines of code requiring manual examination. Specifically, we analyze the results of the three techniques: advanced interprocedural parallelization, sophisticated dynamic execution analyses, and program slicing.

- Cooperation between the Explorer and the user: We report the amount of work in the user-assisted parallelization. In our experiment, we found that the compiler can parallelize the references to many data structures within the sequential loop automatically, leaving only a small amount of work to the programmer.

Figure 4-1 presents some high-level information about the applications and shows the performance data of automatic parallelization using the SUIF compiler. Our experiments of the Explorer are conducted on the Digital AlphaServer 8400[42], except for the experi-

| | mdg | arc3d | hydro | flo88 |
|---|---|---|---|---|
| Program description | Molecular dynamics model | 3-D Euler equations solver | 2-D Lagrangian hydrodynamics | Wing-body analysis solving transonic flow |
| Data set size | 1029x1029 | 64x64x64 | 450x450 | 256x32x48 |
| No. of lines | 1238 | 4053 | 12942 | 7438 |
| Coverage | 73% | 89% | 86% | 81% |
| Granularity | .002 msec | .3 msec | .3 msec | .1 msec |
| Speedup on 8 processors | 1.0 | 1.6 | 2.7 | 1.0 |

**Figure 4-1. Program information and results of automatic parallelization**

ments for the `flo88` application which were run on an SGI Origin. The details of the Origin system are given later in Figure 6-1. We obtain the results of `flo88` on the Origin due to a bug in Digital's system software. The AlphaServer is a bus-based shared-memory multiprocessor that contains eight 300-MHz Digital 21164 Alpha processors. The Digital 21164 Alpha is a quad-issue superscalar microprocessor with two 64-bit integer pipelines and two 64-bit floating point pipelines[34]. The experimental results are machine-dependent because they are sensitive to the cost of synchronization, the interconnect bandwidth, and the memory subsystem. The Digital 21164 Alpha has two levels of on-chip cache: 8 KB first-level instruction cache and 8 KB first-level data cache, and 96 KB of unified level 2 cache. The memory system allows multiple outstanding accesses to off-chip memory. Each processor has 4 MB of 10ns external cache. The architecture provides 32 integer and 32 floating-point registers. The 256-bit data bus, which operates at 75 MHz, supports

265ns memory-read latencies and 2.1 GB per second of data bandwidth. Banked memory modules are attached to the bus[42].

Figure 4-1 shows that the compiler is able to parallelize 73 to 89 percent of the computation, which is a respectable result; however, it only obtains speedups between 1 to 2.7 on an 8 processors. A high speedup, such as a speedup of more than 6 times on 8 processors, requires parallelizing nearly all of the computation in coarse-grain loops.

The chapter is organized as follows. Section 4.1 and 4.2 illustrate the user-assisted parallelization process by presenting the case studies of `mdg` and `hydro`. Section 4.3 evaluates the reduction in the size of the code requiring intervention due to the compiler analysis results, the dynamic information, and the slicing. Section 4.4 reports the amount of remaining analyses required in the user-assisted parallelization. Section 4.5 presents the improved performance data. Our experience with these applications suggests that the concepts in the SUIF Explorer are effective in assisting a programmer in locating coarse-grain parallelism in a program. Section 4.6 presents the relate work and Section 4.7 concludes the chapter.

## 4.1. Case Study: `mdg`

In this section, we illustrate the experience of using the SUIF Explorer with the `mdg` benchmark from the Perfect Club benchmark suite[91]. The program consists of 1238 lines of code. The `mdg` program implements a molecular dynamics model for 343 water molecules in the liquid state at room temperature and pressure. The code uses the Matsuoka-Clementi-Yoshimine configuration interaction potential for rigid water-water interactions while extending it to include the effects of intra-molecular vibration[91].

### 4.1.1. Applying the SUIF Parallelizer and Execution Analyzers

SUIF Explorer starts by invoking the SUIF compiler to parallelize `mdg` and then runs the parallelized code on an 8-processor Digital AlphaServer. The parallelized application shows no speedup over a sequential execution of the program; in fact, it actually takes slightly longer to complete. Then, the execution analyzers are used to characterize the parallel behavior of the application. The SUIF compiler succeeds in parallelizing 73% of the computation; however, by Amdahl's Law, the parallel code is fundamentally limited to less

than a 2.2-times speedup. In addition, the parallel loops are very small, with each loop taking less than 0.002 millisecond on average if executed sequentially.

### 4.1.2. Finding Targets of Parallelization

The Explorer then evaluates the compiler results and profile data to find the important sequential loops that are potentially parallelizable. By far, the best candidate found is the loop `1000` in the `interf` procedure (`interf/1000`), a loop consisting of 109 lines and three procedure calls. This loop accounts for 90% of the total execution time of the program and each instance executes for a relatively long time, unlike its fine-grained inner

**(a)** **(b)**

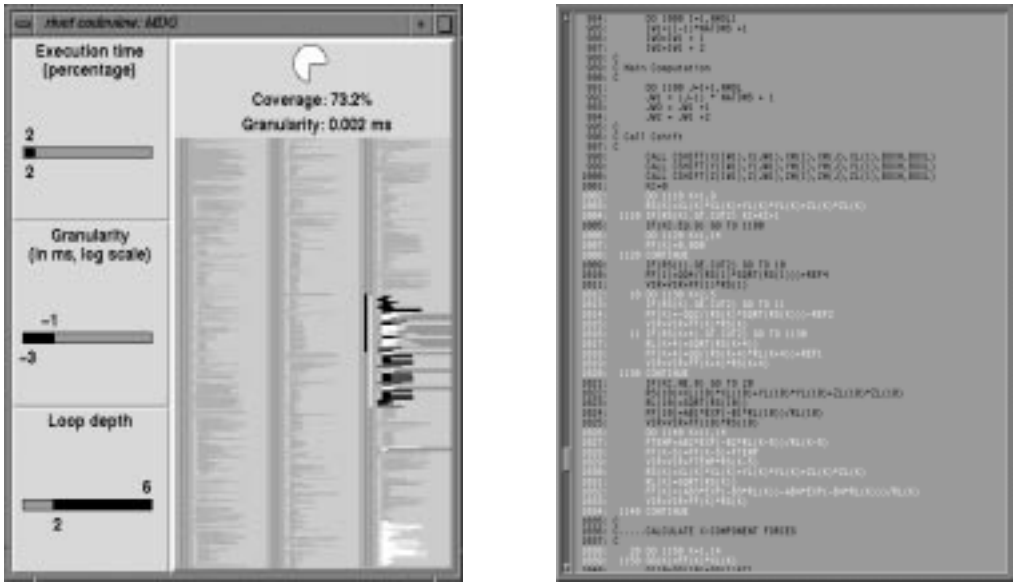


**Figure 4-2. (a) Rivet Codeview and (b) source code viewer displaying `mdg` parallelization information**

loops. This means that parallelizing `interf/1000` will greatly improve the parallel performance of the application. In addition, the single loop-carried dependence (on array `RL`), reported by the compiler, is not observed by the Dynamic Dependence Analyzer (described in Section 2.5.2) for the user-supplied input set. The absence of the dynamic dependence is only a hint that the loop may be indeed parallel. The user is responsible for determining whether the static dependence is real, as shown in the next section.

### 4.1.3. Presenting the Relevant Program Slice to User

This analysis is presented to the programmer using the codeview and the source code viewer, as shown in Figure 4-2. The Codeview provides a "bird's-eye" view of the `mdg` source, where each character in the code is represented by a pixel. The user can move the scroll bar in either viewer to control the code displayed in the source code viewer. The Explorer or the user can use a set of sliders to determine if loops should be filtered from the code view according to their loop depth, granularity and execution time. Filtered loops are shown in gray; unfiltered sequential loops are shown in black; unfiltered parallel loops are shown in white.

A white focus bar in the Codeview indicates that the `interf/1000` loop was selected as a good candidate for hand parallelization. The compiler cannot parallelize this loop because of a static dependence between the accesses of `RL`. Since `K` defines the region of the array `RL` accessed, the Explorer computes the array-restricted and code-region-restricted control slices of the references to `K` in statements 1125 and 1135 as shown in Figure 4-3. If the slice had not been array-restricted, there would be many more nodes in the slice that contribute to the value of the array `RS` referenced in statement 1109.

It is easy to see that each iteration potentially reads and writes `RL[6:9]`. If the read condition implies the write condition, then the array is privatizable. The control slice of the read operation on line 1135 includes all the statements shown except for the `DO 1130` loop. From the slice, the programmer sees that the array elements `RL[6:9]` are read only if `KC` equals 0, but `KC` equals 0 only if elements `RS[1:9]` are all less than or equal to `CUT2`. The control slice of the write operation on line 1125 includes all the statements shown except for the `IF` statement in line 1131. If elements `RS[1:9]` are all less than or

```
      DO 1000 I=1,NMOL1
           ...
           DO 1100 J=I+1, NMOL
                ...
                KC=0
                DO 1110 K=1,9
                     ...
1109                 IF (RS(K) .GT. CUT2) KC=KC+1
1110            CONTINUE
                IF (KC .NE. 9) THEN
                     ...
                     DO 1130 K=2,5
                          IF (RS(K+4) .LE. CUT2)THEN
1125                           RL(K+4)=...
                               ...
                          ENDIF
1130                 CONTINUE
1131                 IF (KC .EQ. 0) THEN
                          DO 1140 K=11,14
1135                           ...=...RL(K-5)...
                               ...
1140                      CONTINUE
                     ENDIF
                     ...
                ENDIF
1100       CONTINUE
1000  CONTINUE
```

**Figure 4-3. Slices of the relevant references to `K` in `interf/1000`**

equal to `CUT2`, then `RL[6:9]` will first be written before they are read. A programmer can easily make this inference once he is presented with the slices. While the Polaris compiler can privatize `RL` using special-purpose symbolic analysis and pattern matching[17], no automatic tool can handle every such case. Thus, we think this example is a fair demonstration of slicing as a general and effective mechanism for interactive parallelization.

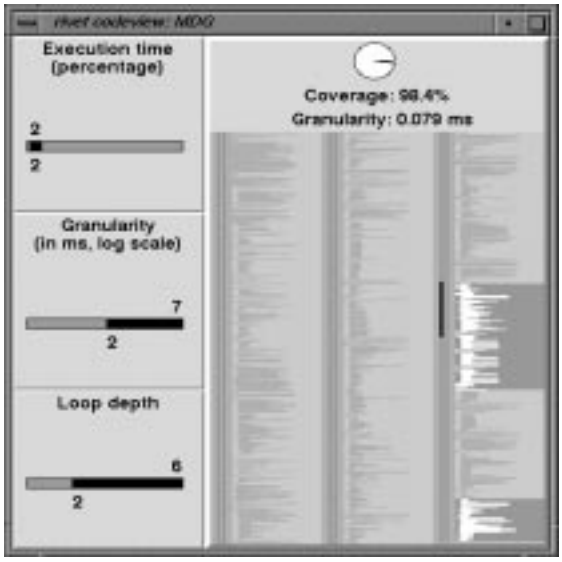


**Figure 4-4. Codeview of the optimized version of `mdg`, showing that `interf/1000` has been successfully parallelized.**

### 4.1.4. Parallelization with User's Input

Once the programmer asserts that the array `RL` is privatizable, the Assertion Checker confirms that the assertion is consistent with its data and adds the annotation to the code. The annotation enables the compiler to successfully parallelize the main loop in this program, as shown in Figure 4-4. As a result, the application achieves a 4-times speedup on 4 processors and a 6-times speedup on 8 processors in a Digital AlphaServer machine. Note that our automatic parallelizer alone produces no speedups on 4- or 8-processors.

This example illustrates that SUIF Explorer is effective in assisting the programmer in his or her parallelization task. By examining just a small fraction of the program, the user is able to supply a few assertions that enable the compiler to produce effective parallel code.

## 4.2. Case Study: `hydro`

In this section, we illustrate the experience of using the SUIF Explorer with the `hydro` program from Los Alamos National Laboratory. The program consists of 12,942 lines of code. It implements a 2-D Lagrangian hydrodynamics model.

### 4.2.1. Invoking the Parallelizer and Execution Analyzers

The SUIF Explorer starts by invoking the SUIF compiler to parallelize the `hydro` application and then runs the parallelized code on an 8-processor Digital AlphaServer. The parallelized application runs 2.7 times faster. The execution analyzers show that the parallelism coverage is 89% of the computation, and the average granularity for each loop is 0.3 millisecond.

### 4.2.2. Locating Targets of Parallelization

The Explorer then evaluates the compiler results and profile data to find the important sequential loops that are potentially parallelizable. Of seven such loops, we will use the loop labeled 85 in the `vsetuv` procedure (`vsetuv/85`) to illustrate the parallelization process. Section 4.3 and Section 4.4 provide the summary for all the loops. The loop `vsetuv/85` consists of 79 lines and four procedure calls. This loop accounts for 8% of the total execution time of the program. In addition, the Dynamic Dependence Analyzer does not report any loop-carried dependence for the user-supplied input set.

### 4.2.3. Highlighting the Relevant Slice to User

The compiler cannot parallelize the loop `vsetuv/85` because of the dependences on two arrays. Figure 4-5 shows the excerpt from this loop. We will examine only the array `dkrc` below, because the other array can be privatized in a similar way. The Explorer reports that `dkrc` is not privatizable statically. One reason is that the range of the array accessed in each iteration is loop variant. Thus, the compiler cannot determine which iteration contains the final value for each array element after the execution of the loop. However, oftentimes the values of the array elements are never used after the loop execution. We developed the array liveness analysis in Chapter 5 to handle this common case.

```
      DO 85 l=2,lmax
         k1=k_lower(l)
         k2=k_upper(l)
         IF (k1 .EQ. 0) GO TO 85
         k1p1=k1
         IF (k1 .EQ. 1) k1p1=k1+1
         k2p1=k2+1
         CALL fvsr
         ...
         DO 60 k=k1p1, k2p1
            dkrc(k)=...
            ...
60       CONTINUE
         DO 80 k=k1,k2
            ... = dkrc(k)+dkrc(k+1)
            ...
80       CONTINUE
85    CONTINUE
```

**Figure 4-5. Slices of the relevant references to `k` in `vsetuv/85`**

The other reason why `dkrc` is not privatizable statically is that the elements of `dkrc` are used for two different purposes. Based on the computation of k1 and k1p1, the programmer can infer that the elements `dkrc(2..n)` are privatizable, but there exists an upwards exposed read on `dkrc(1)` from its value outside the loop. Due to the conditional definition of `k1p1`, the SUIF compiler cannot express the value of `k1p1` in terms of `k1`. Hence, the compiler does not perform this inference, and the loop is reported as sequential.

#### 4.2.4. Parallelizing with User's Feedback

After the user examines the code, he or she decides that the loop `vsetuv/85` is parallelizable. The Assertion Checker confirms that the assertion is consistent with its data and adds the annotation to the code. The annotation enables the compiler to successfully parallelize this loop in this program. The SUIF Explorer parallelizes a total of 6 loops after the user provides 25 assertions on privatization.

Afterwards, the SUIF Explorer recompiles the program and improves the speedup from 2.7 to 4.3 on an 8-processor Digital AlphaServer. Our speedup is only about half of the perfect speedup, due to poor spatial locality and the data reshuffling overhead. Figure 4-6 illustrates the source of the memory performance problem. The loop `vqterm/85` does not have good spatial locality because the inner loop accesses the data by row, which is not contiguous in Fortran. Furthermore, the loops `vsetuv/85` and `vqterm/85` are parallel, but the data are distributed across the processors by column and by row, respectively. These conflicting data decompositions incur data reshuffling overhead. We manually apply array transposes and loop interchanges to some interprocedural non-perfectly-nested loops in order to both eliminate the conflicting decompositions and to make processor-data contiguous. As a result, we obtain a speedup of 5.9 times on an 8-processor Digital AlphaServer. The details of these memory optimizations are beyond the scope of this thesis. Interested readers are referred to [4] and [109] for the description of loop interchanges and array transposes. Notice that their algorithms[4][109] cannot handle the interprocedural non-perfectly-nested loops in the `hydro` program.

## 4.3. Size of Code Requiring Intervention

The SUIF Explorer is effective if it can present a small number of lines of code for the programmer to analyze, assuming it includes every line needed for intervention. The Explorer uses three concepts to filter out code not requiring attention: automatically parallelized loops require no attention, only important sequential loops require attention, and only the slices that contribute to static dependences require attention. We evaluate each of these aspects in turn below.

```
      SUBROUTINE vsetuv
      DO 85 l=2,lmax
         k1=k_lower(l)
         k2=k_upper(l)
         ...
         DO 60 k=k1p1,k2p1
            ...
            duac(k,l)=...
60       CONTINUE
85    CONTINUE

      SUBROUTINE vqterm
      DO 85 k=2,kmax
         l1=l_lower(l)
         l2=l_upper(l)
         CALL fvsr
         DO 80 l=l1+1,l2
            ...=duac(k,l)
80       CONTINUE
85    CONTINUE
```

**Figure 4-6. Source of the memory performance problem in `hydro`**

### 4.3.1. Automatic Parallelization

The statistics on the number of loops parallelized in the four programs automatically and manually are shown in Figure 4-7. The first row shows the total number of loops that are executed at least once for the given input data. The measurements are separated according to whether the loop calls other procedures — “inter” for loops that do and “intra” otherwise. The second row shows the number of loops that have not been parallelized by the compiler.

| Application | mdg | | arc3d | | hydro | | flo88 | | |
|---|---|---|---|---|---|---|---|---|---|
| Number of loops | inter | intra | inter | intra | inter | intra | inter | intra | Total |
| Executed | 4 | 39 | 14 | 269 | 11 | 92 | 121 | 216 | 766 |
| Sequential | 2 | 8 | 6 | 36 | 11 | 46 | 37 | 35 | 181 |
| Important | 2 | 0 | 6 | 5 | 9 | 0 | 0 | 14 | 36 |
| Important and no dynamic dependence | 2 | 0 | 6 | 5 | 7 | 0 | 0 | 13 | 33 |
| User-parallelized | 1 | 0 | 3 | 0 | 6 | 0 | 0 | 7 | 17 |
| Remaining important | 0 | 0 | 0 | 1 | 1 | 0 | 0 | 0 | 2 |

**Figure 4-7. Number of loops requiring user intervention**

We see that the compiler manages to handle about 80% of the loops, with a higher rate of success for loops that do not call other procedures. Altogether 181 loops are found to be sequential. Note that a sequential loop may be nested in another sequential loop and that it may contain parallelized inner loops.

### 4.3.2. Using Dynamic Information

Next, the Explorer identifies the important sequential loops that have sufficient coverage and granularity. The important loops are those whose coverage is larger than 2% and granularity is larger than 0.05 milliseconds. These cut-off numbers are parameterized and can be changed by the user. As shown in Figure 4-7, many of the sequential loops that have no procedure calls are found to be unimportant. The average code size of important loops with

procedure calls is over 100 lines, including called procedure(s) and excluding comment lines. There are only 36 loops found to be important, and only three of these are found to carry true dependences when the code was executed. Using the Explorer, the programmer found seventeen loops to be parallelizable. Because parallelizing an outer loop will also execute the inner loops in parallel, parallelizing these seventeen loops reduces the number of the important loops left to run sequentially to two.

### 4.3.3. Program Slicing

Finally, we show how slicing reduces the number of lines of codes that need to be examined by the programmer. Of the thirty-three important loops found to have no dynamic dependences, seventeen were parallelized by the programmer and two were attempted without success. The remaining fourteen loops were nested within the parallelized loops and thus needed no further attention. For each dependence, the Explorer needs to show to the programmer two sets of slices, one for each reference sharing the data dependence relationship. The average size of the combined slices for each dependence, measured as a percentage of the loop size, is reported for each of the 19 loops examined by the programmer in Figure 4-8.

The first column in the table gives the subroutine name and the label for each loop. The second column reports the number of lines in a loop, including those in the callees while excluding comment lines. Statistics are reported both for the program slices and control slices. Columns labeled “full” are the unrestricted slice sizes. Note the full program slice may include statements outside the loop and can be much larger than the number of lines in the loop. Programmers would mainly concentrate on statements inside the loop of interest, and hence, the columns labeled “loop” count only those statements in the full slice that are inside the loop.

The columns labeled “CR” report the sizes of the code-region-restricted slices. They are much smaller than those under the “loop” column, as they are computed by pruning the slice computation at the first nodes that leave the loop. In contrast, tracking nodes outside the loop in a full slice computation may lead to more nodes inside the loop. They represent dependences between multiple invocations of the same loop, which should not prevent the

| loop | No. lines | Program slice (%) | | | | Control slice (%) | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | full | loop | CR | AR | full | loop | CR | AR |
| `mdg` | | | | | | | | | |
| interf/1000 | 109 | 342 | 86 | 31 | 9 | 342 | 86 | 20 | 9 |
| `arc3d` | | | | | | | | | |
| filter3d/701 | 67 | 33 | 10 | 10 | 10 | 33 | 10 | 10 | 10 |
| stepf3d/701 | 261 | 8 | 2 | 2 | 2 | 7 | 1 | 1 | 1 |
| stepf3d/702 | 242 | 10 | 3 | 3 | 3 | 7 | 1 | 1 | 1 |
| stepf3d/801 | 260 | 13 | 2 | 2 | 2 | 7 | 1 | 1 | 1 |
| `hydro` | | | | | | | | | |
| update/1000 | 268 | 220 | 41 | 3 | 3 | 220 | 41 | 3 | 3 |
| vh2200/1000 | 45 | 1309 | 27 | 11 | 11 | 1309 | 27 | 11 | 11 |
| vqterm/85 | 51 | 1155 | 29 | 22 | 22 | 1155 | 29 | 22 | 22 |
| vsetgc/200 | 47 | 1253 | 36 | 23 | 23 | 1253 | 36 | 23 | 23 |
| vsetuv/85 | 79 | 746 | 37 | 11 | 11 | 746 | 37 | 11 | 11 |
| vsetuv/105 | 49 | 1202 | 29 | 10 | 10 | 1202 | 29 | 10 | 10 |
| vsetuv/155 | 130 | 453 | 46 | 6 | 6 | 453 | 46 | 6 | 6 |
| `flo88` | | | | | | | | | |
| psmoo/50 | 34 | 121 | 29 | 29 | 29 | 121 | 29 | 29 | 29 |
| psmoo/100 | 34 | 121 | 29 | 29 | 29 | 121 | 29 | 29 | 29 |
| psmoo/150 | 34 | 121 | 29 | 29 | 29 | 121 | 29 | 29 | 29 |
| eflux/50 | 117 | 38 | 9 | 9 | 9 | 38 | 9 | 9 | 9 |
| dflux/30 | 48 | 94 | 15 | 15 | 15 | 94 | 15 | 15 | 15 |
| dflux/50 | 50 | 84 | 16 | 16 | 16 | 84 | 16 | 16 | 16 |
| dflux/70 | 50 | 88 | 20 | 20 | 20 | 88 | 20 | 20 | 20 |
| Average | 104 | 390 | 26 | 15 | 13 | 389 | 26 | 14 | 13 |

**Figure 4-8. Average size of the slices requiring intervention, as a percentage of the loop size.**

parallelization of iterations within the loop. The last columns, labeled “AR”, represent slices with both the code-region and array restrictions.

Our results show that full program slices and control slices can be large in size. Code-region restriction is successful in reducing the slices to about 15% of the loop size on average. Applying the array restriction to the code-region-restricted slices reduces them from 31% to 9% in the `interf/1000` loop and has no effect on the other loops.

In summary, we have shown that the Explorer is successful in minimizing the size of the code that requires intervention. It focuses the programmer’s attention on 33 of the 766 executed loops in the three programs, based on static and dynamic analyses of the programs. The use of slicing requires the programmer to read only about 13% of the code in each loop to resolve a data dependence.

## 4.4. Cooperation Between the SUIF Explorer and the Programmer

Even for those loops that require user intervention, the SUIF parallelizer still plays an important role in reducing the user’s effort. Figure 4-9 shows the number of data structures analyzed automatically and analyzed manually in the seventeen loops parallelized with the programmer’s help. Ten of the seventeen loops contain procedure calls.

Arrays whose accesses in a loop do not create a loop-carried dependence are classified as parallel arrays in this table. The privatizable arrays and privatizable scalars are those arrays and scalar variables whose dependences are not carried from one iteration to another. Arrays and scalar variables whose updates are commutative are classified as reduction arrays and reduction scalars, respectively. Note that in our previous experience with the parallelizer, we have already found that parallelizing reductions is important to obtaining good performance. We have developed a powerful reduction analysis and integrated it into the parallelizer, as discussed in Chapter 6. All the experimental results in this chapter are obtained in the system that includes the reduction analysis.

Figure 4-9 shows that the programmers need to examine only 19 of the 766 loops in the programs; and for these 19 loops, the compiler automatically parallelizes 363 of the 426 variables used. The slicing algorithm requires the programmer to read only about 13% of

| | | mdg | arc3d | hydro | flo88 | Total |
|---|---|---|---|---|---|---|
| Automatic | Parallel arrays | 6 | 27 | 96 | 30 | 159 |
| | Privatizable arrays | 8 | 19 | 29 | 13 | 69 |
| | Privatizable scalars | 26 | 48 | 42 | 15 | 131 |
| | Reduction arrays | 3 | 0 | 0 | 0 | 3 |
| | Reduction scalars | 1 | 0 | 0 | 0 | 1 |
| | Total | 44 | 94 | 167 | 58 | 363 |
| User Input | Privatizable arrays | 1 | 9 | 25 | 25 | 60 |
| | Privatizable scalars | 0 | 3 | 0 | 0 | 3 |
| | Total | 1 | 12 | 25 | 25 | 63 |

**Figure 4-9. User-assisted parallelization of 17 loops in four applications**

the code in a loop to resolve a dependence. The compiler can automatically parallelize the references to many data structures within the sequential loop, leaving only a small amount of work to the programmer.

### 4.4.1. Analysis by the Programmer

This section identifies the user's analyses needed for the parallelization of the four applications. Starting with the main causes for failures in the automatic parallelization, we observe that the compiler successfully finds all 159 parallel arrays in these applications but finds only 69 of the 129 privatizable arrays. Array privatization requires the analysis of the flow of values, whereas finding parallel arrays requires only the analysis of locations. Therefore, it is not surprising that the compiler is not as successful with array privatization.

The compiler can identify almost all privatizable scalar variables and all of the reduction variables in these loops.

In the following, we discuss the reasons why the compiler fails to privatize the arrays automatically. In the case of `mdg`, the programmer needs to perform some inference on the control flow in order to privatize the array `RL` in Figure 4-3. The programmer first infers that the condition "`KC` equals 0" implies that the range of `RS[1:9]` is less than or equal to `CUT2`. Next, the user identifies the former as the read condition and the latter as the write condition; thus, the array `RL` is privatizable.

In the `arc3d` application, the user also needs to analyze the control flow to enable privatization. Consider the user-parallelized loop `stepf3d/701:`

```
DO 701 L = 2, LM
   DO 300 N = 3, 5
      ...
      if (N .eq. 3)
         SN = ...
      if (N .eq. 4)
         SN = ...
      if (N .eq. 5)
         SN = ...
```

The variable `SN` is initialized when `N` is 3, 4, or 5. The user observes that the initialization code covers the entire iteration space of the loop; thus, `SN` is privatizable. The other two user-parallelized loops have similar problems with control flow.

In the case of the loop `vsetuv/85` in `hydro`, the user needs to infer that the upwards exposed reads in one iteration are not defined in any previous iterations. As shown in Figure 4-5, the user can infer this situation by studying the conditional definition of `k1p1`. Although the programmer needs to privatize 25 arrays in `hydro`, the arrays are accessed in a similar manner and are subject to the same analysis.

In order to privatize the arrays in `flo88`, the user needs to know the relationships between the scalar variables which determine the range of the array accessed. These scalar variables

| | | mdg | arc3d | hydro | flo88 |
|---|---|---|---|---|---|
| Automatic | Coverage | 73% | 90% | 86% | 81% |
| | Granularity | 0.002 msec | 0.3 msec | 0.3 msec | 0.1 msec |
| | Speedup (4 processors) | 1.0 | 2.1 | 2.4 | 1.1 |
| | Speedup (8 processors) | 1.0 | 1.6 | 2.7 | 1.0 |
| With User Input | Coverage | 98% | 98% | 94% | 98% |
| | Granularity | 0.08 msec | 50.9 msec | 0.6 msec | 34.2 msec |
| | Speedup (4 processors) | 4.0 | 5.4 | 3.2 | 3.1 |
| | Speedup (8 processors) | 6.0 | 4.9 | 4.3 | 5.5 |

**Figure 4-10. Results of parallelization with and without user**

are initialized from the input file in the initialization routine. For example, the user needs to know the relationship between the scalar `IE` and the scalar `IL` (that is, `IE=IL+1`) in order to privatize the arrays in the loop `psmoo/50` in `flo88`.

In summary, the failures of the compiler in privatizing arrays are often due to the complex control flow or unknown input values. The latter cannot be determined at compilation time.

## 4.5. Performance Results

This section demonstrates how having the user analyze a relatively small number of lines of a code pays off tremendously. Our parallelizer generates an SPMD (Single Program Multiple Data) parallel C version of the program that can be compiled by native C compilers on a variety of architectures. Only the outermost loop that the compiler or the user has proven to be parallelizable is parallelized. Our parallelizer suppresses parallel execution if

the overhead involved is expected to overwhelm the benefits. The run-time system estimates the amount of computation in each parallelizable loop using the knowledge of the iteration count at run time, and runs the loop sequentially if it is considered too fine-grained to have any parallelism benefit. The iterations of a parallel loop are evenly divided between the processors at the time the parallel loop is spawned.

As shown in Figure 4-10, both the parallelism coverage and parallelism granularity improve significantly for each of the applications. The improvement in parallel performance is substantial for `mdg`, `arc3d`, and `flo88`, and is notable for `hydro`. The `mdg` benchmark improves from no speedup at all to a speedup of 6 times on 8 processors, `arc3d` improves from 1.6 to 4.9, `hydro` improves from 2.7 to 4.3, and `flo88` improves from 1 to a 5.5-times speedup.

The performance of `arc3d` degrades as the processors increase from 4 to 8 due to poor memory performance. To enhance spatial locality in several non-perfectly-nested loops, we apply loop interchanges manually and obtain a speedup of 10 times on 8 processors. Further memory optimizations such as alignment and padding on the user-assisted parallel code yield over an 11-times speedup on 8 processors. The details of the memory optimizations applied are beyond the scope of this thesis.

The speedup of `hydro` is only 4.3 on 8 processors due to such memory performance problems as poor spatial locality and data reshuffling overhead. Section 4.2.4 illustrates the problems. The speedup improves from 4.3 to 5.9 on 8 processors if these memory optimizations are applied.

The SUIF Explorer effectively helps the programmer to locate the coarse-grain parallelism in `flo88`. This results in substantial performance improvement. The speedup increases from 1.0 to 5.5 on 8 processors. Furthermore, the affine partitioning algorithm[80] and the array contraction discussed in Chapter 5 can increase the speedup from 5.5 to 6.4 on 8 processors.

## 4.6. Related Work

Several papers report on the effectiveness of existing parallelizing compilers on large applications[4][20][27][97] and of interactive tools[27][54]. Three of these papers suggest compiler-programmer interaction to parallelize programs[27][54][97].

Cheng and Pase use both the Cray Fortran compiler and ForgeExplorer[11] to parallelize 25 programs on an 8-processor Cray Y-MP[27]. Their paper offers two suggestions. First, a parallelizer should provide the user with the insight about what loops to parallelize, either through profiling or performance estimation. Second, a parallelizer should be able to query the user and use his or her feedback in analysis to eliminate dependences. The SUIF Explorer benefits from their suggestions and demonstrates that substantial speedups can be obtained. Most of the 25 programs parallelized by Cheng and Pase demonstrate a speedup of less than 2 on an 8-processor Cray Y-MP. However, it is difficult to make direct comparison between the two studies, because they use a different set of applications on a different machine.

Hall and her colleagues evaluate the effectiveness of the Parascope Editor (PED) on 8 Fortran programs[54]. They suggest the importance of providing both high-level analysis results and guidance in transforming programs. Their study does not provide any performance results. In comparison, the SUIF Explorer guides users with the interprocedural array data-flow analysis results. Furthermore, we demonstrate that the Explorer helps the users develop parallel codes effectively. We obtain substantial performance improvement on several real-world programs.

Singh and Hennessy examine the parallelization of three programs[97]. They observe that certain programming style, such as specialized user of the boundary elements in an array, interfere with compiler analysis. They suggest user access to profiling information and assertion facilities that allow specifying ranges of symbolic variables. The SUIF Explorer provides profiling information and more advanced analysis such as interprocedural symbolic propagation.

## 4.7. Chapter Summary

This chapter shows that SUIF Explorer is effective in assisting a programmer to find coarse-grain parallelism in several real-world programs. The case studies illustrate how the Explorer minimizes the lines of code requiring manual examination using three techniques: advanced interprocedural parallelization, sophisticated dynamic execution analyzers and program slicing.

The key to the Explorer's success in these case studies lies in having sufficiently powerful analyses that restrict the need for user assistance to a small number of lines of code. It is critical that the SUIF compiler parallelize many of the loops automatically and leave only a few unresolved dependences in the remaining sequential loops. As a result, the programmer only needs to examine a few variables for those loops examined in the case studies.

Furthermore, the case studies show that the concept of program slicing can be applied to interactive parallelization effectively. Our context-sensitive slicing algorithm is successful in reducing the number of lines that need to be analyzed while minimizing the likelihood of human error.

Finally, we study the remaining analyses required in the user-assisted parallelization. We show that the analysis by the user is reasonable and is usually only on a relatively small number of lines of codes. This user's analysis results in substantial performance improvement.

# 5 Array Liveness Analysis and Its Applications

It has been shown that many high-level interprocedural analyses and optimizations are needed to create code that effectively runs on a multiprocessor[53]. Detecting coarse-grain parallelism[51] requires interprocedural dependence analysis[24][51], privatization analysis, and reduction recognition for all scalar and array variables. In addition, symbolic analysis to detect linear relationships between scalar variables is useful in improving the precision of the array analyses. Also, various techniques have been developed to minimize communication and maximize cache locality by changing the execution order of the computation, mapping the computation intelligently across processors[10][26][47][69][80][96][109][110], and changing the data layout[9][16][67]. Through our experience with using SUIF Explorer, we discovered that several of these transformations and optimizations can benefit from *array liveness analysis*. Global liveness determines whether a variable is used after a certain point in the program. Array liveness analysis computes the global liveness information for each element of an array.

We will illustrate the importance of array liveness analysis with a real-world example. In parallelizing the hydro application from Los Alamos National Laboratory, we discovered that the loop `vsetuv/85` in Figure 5-1 is not automatically parallelized because the parallelizer did not know whether the array `aif3` is dead at the end of the loop. Furthermore, since each iteration of the `vsetuv/85` accesses different parts of the array `aif3`, the processor that executes the last iteration cannot simply finalize the values of the array. We need to implement the array liveness analysis in order to privatize `aif3` and parallelize the loop.

An array variable is often used to hold totally disjoint sets of data at different times in a program execution. In the case of Fortran programs, locations in a common block sometimes store data of even different dimensionalities and types. For instance, in Figure 5-1 the

```
      SUBROUTINE vsetuv
      DO 85 l = 2,lmax
         k1 = k_lower(l)
         k2 = k_upper(l)
         ...
         CALL init(aif3(k1), k2-k1+1)
         DO 60 k = k1p1, k2p1
            ...
            ... = aif3(k)
60       CONINUE
85    CONTINUE

      SUBROUTINE vqterm
      DO 115 k = 2,kmax
         i1 = i_lower(k)
         i2 = i_upper(k)
         CALL init(aif3(i1), i2-i1+1)
         DO 110 i = i1,i2
            ... = aif3(i)
110      CONTINUE
115   CONTINUE

      SUBROUTINE init(q, n)
      DO j = 1,n
         q(j) = ...
```

**Figure 5-1. Excerpt from hydro that illustrates the need for array liveness information**

variable `aif3` can be deduced to be dead at the entry of the `vsetuv/85` because any subsequent computation does not use any live array sections of `aif3`. Similarly, `aif3` is dead at the exit of the `vsetuv/85`. Hence, `aif3` in subroutines `vsetuv` and `vqterm` belongs to different live ranges. Array liveness analysis allows the compiler to recover the original semantics of the program by separating the live ranges of an array variable, thereby giving the compiler more freedom in transforming the code and the array data lay-

out. We will show in Section 5.1 that array liveness information can improve three different program optimizations: array privatization, data decomposition, and array contraction. As an enabler of several optimizations, the liveness information is effective in speeding up most of the applications from our study. Thus, the array liveness analysis is a key analysis that should be included in any modern parallelizing compiler.

Our proposed liveness analysis is a context-sensitive, flow-sensitive, interprocedural, region-based algorithm. This analysis uses a standard two-phase elimination-style algorithm[87][95], where a bottom-up phase summarizes the effect of each interval or region and a top-down pass propagates the actual value to the different points in the code. The array sections are represented as sets of systems of linear inequalities. The bottom-up phase is an enhanced version of the bottom-up array data-flow analysis in the SUIF compiler[51]. In particular, the analysis is more accurate for code with a simple array access pattern involving recurrences. While the bottom-up phase answers the question of whether there are any exposed uses *in* the region, the top-down phase determines at the end of a region whether there are any exposed uses in the rest of the execution. We implemented our proposed liveness analysis in an interprocedural region-based analysis framework[51][55] in SUIF[106]. We found our algorithm to be both precise and efficient across a suite of real-world applications, despite the fact that potentially exponential algorithms are used in manipulating the array sections.

Because array liveness analysis was deemed to be too expensive[104], we also compare our technique with two other cheaper and less accurate analyses. We show that our proposed algorithm is reasonably efficient and can uncover many more dead variables across the benchmark suite.

This chapter makes the following contributions:

- We identify several uses of array liveness. They are array privatization, data decomposition, and array contraction.
- We experiment with large programs and show the benefits of the analysis for three different uses.

- We design an array data-flow analysis for the liveness problem and show that fast liveness analysis on arrays can be achieved without sacrificing precision.

- We evaluate an array data-flow analysis for the liveness problem and show that fast liveness analysis on arrays can be achieved without sacrificing precision.

The chapter is organized as follows. Section 5.1 describes the uses of array liveness analysis. We present our algorithm in Section 5.2 and evaluate it in Section 5.3. Next, we apply the algorithms to privatization, data decomposition, and array contraction in Section 5.4, 5.5, and 5.6, respectively. We discuss the related work in Section 5.7 and summarize the chapter in Section 5.8.

## 5.1. Uses of Liveness Analysis

Array data-flow analysis[82][83][84] determines whether the value of an array element update is used by another read operation; whereas the more commonly used array data dependence analysis determines only whether two operations use the same locations. Feautrier shows that for programs, whose loop bounds and array indices are all affine expressions of outer loop indices and symbolic constants, parametric integer programming[39] can be used to identify the precise instance of a write operation that generates the value read in each iteration of a loop[40][41]. Unfortunately, the algorithm is too expensive to apply across all the procedures in the whole program, which is necessary if we wish to find coarse-grain parallelism. Instead of finding the def-use relationships between all pairs of accesses, our strategy is to examine how liveness can benefit our program optimizations and design algorithms that generate the necessary information effectively.

Liveness on scalar variables is well known and used in many optimizations and transformations. Data-flow analysis on arrays has intrinsically higher space and time requirements than analysis on scalars, as it is necessary to keep track of accesses to individual array elements. As we continue to develop more and more advanced systems that operate on high-level program constructs and manipulate aggregate array data structures, we expect that liveness on array variables will also find many uses. Fundamentally, array liveness increases our accuracy in analyzing the def-use chain between array accesses. For exam-

ple, general analyses, such as slicing[78], will be stronger if array liveness is available. Below, we discuss three specific optimizations in the parallelizing compiler domain that can benefit from array liveness. Experimental results on these optimizations will be presented in Sections 5.6, 5.7, 5.8, respectively.

### 5.1.1. Array Privatization

Array data-flow analysis has been used in array privatization, a technique critical to the success of automatic parallelization. Many coarse-grain loops reuse the same working array in different iterations; thus there is no parallelism unless each processor is given its own private copy of the array. Instead of computing the perfect data flow information, existing algorithms tend to use approximate algorithms to detect the common opportunities for array privatization[51][104]. Inaccuracies are introduced in two places. The first is in the calculation of the upwards-exposed read section of a loop. It is computed by simply finding the closure of the upwards-exposed read of each iteration, while not accounting for the write operations that may precede the read operations in different iterations of the same loop. Thus, this calculation over-estimates the area that is upwards exposed. The second place where inaccuracy is introduced is in the handling of array *finalization*. The idea is that the final values must be written back into the original array if they are read after the loop was executed. The SUIF compiler does not compute any liveness information as such; instead, it simply tests if every iteration must write to exactly the same region of the array. If so, only the values produced in the last iteration can be live after the loop, so the processor assigned to execute the last iteration of the loop uses the original data array as its own private copy. Our experience with the SUIF compiler reveals that the latter of the two inaccuracies causes the compiler to miss many important array privatization opportunities. Thus, finding array liveness is useful in increasing the applicability of array privatization.

### 5.1.2. Array Layout Optimizations

It has been shown that changing the layout of an array can improve the performance of a multiprocessor as well as the cache performance of a uniprocessor. For example, by laying out the data so that consecutive data is accessed by the same processor, spatial locality is maximized, while minimizing false sharing. However, data layout changes require that all

uses of the same location be identified and updated accordingly. In the case where accesses to the same array in different parts of a program can benefit from different data layouts, the compiler must either choose to restructure the array dynamically or execute parts of a program suboptimally. Anderson shows that one reason for conflicting access patterns is that the same array or common block is used for totally disjoint purposes at different times of a program execution[8]. The availability of array liveness allows the compiler to split the live ranges of these uses and eliminate the artificial conflict in the data access patterns, thereby improving the overall performance.

### 5.1.3. Array Contraction

A lot of scientific code in use was developed and optimized for vector computers. These programs tend to have many small loops, with many temporary arrays holding the result of the computation. Such code does not perform well on multiprocessors with caches, as it tends to fill the cache with data that is flushed out of the cache before they are reused. It is thus useful to fuse the loops together so that data is consumed as it is produced. Once all the uses of the data are co-located with the producer, there is no need to hold all the temporary results in an array. Instead, we can replace the array with a scalar variable. However, this optimization requires that there are no other uses of this data. This, again, requires array liveness information.

## 5.2. The Interprocedural Array Liveness Algorithm

Our array liveness analysis is a two-phase, bottom-up and top-down, region-based interprocedural analysis[51]. A *region graph* is a hierarchical program representation where every procedure, loop, and loop body in the program is represented as a region. The *edges* connect a region to its subregions, i.e. from callers to callees, and from code representing an outer scope to that of an inner scope. Our algorithm currently does not handle recursion; thus the region graph is simply a DAG (directed acyclic graph). Recursion can be handled by applying a fixed point calculation to each of the strongly connected components in the call graph in both the bottom-up and top-down phases.

A region-based analysis summarizes the side effects during the bottom-up traversal of the region graph and then propagates down the execution context during the top-down traversal. The region-based analysis computes *context-sensitive* interprocedural information efficiently and accurately. Unlike common iterative approaches, it avoids the inaccuracies of *unrealizable paths*[73], where information from one caller propagates to another caller of the same function. The analysis of a region may be either *flow-insensitive*, which ignores the control flow in the code, or it may be *flow-sensitive*. It is a forward-flow analysis if the information flows in the direction of the control flow edges, or it is a backward-flow analysis if the information flows in the reverse direction.

### 5.2.1. Representation of Array Sections

Array data-flow analysis has higher space requirements than analysis on scalar variables, as it is necessary to keep track of accesses to individual array elements. Thus, the challenge to the compiler is to keep precise enough information for the array analysis while maintaining efficiency.

Our array data-flow analysis keeps multiple summaries per array for each program region, and merges the summaries when no information is lost by doing so. One array summary consists of a four-tuple, $<R, E, W, M>$, where $R$ is all of the array sections that may have been read, $E$ is all of the upwards-exposed read array sections, $W$ is all of the may-write array sections, and $M$ is all of the must-write array sections. Note that $W$ and $M$ are disjoint. Each array section is represented by a set of systems of integer linear inequalities, whose integer solutions determine array indices of accessed elements. The denoted index tuples can also be viewed as a set of integral points within a convex polyhedron. The accessed region of an array is represented as a set of such polyhedra. We create for each array access an *array section descriptor*[4], which consists of the set of linear inequalities that define the polyhedron.

All the analyses are formulated as operations on these systems of linear inequalities[4]. This representation accurately represents all affine array index expressions. A non-affine index in a dimension is replaced by a conservative approximation: the entire dimension may be accessed. The intersection operator used on the array sections also introduces

imprecision, but it is conservative and avoids expensive calculations that produce excessively large results[4]. The system of linear inequalities, representing a data access, are derived from the loop bounds and the array indices in the program. Since only linear relationships with respect to loop indices and symbolic variables in the loop are accurately represented, our algorithm applies an interprocedural symbolic analysis to find linear relations between scalar variables before the array analysis so as to increase the precision.

Many previous approaches to array data-flow analysis manage the costs of representing array sections by summarizing all the accesses to an array with a single array summary that represents the conservative approximation of all the accesses. While it is space efficient, such an approach may lose precision if there are very different accesses to the array in a loop, or if the accesses are spread across multiple loops nested inside outer loop. Our representation is more powerful than previous work based on regular sections[25][77] and less expensive than attempts using convex hulls[100].

### 5.2.2. The Proposed Algorithm

Our interprocedural array liveness algorithm is a region-based backward-flow analysis. It has two phases. Phase 1 analyzes the region graph in a bottom-up manner, starting with the leaf regions. For each region, a flow-sensitive analysis is used to compute a summary of the array access information at the beginning of each subregion to the end of the parent region. The summary for the entire region is simply the summary information from the start of the entry node to the exit node in the control flow graph. Then, this information is used to calculate the summary for its parent region and so forth. Phase 2 finds the summary information from the end of a region to the end of a program. In a top-down fashion, this phase propagates the initial liveness information at the end of a region to all its subregions by applying all the transfer functions computed in Phase 1.

#### 5.2.2.1. The Bottom-Up Phase

An outline of the bottom-up phase is given in Figure 5-2. In the bottom-up phase, a region is not analyzed until all its subregions are analyzed. We assume in the following that every region $r$ has a single entry and a single exit. Each region is represented as a four-tuple ($N$,

**for** each procedure $P$ from leaf procedures to main (bottom to top) **do**

  **for** each region $r = (N, A, s, e)$ from innermost to outermost of $P$ **do**

    $V_e = <\emptyset, \emptyset, \emptyset, \emptyset>$

    **for** each node $n \in N$ in reverse topological order **do**

      $V = \bigwedge_{(n,n') \in A} V_{n'}$

      **if** $n$ is a loop or a call-site

        $S_{r,n} = V$

      $V_n = T(V, FindSummary(n))$

    **if** $r$ is a loop node

      $S_r = Closure(V_s)$

    else

      $S_r = V_s$

**Figure 5-2. The bottom-up phase of the array liveness analysis**

$A, s, e$) where $N$ is a set of nodes representing either a basic instruction or a subregion, $A$ is the set of control flow edges, $s$ is the start node, and $e$ is the end node. The array summary of the exit node is initialized to a four-tuple of empty sets.

The analysis visits the nodes in the region in a reverse topological order, starting with the exit node. For each node, we apply the following steps:

1. If a node $n$ has multiple successors, the summaries at the beginning of the successor nodes are first combined by using the meet operator $\wedge$:

   $<R_1, E_1, W_1, M_1> \wedge <R_2, E_2, W_2, M_2> = <R_1 \cup R_2, E_1 \cup E_2, W_1 \cup W_2, M_1 \cap M_2>$.

The result $S_{r,n}$ represents a summary of the array accesses from the end of node *n* to the exit node of the immediately enclosing region *r*. The result $S_{r,n}$ is saved only if the node is a procedure call or a loop region that will be used in the top-down phase.

2. The next step is to calculate the effect of the node itself using the function *FindSummary*. If the node is a simple instruction, the function simply extracts the array access information from the source program. All the bounds and linear relationships on variables used in the access functions make up the linear constraints of the read and write array sections. If the node is a procedure call, then the summary of the region representing the procedure is mapped from the callee space to the caller space. If the formal array parameters are declared differently from the actual array parameters, the array sections are reshaped across the procedure boundaries[4]. If the node is a region itself, the function simply returns the summary that has already been computed for the subregion.

3. The results from Step 1 ($<R, E, W, M>$) and Step 2 ($<R_n, E_n, W_n, M_n>$) are then composed to give the summary at the beginning of the node using the transfer function
$T(<R, E, W, M>, <R_n, E_n, W_n, M_n>) = <R_n \cup R, E_n \cup (E - M_n), W_n \cup W, M_n \cup M>$.

After all of the nodes have been visited, we compute the summary for the entire region *r*, $S_r$ . If the region is not a loop, then the summary is simply the flow value of the start node. Otherwise, the summary is the closure of the summary of the loop body subregion; the closure operator projects away the loop index variable in each of the four array sections in the tuple. This closure operator introduces some imprecision because it does not use any must -write sections from different iterations to eliminate the upwards-exposed read sections. Section 5.2.2.3 describes our algorithm for improving the precision of upwards-exposed read sections.

#### 5.2.2.2. The Top-Down Phase

The top-down phase of the analysis calculates for each region *r*, $S_{r0,r}$, the summary of the accesses from the end of the region *r* to the end of the program. (The end of a program is the same as the end of the topmost region $r_0$.) From this summary, we compute $L_r$, array

$S_{r0,r0} = <\emptyset, \emptyset, \emptyset, \emptyset>$

**for** each procedure *P* from main to leaf procedures (top to bottom) **do**

$S_{r0,P} = \bigwedge_{n} MapToCallee(S_{r0,n}))$, where *n* is any call-site that calls *P*

**for** each region $r = (N, A, s, e)$ from outermost to innermost of *P* **do**

**if** *r* is a loop body

**Let** $S_{r0,Parent(r)} = <R_1, E_1, W_1, M_1>$ and $S_r = <R_2, E_2, W_2, M_2>$

$$S_{r0,r} = <R_1 \cup R_2, E_1 \cup E_2, W_1 \cup W_2, M_1>$$

**else**

$$S_{r0,r} = T(S_{r0,Parent(r)}, S_{Parent(r),r})$$

**for** each call-site node $n \in N$ **do**

$$S_{r0,n} = T(S_{r0,Parent(n)}, S_{Parent(n),n})$$

**Let** $S_{r0,r} = <R_1, E_1, W_1, M_1>$ and $S_r = <R_2, E_2, W_2, M_2>$

$$L_r = E_1 \cap (W_2 \cup M_2)$$

**Figure 5-3. The top-down phase of the array liveness algorithm**

sections that are both (1) written in the region and (2) live after the region. The algorithm is shown in Figure 5-3. Note that the function *Parent* takes the region as input and returns the region's enclosing loop body or procedure.

The array summary of the exit node of the topmost region is initialized to a four-tuple of empty sets. This phase visits the procedures in a top-down manner. By the time a procedure is visited, the summary for all of the sites that call the procedure would have been computed. These summaries which describe the code region following the call-sites and ending at the exit of the program are mapped to the callee space by using the *MapToCallee* func-

tion, which also reshapes the array if the type declarations are different. The *meet* operator is applied, combining all the mapped summaries, and calculating the summary from the end of the procedure to the end of the program.

The algorithm then visits the regions in the procedure starting from the outermost one and working its way inwards. If the region is a loop body, its execution may be followed by zero or more iterations of the same loop body, which is then followed by code after the enclosing loop. Thus, the exposed read is simply the union of the array sections exposed in the loop body and those exposed after the loop body. Otherwise, the desired summary is a composition of the summary from the end of its parent region to the end of the program and the summary from the end of the region to the end of its parent region. We compute the summary at the end of each call site in a similar manner.

Finally, the array sections written by the region that are live afterwards are simply the intersection of the exposed array sections at the end of the region and the array sections that may or must have been written in the region.

#### 5.2.2.3. Improving the Precision of Upwards-Exposed Read Sections

The upwards-exposed read sections as described are not precise, because the closure operator simply combines all the upwards-exposed read sections in all the iterations. A precise algorithm can reduce the exposed set by removing all those elements that are written in the loop before they are read.

Consider the code excerpt from the `flo88` application in Figure 5-4. In this code, `psmoo/20` writes to the array `d(1, 2:jl)`. `Psmoo/30` reads the data written by `psmoo/20` as well as the data written by `psmoo/30` itself in an earlier iteration (that is, `d(2:il-1,2:jl)`). This means that there are no upwards-exposed reads to the array `d` in the body of `psmoo/50`; thus the array is privatizable.

However, our closure operator does not take any write operations from different iterations of the same loop into consideration; the upwards-exposed read section is thus `d(2:il-1,2:jl)`. As the exposed section has a non-empty intersection with the written section, our algorithm, as described above, cannot determine that the array is privatizable.

```
      SUBROUTINE psmoo
      DO 50 k=2,kl
         ...
         DO 20 j=2,jl
            d(1,j) = 0
20       CONTINUE
         DO 30 i=2,il
            DO 30 j=2,jl
               t(i,j) = d(i-1,j)*...
               d(i,j) = t(i,j)
30       CONTINUE
         ...
50    CONTINUE
```

**Figure 5-4. Code excerpt from the `flo88` program**

This example shows an important weakness in our calculation of upwards-exposed read sections, since recurrences such as those in the above example are very common. We have thus enhanced our algorithm as follows. For loops that do not make any procedure calls, besides finding summaries of the array accesses, we also run the data dependence test. If the loops do not contain any anti-dependences between reads and writes of an array variable, and all of the writes are “must-writes” and not conditionally executed, then all of the write operations must precede any reads to the same location. Under these circumstances, we can subtract the written section from the upwards-exposed section of the array. This optimization succeeds in recognizing that the array `d` in the code segment in Figure 5-4 is privatizable.

### 5.2.3. Trading off Precision for Efficiency

Our algorithm above uses array section operations both in the bottom-up and top-down phases. Since operations on array summaries use the potentially exponential Fourier-Motzkin method[32][33][39], other researchers have suggested that using a simpler algorithm

that is more efficient may suffice[104]. Below, we propose two variants of the top-down phase that are cheaper but less precise.

#### 5.2.3.1. Reducing the Array Summaries to One Bit

In this algorithm, the top-down phase uses only one bit to indicate whether an array has any upwards-exposed reads. The rationale for this fast version is that temporary arrays are often not upwardly exposed beyond the region in which they are accessed. If we are only interested in finding temporary arrays, a single bit in the top-down phase suffices to represent this information. Our goal is to evaluate whether this approximation, which eliminates the need of Fourier-Motzkin operations in the top-down phase, is much faster and whether it is adequately precise. The results are presented in Section 5.3.

In this algorithm, the bottom-up phase is the same as the precise algorithm, except that the summaries for loops and call sites are represented only by a single bit per variable. The union operator in the top-down phase, as shown in Figure 5-3, is simply the logical-or operation and the intersection operator is the logical-and operator. The transfer function $T$ is defined as

$$T(<R_1, E_1, W_1, M_1>, <R_2, E_2, W_2, M_2>) = <R_1 \cup R_2, E_1 \cup E_2, W_1 \cup W_2, M_1 \cup M_2>.$$

With the single-bit approximation, the summary either indicates an empty set or the potentially entire set; thus there is no longer a subtraction (i.e., *kill*) operator in the transfer function $T$.

#### 5.2.3.2. A Flow-Insensitive Algorithm

To evaluate the amount of flow-sensitivity needed, we further simplify the one-bit algorithm to derive a *flow-insensitive* algorithm. It is flow-insensitive only in the top-down pass; that is, the control flow among subregions, belonging to the same region, is ignored in the top-down phase. In this algorithm, the bottom-up phase only uses one bit per variable per region to record whether the variable is exposed in the region. That is, we do not need to record any of the $S_{r,n}$ values. In the top-down phase, a variable is live at the end of a

region if it is live at the end of its parent region or if it is upwards exposed in one of its siblings, including itself. That is,

$$S_{r0,r} = S_{ro,Parent(r)} \cup \left( \bigcup_{r' \in Sibling(r)} S_{r'} \right)$$

## 5.3. Evaluation of the Algorithms

The goal of this section is to evaluate the trade-off between efficiency and precision among the algorithms described above. We evaluate the array liveness analysis by presenting the empirical data gathered from five programs: `hydro` from Los Alamos National Laboratory, `flo88` from the Center for Integrated Turbulence Simulations at Stanford, `arc3d` from NASA Ames Research Center, `wave5` from the SPEC95 benchmark, and `hydro2d` from the SPEC92 benchmark. Figure 5-5 presents the high-level information about the programs including the description and the number of lines of code in each program. The largest program, `hydro`, consists of about 13,000 lines of code.

| | Program description | No. of lines |
|---|---|---|
| `hydro` | 2-D Lagrangian hydrodynamics | 12942 |
| `flo88` | Wing-body analysis solving transonic flow | 7379 |
| `arc3d` | 3-D Euler equations solver | 4053 |
| `wave5` | Maxwell's equations and particle equations of motion | 7764 |
| `hydro2d` | Astrophysical program using Navier Stokes equations | 4461 |

**Figure 5-5. Program information**

### 5.3.1. Evaluating the Efficiency

Our liveness analysis requires two passes over the procedures in the program. The information collected in the bottom-up phase is also used to determine whether the variables can be privatized, whether the variables incur dependences, and whether the loops are parallelizable. The liveness analysis therefore only adds the top-down phase to the compilation. While the bottom-up phase analyzes every single statement in the program, the top-down pass only needs to compute the array summary for each region and each call site in the program.

Figure 5-6 reports the total running time of the interprocedural analysis on a single processor of a 300-MHz AlphaServer. The base version includes a scalar mod/ref analysis, a symbolic analysis to determine linear relationships between scalar variables, and the scalar liveness analysis. The bottom-up version includes the bottom-up array summary pass and the parallelization pass. The last three columns denote the three algorithmic choices of liveness analysis: the flow-insensitive version, the 1-bit version, and the full version. As the code in our research compiler has not been tuned for speed, the entire parallelization process takes up to 9 minutes on some of the programs, with the two-phase array analysis taking less than half the time. Note that `flo88`, `hydro2d`, `wave5` have many small nested loops, so the top-down phase takes relatively longer.

Figure 5-6 also shows that the one-bit algorithm is not much faster than the full algorithm. Although our implementation is not fine-tuned, the full algorithm runs, at the most, two minutes longer than the one-bit version.

### 5.3.2. Evaluating the Precision

We say that a variable is *dead* with respect to a loop, if the variable is written in the loop but not used after the loop boundary. As our goal is loop parallelization, we measure the precision of the algorithm by the number of dead array variables found at loop boundaries. An algorithm is more precise if it finds more dead variables. The first three columns in Figure 5-7 show the program names, the number of loops in each program, and the number of array variables modified in loops, respectively. The last three columns report the per-

| | base | bottom-up | top-down | | |
|---|---|---|---|---|---|
| | | | flow-insensitive | 1-bit | full |
| `hydro` | 59 | 78 | 81 | 82 | 89 |
| `flo88` | 308 | 421 | 436 | 437 | 548 |
| `arc3d` | 60 | 99 | 106 | 106 | 113 |
| wave5 | 274 | 410 | 443 | 444 | 508 |
| `hydro2d` | 22 | 35 | 39 | 38 | 64 |

**Figure 5-6. Total running time of the interprocedural analysis (in seconds) on a 300-MHz AlphaServer**

centage of the number of modified variables found dead at the end of loops by the full algorithm and its two simpler versions.

The flow-insensitive algorithm misses many dead variables found by the more precise algorithms, and the full algorithm also finds significantly more dead variables than the 1-bit version in most of the programs. Furthermore, in the following sections on the applications of the liveness analysis we will show that the precise algorithm can help parallelize more loops and find better data decompositions. Thus, we advocate the adoption of the most precise liveness analysis on arrays.

## 5.4. Application to Privatization

We have identified several uses of array liveness. In this section, we present the use of liveness information for the *finalization* of private arrays. If an array is live on exit of the loop, then after a parallel execution of the loop, the array must contain the same values as those obtained if the loop was executed sequentially. That is, the array needs to be *finalized*. The

| | #loop | #mod | %dead | | |
|---|---|---|---|---|---|
| | | | flow-insensitive | 1-bit | full |
| `hydro` | 122 | 416 | 47% | 70% | 72% |
| `flo88` | 421 | 845 | 18% | 39% | 46% |
| `arc3d` | 384 | 837 | 17% | 37% | 43% |
| `wave5` | 361 | 668 | 3% | 22% | 32% |
| `hydro2d` | 155 | 287 | 1% | 5% | 18% |

**Figure 5-7. Numbers of loops, modified variables in loops, and percentage of modified variables dead at loop exits**

previous version of the SUIF compiler does not test whether arrays are live on exit. Privatization is limited to those cases where every iteration in the loop writes to exactly the same region of data. The processor executing the last iteration writes to the original array, whereas all other processors write to their private copies of the array.

If liveness information is available, only the section of array that is written in the loop and live on exit of the loop needs to be finalized. If none of the written data is live, no finalization is needed. Therefore, we do not need to limit privatization to those variables whose written section is identical for every iteration.

We applied our analysis to the five Fortran programs. Figure 5-8 shows the number of privatizable arrays found to be dead at loop exits, the number of extra loops parallelized, and the resulting speedup on a 4-processor AlphaServer. The speedup for `flo88` was obtained on a 4-processor SGI Origin instead, due to a bug in Digital's system software. The details of the Origin system are given in Figure 6-1. Note that unless otherwise noted,

| | base | flow-insensitive | | | 1-bit | | | full | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | sp. up | #dead priv. | #par. loop | sp. up | #dead priv. | #par. loop | sp. up | #dead priv. | #par. loop | sp. up |
| `hydro` | 2.4 | 25 | 5 | 3.1 | 31 | 8 | 3.3 | 31 | 8 | 3.3 |
| `flo88` | 1.0 | 88 | 5 | 1.3 | 93 | 5 | 1.3 | 97 | 5 | 1.3 |
| `arc3d` | 2.0 | 58 | 8 | 2.1 | 79 | 13 | 2.4 | 83 | 13 | 2.4 |
| `wave5` | 1.0 | 0 | 0 | 1.0 | 15 | 9 | 1.0 | 19 | 12 | 1.0 |
| `hydro2d` | 2.6 | 0 | 0 | 2.6 | 0 | 0 | 2.6 | 0 | 0 | 2.6 |
| Total | — | 171 | 18 | — | 218 | 35 | — | 230 | 38 | — |

**Figure 5-8. Number of private arrays that are dead at exit, number of improved parallel loops, and the resulting speedup on four processors**

the parallel versions in Chapter 5 and 6 were generated completely automatically with no source code changes to the standard versions of these programs. The base version uses the SUIF interprocedural parallelizer without any array liveness analysis.

Indeed there are many private arrays that are dead at loop exits. Our liveness analysis finds 230 such arrays and improves the performance of 38 parallel loops in the four programs in Figure 5-8. The speedup of `hydro` improves from 2.4 to 3.3 on a 4-processor Digital AlphaServer, `flo88` improves from 1.0 to 1.3, and `arc3d` improves from 2.0 to 2.4. The twelve newly parallelized loops in `wave5` are small, and hence their parallelization is suppressed. Thus, the performance of `wave5` does not improve. Out of the 287 arrays modified in the 155 loops in `hydro2d`, 52 arrays are dead at loop exits, but none of them are private arrays. As a result, our liveness analysis does not help parallelize more loops in `hydro2d`.

Figure 5-8 also reports the results of using the flow-insensitive algorithm and the 1-bit algorithm. Although the 1-bit algorithm found fewer dead variables, the algorithm still works well in many cases because of the precise upwardly exposed array summaries in the bottom-up phase. However, the full algorithm is more precise and may result in better speedups in general.

## 5.5. Application to Data Decomposition

Memory optimizations, such as data decomposition[10][23], are critical to achieving high performance on parallel machines. However, finding a good data layout for a parallel program is challenging, as finding optimal dynamic decompositions is NP-complete. The emphasis of the decomposition algorithm in SUIF is on finding static decomposition regions that are as large as possible, because a static decomposition incurs no data reorganization[8]. For each static decomposition region, the algorithm proceeds to find the single optimal decomposition with the maximum degree of parallelism. The algorithm may decide to sacrifice some degrees of parallelism, ensuring that an array has no conflicting decompositions throughout the static decomposition region. Liveness information can eliminate artificially conflicting decompositions by allowing separate decompositions in separate live ranges of an array.

A common block variable in the Fortran program may have different shapes. The aliases among different shapes often result in false interferences. Liveness analysis can eliminate such interference and allow the data decomposition algorithm to obtain better results. Specifically, we use the liveness information to split up the Fortran common block variable into disjoint variables.

Consider an excerpt from `hydro2d` as shown in Figure 5-9. We observe that the common block `varh` is referred to as `vz` in subroutines `tistep` and `vps`, and referred to as `vz1` in subroutines `trans2` and `fct.` The variables `vz` and `vz1` are of different types. From the code, we see that the live ranges of the two variables are disjoint: `trans2` writes `vz1` which is then read by `fct`, and `vps` writes `vz` which is then read by `tistep` in the next iteration. Knowing that the live ranges are disjoint allows the variables to have different layouts.

```
      PROGRAM hydro2d
      DO 100 icnt=1,istep
         CALL tistep
         CALL advnce
         CALL check
100   CONTINUE

      SUBROUTINE tistep
      COMMON/varh/vz(mp,np)...
      DO j
       DO i
         =vz(i,j)

      SUBROUTINE advnce
      CALL trans2
      CALL fct

      SUBROUTINE trans2
      COMMON/varh/vz1(0:mp,np)...
      DO j
       DO i
         vz1(i,j)=

      SUBROUTINE fct
      COMMON/varh/vz1(0:mp,np)...
      DO j
       DO i
         =vz1(i,j)

      SUBROUTINE check
      CALL vps

      SUBROUTINE vps
      COMMON/varh/vz(mp,np)...
      DO j
       DO i
         vz(i,j)=
```

**Figure 5-9. Code excerpt from the `hydro2d` program**

Detecting whether the live ranges of any two variables in the same common block are disjoint is simple, once liveness information is available. The live ranges of two variables are disjoint if their live ranges are disjoint in every code region considered in isolation, and if no code region writes into an array section that overlaps with any of the live sections of the other variable at the end of the code region.

Our proposed algorithm is strong enough to determine that `vz` is dead at the end of `trans2` and `vz1` is dead at the end of `vps` for the above example. This enables the decomposition analysis to treat the two variables as unaliased. On the other hand, finding that the variable `vz` is upwardly exposed at the beginning of the loop body of Loop/100, the weaker top-down phases cannot tell that the subroutine `vps` kills the live section of `vz`; therefore the weaker top-down phases conclude that the two variables may have overlapping live ranges. This example illustrates the need for our proposed algorithm.

The common blocks in three of the programs in our benchmark suite have aliased variables of different types. Of these, our full algorithm is able to separate the live ranges of five common blocks in `hydro2d`[1] and one each in `arc3d` and `wave5`, as shown in Figure 5-10. The figure also shows the resulting speedup before and after the optimization. The performance of `arc3d` does not improve because the old privatization algorithm manages to privatize the arrays in the common block without the liveness information, since every iteration writes to the same locations. Also, the code does not benefit from changing the layouts of arrays. The performance of `wave5` does not improve because the common block consists of one-dimensional arrays. `hydro2d` improves slightly with the separation of the live ranges.

## 5.6. Application to Array Contraction

Array contraction is a transformation that converts an array variable into a scalar variable by mapping multiple array elements to the same location. This is legal when the live ranges of the individual elements of the array do not interfere with each other. We generalize this

1. Two of the live ranges in `hydro2d` can be separated only if the compiler knows that the input value of the variable `mqflag` is 0. This information is manually inserted into the system.

| | #common block splits | speedups before splits | speedups after splits |
|---|---|---|---|
| `arc3d` | 1 | 2.4 | 2.4 |
| `wave5` | 1 | 1.0 | 1.0 |
| `hydro2d` | 5 | 2.6 | 2.8 |

**Figure 5-10. Number of common block splits and the resulting speedup on a 4-processor AlphaServer**

notion of array contraction by allowing arrays to be mapped to not only scalar variables but also arrays of lower dimensionality.

Many vector codes and programs written in array languages such as Fortran 90 and HPF can benefit from array contraction[26][76][94]. To aid vectorization, the vector codes often consist of many loops with only a small amount of computation in each iteration, so that they can be mapped into vector operations easily. This results in having many arrays to carry the temporary results from one loop to another loop. Similarly, programs in array languages are translated by the early stages of the compiler to use temporary arrays to hold the partial terms in the array expressions. These programs may run slowly on non-vector machines because of the large intermediate arrays, which are either introduced by the programmer or by the compiler. Array contraction can reduce these array references to scalar references, which result in a smaller memory footprint and better cache utilization.

Figure 5-11 shows an excerpt from one of the frequently executed loops in the `flo88` application before and after the code transformation. We use the user-parallelized version in Chapter 4 as our baseline version. The transformed code is the result of two steps. First, we apply the affine partitioning algorithm[79][80] to the source and arrive at the code in Figure 5-11(b). Here, the structure of the code is changed such that all the operations involving `t(*,j)` and `d(*,j)` are executed together in the `j`-th iteration of an outer

(a)
```
      SUBROUTINE psmoo
      DO 50 k=2,kl
         ...
         DO 20 j=2,jl
            d(1,j)=0
20       CONTINUE
         DO 30 i=2,il
            DO 30 j=2,jl
               t(i,j)=d(i-1,j)* ...
               d(i,j)=t(i,j)* ...
30       CONTINUE
         DO 40 i=il-1,2,-1
            DO 40 j=2,jl
               =d(i,j)*...
40       CONTINUE
50    CONTINUE
```

(b)
```
      SUBROUTINE psmoo
      DO 50 k=2,kl
         ...
         DO 50 j=2,jl
            d(1,j)=0
            DO 30 i=2,il
               t(i,j)=d(i-1,j)* ...
               d(i,j)=t(i,j)* ...
30          CONTINUE
            DO 40 i=il-1,2,-1
               =d(i,j)*...
40          CONTINUE
50    CONTINUE
```

(c)
```
      SUBROUTINE psmoo
      DO 50 k=2,kl
         ...
         DO 50 j=2,jl
            d(1)=0
            DO 30 i=2,il
               t=d(i-1)*...
               d(i)=t*...
30          CONTINUE
            DO 40 i=il-1,2,-1
               =d(i)*...
40          CONTINUE
50    CONTINUE
```

**Figure 5-11. (a) Excerpt from `flo88`, (b) after affine partitioning, and (c) after array contraction**

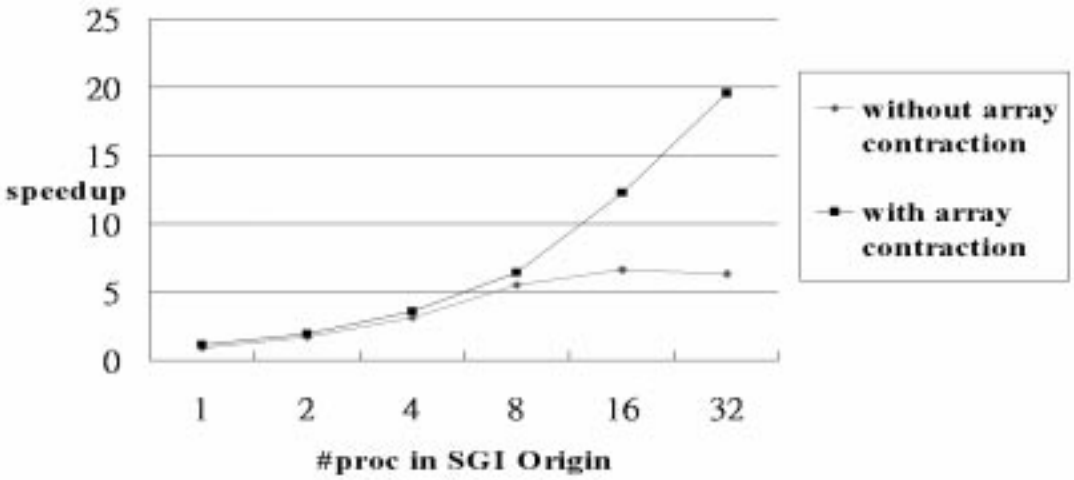


**Figure 5-12. Speedups for Flo88 without and with array contraction on a 32-processor SGI Origin**

loop. Interested readers are referred to [79][80] for details of the algorithm. The second step is to contract the two dimensional arrays down to a linear array in the case of `d` and a scalar in the case of `t`. An array can be contracted in a loop if the array has no upwards-exposed reads in the loop, if there are no loop-carried dependences, and if the array is not live at the exit of the loop. The size of the contracted array is the footprint of the data written in an iteration.

With its large number of intermediate arrays like those shown in Figure 5-11, the `flo88` program serves as an interesting case study. This code shows little speedup when parallelized by a commercial compiler. In Figure 5-12, the data are collected on a 32-processor SGI Origin machine. The characteristics of the Origin system are presented in Section 6.5.1.1. The code delivers a speedup of only 6.3 on 32 processors. In the next step, we apply loop

fusion and array contraction to the program. Contraction is applied to 20 arrays, 9 out of which are 3-dimensional arrays and the rest are 2-dimensional ones. Because array contraction reduces the memory footprint, the entire working set of the program may now fit into the cache. As a result, array contraction helps reduce both memory latency on a uniprocessor and memory contention between processors. The uniprocessor code speeds up by 10%, and because fewer memory references are needed, the contracted version is more scalable. Array contraction with the liveness information improves the performance from 6.3 to 19.6 on a 32-processor SGI Origin.

## 5.7. Related Work

Data-flow analysis for computing live variables[3][68] is well understood and has been applied to scalar variables for optimizations such as scalar privatization and dead code elimination. However, how to design an efficient and precise liveness analysis on array elements was still an open question. The previous SUIF compiler only performs liveness analysis on scalars[51]. It uses an interprocedural bottom-up pass to find the upwards-exposed read array sections in each region in the program, but it does not have a top-down pass to compute array liveness. The Polaris compiler[18] has an intraprocedural array liveness analysis for detecting privatizable variables that need not be finalized[104]. Their algorithm summarizes the effect of each loop with array sections representing the exposed uses of the loop. The liveness problem is formulated as an iterative data flow problem with the following distinctive features. First, the loops are the nodes in the flow graph and liveness is calculated at loop boundaries. Second, the data flow values are sections of array accesses. Third, unlike the conventional data flow equations, their implemented algorithm does not use any of the write sections to “kill” or reduce the exposed sections. The result is therefore less precise and degenerates to a largely flow-insensitive analysis. The reason is that the authors thought that generating more precise information is too expensive and unnecessary. Our proposed array liveness algorithm is more precise, while incurring only a small fraction of the cost of the parallelization analysis. Our experimental results show that the more precise algorithm can find many more private variables than the simpler versions. This is significant as liveness analysis is useful for many different program optimizations.

Finally, there has been a lot of work on program transformations such as loop fusion and array contraction. Bacon, Graham, and Sharp[12] provide a comprehensive survey on these optimizations.

## 5.8. Chapter Summary

We show that analyzing the liveness of the elements of an array can be precise and efficient in our two-phase region-based analysis framework. Our proposed interprocedural algorithm is both context-sensitive and flow-sensitive. The algorithm can find many more dead variables than simpler schemes that do not differentiate between array elements or that ignore the control flow within regions. The precision in the algorithm is important because we show that simpler versions yield inferior performance data. The algorithm was implemented in the SUIF parallelizing compiler.

We show that array liveness has many uses. They include eliminating the need of finalizing a privatizable array[51], separating live ranges of array variables so that their layouts can be optimized independently[8], and enabling contraction of arrays that are not live at loop exits[46].

We demonstrated that our liveness algorithm can effectively find the liveness information to support the above three uses. As a result, we improved the performance of four out of five Fortran programs. The speedup of `hydro` increases from 2.4 to 3.3, the speedup of `arc3d` increases from 2.0 to 2.4, and the speedup of `hydro2d` increases from 2.6 to 2.8 on a 4-processor Digital AlphaServer. The speedup of `flo88` improves from 6.3 to 19.6 on a 32-processor SGI Origin. The extent to which array liveness can speed up a program is highly dependent on the program itself. With so many uses, array liveness should be included as one of the standard analyses in parallelizing compilers.

Array liveness analysis can be integrated into programming tools as well. For example, our slicing tool[78] will benefit from the accurate array liveness information. The separate live ranges result in separate smaller slices and simplify the program structure, making the code easier for programmers to understand and optimize a program.

# 6 Interprocedural Reduction Analysis

Reduction operations occur often in scientific programs. A reduction is the application of an associative operation (for instance, addition, multiplication, and finding minimums and maximums) to combine a data set. Because of the associativity of a reduction operation, the compiler may reorder the computation, and in particular, may execute portions of the computation in parallel. However, due to the finite-precision arithmetic in a computer, parallelizing the reduction operations can potentially yield incorrect results. But, in practice most associative computations can be parallelized without losing the precision. We find that all of our benchmark programs in this chapter validate even when the reduction operations are executed in parallel. Reduction recognition is an important component of automatically parallelizing scientific programs.

Our goal is to exploit the full potential of reductions in improving the performance of automatically parallelized scientific codes. In this chapter, we make the following contributions:

- **A powerful reduction recognition algorithm.** We present a general and powerful algorithm for finding reductions on both scalar and array variables. The algorithm has been implemented in our fully functional parallelizer. The algorithm extends beyond previous approaches in its ability to locate reductions to array regions, even in the presence of arbitrarily complex data dependences. As an important example, the algorithm can locate reductions on indirect array references through index arrays.

- **Interprocedural reductions.** The algorithm is the first to locate *interprocedural reductions*, reduction operations that span multiple procedures. We show that interprocedural reductions occur in some computationally-intensive loops in scientific programs.

- **Extensive evaluation of importance in standard benchmarks.** We measure the impact of reduction recognition on parallelization of a collection of programs from the Perfect Club benchmark[91], NAS parallel benchmark[13], and the SPEC floating point benchmark[105]. These results demonstrate that parallelizing reductions makes a tremendous difference in the amount of the computation that can be parallelized.

- **Implementation evaluation.** To get a performance advantage from parallelizing a reduction operation, the implementation of reductions must avoid adding significant overhead. We present analysis of several approaches to implementing reductions, describe the trade-offs for these implementations, and present performance measurements. We also present a novel implementation technique to make reductions to array variables efficient.

This chapter is organized into the following sections. Section 6.1 discusses the scope of our reduction analysis. Section 6.2 describes the interprocedural reduction recognition algorithm. Section 6.3 discusses various issues about how the implementation manages the overhead of reduction. Section 6.4 provides a collection of examples from the benchmark suites that illustrate the power of the reduction recognition algorithm. Section 6.5 provides results indicating the frequency with which reductions occur in the benchmark suites and quantifying their impact on the parallelization of these programs. The related work is presented in Section 6.6. Section 6.7 summarizes the chapter.

## 6.1. Scope of Our Reduction Analysis

A reduction is the application of an associative operation to combine a data set. Our algorithm can parallelize the associative operations such as addition, multiplication, and finding minimums and maximums. The algorithm can analyze both scalar reductions and array

reductions, as presented below. In addition, we want to recognize reductions which consist of multiple updates to the same variable.

### 6.1.1. Scalar Reductions

A summation of an array `A[1:N]` is typically coded as:

```
DO I = 1, N
     SUM = SUM + A(I)
ENDDO
```

The values of the elements of the array `A` is reduced to the scalar `SUM`. As shown in this example, reductions, when coded in sequential programming languages, are not readily recognizable as commutative operations. However, most parallelizing compilers will recognize scalar reductions such as this accumulation into the variable `SUM`. Such reductions can be transformed to a parallel form by creating a private copy of `SUM` for each processor, initialized to 0. Each processor updates its private copy with the computation for the iterations of the `I` loop assigned to it, and following execution of the parallel loop, *atomically* adds the value of its private copy to the global `SUM`.

### 6.1.2. Regular Array Reductions

To discover coarse granularity of parallelism, it is important to recognize reductions that write to not just simple scalar variables but also to array variables. Reduction on array variables are also common in scientific codes and are a potential source of significant improvements in parallelization results.

There are several variations on how array variables can be used in reductions. For example, we can simply replace the `sum` variable by an array element:

```
DO I = 1, N
     B(J) = B(J) + A(I)
ENDDO
```

Or, the reduction may write to the entire or a section of an array:

```
      DO I = 1, N
C           ... a lot of computation to calculate A(I,1:3)
            DO J = 1, 3
                 B(J) = B(J) + A(I,J)
            ENDDO
      ENDDO
```

Suppose, in this example, the calculations of `A(I,1:3)` for different values of `I` are independent. Standard data dependence analysis would find the `I` loop (the loop with index `I`) not parallelizable because all the iterations are reading and writing the same locations `B(1:3)`. It is possible to parallelize the outer loop by having each processor accumulate to its local copy of the array `B` and then sum all the local arrays together.

### 6.1.3. Sparse Array Reductions

Sparse computations pose what is usually considered a difficult construct for parallelizing compilers. When arrays are part of subscript expressions, a compiler cannot determine the location of the array being read or written. In some cases, loops containing sparse computations can still be parallelized if the computation is recognized as a reduction. In the example below, we observe that the only accesses to the sparse vector `HISTOGRAM` are commutative and associative updates to the same location, so it is safe to transform this reduction to a parallelizable form.

```
      DO I = 1, N
            HISTOGRAM(A(I)) = HISTOGRAM(A(I)) + 1
      ENDDO
```

It is possible to parallelize the code by having each processor compute a part of the array `A` and collect the information in a local histogram, and sum the histograms together at the end. Our reduction analysis can parallelize this reduction even when the compiler cannot predict the locations that are written.

## 6.2. Reduction Recognition

As defined previously, a reduction occurs when a location is updated on each loop iteration, where a commutative and associative operation is applied to that location's previous contents and some data value. We have implemented a simple, yet powerful approach to recognizing reductions, in response to the common cases we have encountered in experimenting with the compiler. The reduction recognition algorithm for both scalar and array variables is similar, as scalar reductions are just a degenerate version of array reductions. This section focuses on array reduction recognition, which is integrated with the array data-flow analysis in the SUIF Compiler.

### 6.2.1. Problem Formulation for Reduction Analysis

The formulation of our reduction recognition algorithm is different from that used in previous compilers, and is powerful enough to allow our compiler to parallelize all the examples in Section 6.1. We model a reduction operation as consisting of a series of *commutative updates*. An update operation consists of reading from a location, performing some operation with it, and writing the result back to the same location. We say that a (dynamic) series of instructions contains a reduction operation to a data section *r* if all the accesses to locations in *r* are updates that can commute with each other without changing the program's semantics. Under this definition, it is easy to see that the examples above contain a reduction to, respectively, the regions `SUM`, `B(J)`, `B(1:3)` and `HISTOGRAM(1:M)` where `M` is the size of the array `HISTOGRAM`.

Not only is this model powerful, the analysis technique can be easily integrated with interprocedural array data-flow analysis. We will show how the reduction analysis is a *simple extension* of array data-flow analysis. The representation of array sections is common to both array data-flow analysis and array reduction analysis. The basic unit of data representation is a system of integer linear inequalities, whose integer solutions determine array indices of accessed elements. As described in Chapter 5, the denoted index tuples can also be viewed as a set of integral points within a polyhedron. The accessed region of an array is represented as a set of such polyhedra.

### 6.2.2. Interprocedural Reduction Recognition

The reduction recognizer is integrated with our array data-flow analysis. We will first describe the criteria for reductions and then the integration with the interprocedural data-flow analysis framework. The basic unit of data representation is a system of linear inequalities, whose integer solutions determine array indices of accessed elements. In addition, we add to the array section descriptor all the relationships among scalar variables that involve any of the variables used in the array index calculation.

#### 6.2.2.1. Locating Reductions

The reduction recognition algorithm searches for computation that meet the following criteria.

1. The computation is a commutative update to a single memory location *A* of the form, *A* = *A* `op` ..., where `op` is one of the commutative operations recognized by the compiler. Currently, the set of such operations includes +, *, `MIN`, and `MAX`. The `MIN` (and, similarly, `MAX`) reductions of the form "`if (a(i) < tmin) tmin = a(i)`" are also supported.
2. In the loop, the only other reads and writes to the location referenced by *A* are also commutative updates of the same type described by `op`.
3. There are no dependences on any operands of the computation that cannot be eliminated either by a privatization or reduction transformation.

This approach allows any commutative update to an array location to be recognized as a reduction, even without precise information about the values of the array indices. This point is illustrated by the sparse reductions in Section 6.1.3. The reduction recognition correctly determines that updates to `HISTOGRAM` are reductions, even though `HISTOGRAM` is indexed by another array A and so the array access functions for `HISTOGRAM` are not affine expressions.

In the following, we will first summarize our array data-flow analysis and then present our interprocedural reduction analysis in the data-flow analysis framework.

#### 6.2.2.2. Array Data-Flow Analysis

As described in Section 5.2, the bottom-up phase of our array data-flow analysis summarizes the data that has been read and data that has been written within each loop and procedure. The bottom-up algorithm analyzes the program starting from the leaf procedures in the call graph and analyzes a region only after analyzing all its subregions.

We compute the union of the array sections to represent the data accessed in a sequence of statements, with or without conditional flow. At loop boundaries, we derive a loop summary by performing the *closure* operation, which projects away the loop index variables in the array regions. We summarize the sections of data accessed in a loop to eliminate the need to perform $n^2$ dependence tests for a loop containing $n$ array accesses. At procedure boundaries, we perform parameter mapping, reshaping the array from formal to actual parameter if necessary. At each loop level, we apply a data dependence test and privatization test to the read and written data summaries[4][51].

#### 6.2.2.3. Integration into the Data-Flow Analysis Framework

In terms of the data-flow analysis framework, reduction recognition requires only a flow-insensitive examination of each loop and each procedure body. Array reduction recognition is integrated into the array data-flow analysis from the previous section. Whenever an array element is involved in a commutative update, the array analysis derives the union of the summaries for the read and written sub-arrays and marks the system of inequalities as a reduction of the type described by `op`, where `op` is either +, *, `MIN`, or `MAX`. When meeting two systems of inequalities during the interval analysis, the resulting system of inequalities will only be marked as a reduction if both reduction types are identical.

#### 6.2.2.4. Interprocedural Algorithm

Working in a bottom-up manner, the interprocedural algorithm starts by detecting statements that update a location via an addition, multiplication, minimum or maximum operator. The algorithm keeps track of the operator and the reduction region, which is calculated in the same manner as above if an array element has been updated. To calculate the reductions carried by a sequence of statements, we find the union of the reduction regions for

each array and each reduction operation type. The result of the union represents the reduction region for the sequence of statements if it does *not* overlap with other data regions accessed via non-commutative operations or other commutative operations. At loop boundaries, we derive a summary of the reduction region by projecting away the loop index variables in the array region. Again, the summary represents the reduction region for the entire loop if it does not overlap with other data regions accessed.

The way we determine if a loop is parallelizable is as follows. We first apply the data dependence test and the privatization test on the read and write summaries and determine whether there is any dependence. If not, the loop is parallelizable and reductions are not necessary. Otherwise, we check if all data dependences on an array result from its reduction regions. If so, we parallelize the loop by generating parallel reduction code for each such array.

## 6.3. Implementation of Parallel Reductions

Even though there are many reductions, they do not necessarily translate to high parallel efficiency. This section discusses various issues in how the implementation manages the overhead of reduction.

### 6.3.1. Implementing Scalar Reductions

Replacing reductions to a scalar variable with a parallel implementation is rather straightforward. We first allocate a private copy of the variable on each processor. The private copies are initialized to the identity under the specific reduction operator; for example, the identities for the summation, product, minimum, and maximum operations on integers are zero, one, the maximum integer, and the minimum integer (representable on the machine), respectively. Each processor simply updates its private copy in the parallel loops. At the end of the parallelized loop computation, each processor performs a global accumulation whereby all non-identity elements of the local copies of the variable are accumulated into the original variable. Synchronization locks are used to guard accesses to the original variable to guarantee that the updates are atomic. It is important to note that there is no conten-

tion during the execution of the parallelized loop because each processor has a local copy. Critical sections only occur after the loop computation.

For example, our parallelizer transforms the following loop:

```
DO I = 1, N
     S = S + A[I];
ENDDO
```

to the following corresponding SPMD (Single Program Multiple Data) C version, where `pid` is the processor id and `P` is the number of processors.

```
/* Initialization of the private variable “priv_s” */
priv_s = 0;

for (i =  max(n*pid/P, 0); i < min(n*(pid+1)/P, n); i++)
     priv_s = priv_s + a[i];

/* Finalization */
lock();
s = s + priv_s;
unlock();
```

The contention at the critical section is negligible for most shared-memory multiprocessors. Tree combinations can be used to reduce the serialization if the number of processors is large.

### 6.3.2. Overheads of Array Reductions

A simple way to implement parallel reductions on arrays is to use a similar approach as above. Take the loop below as an example.

```
DO I = 1, N
     DO J = 1, M
          B(J) = B(J) + A(I,J)
     ENDDO
```

```
ENDDO
```

The loop `I` can be parallelized in the following way.

```
/* Initialization of the private variable “priv_b” */
for (j = 0; j < m; j++)
     priv_b[j] = 0;

for (i =  max(n*pid/P, 0); i < min(n*(pid+1)/P, n); i++)
     for (j = 0; j < m; j++)
          priv_b[j] = priv_b[j] + a[j][i];

/* Finalization */
lock();
for (j = 0; j < m; j++)
     b[j] = b[j] + priv_b[j];
unlock();
```

Each processor has its own private copy of array `B`, `priv_b`, whose elements are initialized to 0’s. After accumulating a share of the data to its own local copy, the processor adds the local array to the original array `B` in a critical section.

Examining the code above, we observe that there are two major sources of overhead: initialization and finalization. The initialization and finalization costs are proportional to the size of the reduction region. The number of iterations in the parallelized loop is unrelated to the size of the reduction region, as illustrated by the example in Section 6.4.1. If the array index is data dependent, we must assume that the reduction region is the entire region. It is important to find the smallest possible region. Furthermore, the amount of total time spent on the initialization and finalization overhead increases linearly with the number of processors, since each processor must have its own copy. More importantly, the finalization done in the manner above is serialized since only one processor can update the final copy at a time. There are many facets to solving this problem. As discussed later, they include

reducing the initialization and parallelization overheads and reducing the serialization of finalization.

### 6.3.3. Minimizing the Reduction Region

Keeping track of array regions is important for minimizing the overheads in parallelizing reductions. As an example, consider the following code in `bdna` from the `Perfect` benchmarks.

```
      DO I=1,NSP
           DO IA=1,NATOMS
                FAX(IA) = FAX(IA) + ...
C               ... a lot of computation in this loop body ...
           ENDDO
      ENDDO
```

The loop `I` can be parallelized by performing parallel reductions on `FAX`, a 2000-element array. However, the reduction region on `FA` is only for the region between 1 and `NATOMS`, a small integer. The reduction overhead is minimized by initializing and finalizing only the region that is used.

### 6.3.4. Minimizing Serialization of Finalization

While the total amount of time spent on initialization increases with the number of processors, at least the overhead is parallelizable. In other words, the elapsed time spent on initialization does not increase with the number of processors. If finalization is serialized, as in our naive implementation, the elapsed time actually increases linearly with the number of processors involved in the parallel computation.

We reduce the serialization by two methods. First, each reduction region has its own locks, so that different processors can perform finalizations to different regions simultaneously. Second, for arrays that are determined to be large, we partition the arrays logically into several sections and assign different locks to them. Furthermore, instead of each processor finalizing in the same order, we stagger the starting points. The *i*-th processor finalizes the

sections in the following order: *i, i+1, ..., n, 1, ..., i-1*. This order minimizes contention in the case where processors reach the finalization stage at roughly the same time.

### 6.3.5. Critical Sections for Individual Updates

It is possible to eliminate all initialization and finalization overheads, at the risk of contention while the processors are executing the parallel loop itself. Instead of having a private copy of the array, each processor enters a critical section for each individual update to the reduction array. To minimize contention, we can assign a lock to a single array element or to a group of array elements. For example, the loop below is an excerpt from the `bdna` program in the Perfect Club benchmark.

```
DO J=1,L
     FOX(IND(J)) = FOX(IND(J)) + FOXP(J)
ENDDO
```

The parallelized code for the above loop is the following.

```
DO J=...
     LOCK(IND(J))
     FOX(IND(J)) = FOX(IND(J)) + FOXP(J)
     UNLOCK(IND(J))
ENDDO
```

No initialization and finalization code is needed. The array `FOX` is accessed directly by every processor, but the access to the same element is serialized.

Using individual locks is useful for sparse computation whose region information is difficult to summarize. But the locking overhead in the loop body must be amortized by the computation outside the reduction code to make the code run efficiently. The performance is also limited by the number of locks that the run-time system provides. This scheme of using individual locks is not implemented in our compiler. Among the benchmarks that we

studied, we have not found a sparse computation with reductions that have a lot of code outside the reduction code.

## 6.4. Reduction Examples

We now illustrate the strength of our algorithm by showing some of the representative cases encountered in our experiments. These four case studies shows that the formulation of a reduction analysis should enable the parallelization of interprocedural and sparse reductions.

### 6.4.1. Array Sections

The following is a simple example illustrating how our formulation of the reduction can easily handle non-trivial array access patterns. Consider the following loop in the Perfect benchmark `ocean`.

```
DO I=2,N2P
     WORK(1) = MAX(WORK(1), WORK(I))
     ...
ENDDO
```

The algorithm correctly determines that the reduction region is simply `WORK(1)` whereas the data region read within the loop is `WORK(2:N2P)`. Since the reduction region does not overlap with the read region, the algorithm correctly deduces that the loop constitutes a reduction even though the loop uses the array *both* as the source data set and as the target of a reduction.

### 6.4.2. Indirect References

Our ability to detect reductions to write to data-dependent locations in a program has enabled our compiler to parallelize some key computations in the benchmark suite that would otherwise be serialized. The following code excerpt constitutes the main computation in the `cgm` program of the NAS parallel benchmark that our compiler is able to parallelize:

```
      DO J = 1, N
           XJ = X(J)
           DO K = COLSTR(J), COLSTR(J+1)-1
                Y(ROWIDX(K)) = Y(ROWIDX(K)) + A(K) * XJ
                ...
           ENDDO
      ENDDO
```

Starting with the statement in the innermost loop, our analysis recognizes that the add operation is an accumulation to the data element `Y(ROWIDX(K))`. Since the compiler cannot tell the value of the index, it has to conservatively assume that any location in the array could be updated. When considering each loop, it finds that the array `Y` is written only by accumulation operations within each loop, and is thus a potential reduction target; for each loop the potentially updated region is the entire array. Since the entire array is a superset of data touched, our compiler can parallelize the outermost loop by having each processor accumulate to its private copy of the whole `Y` vector and summing up all the vectors at the end of the computation. This simple technique allows the compiler to find parallelism even among sparse computations, which is very hard to parallelize with standard parallelization techniques.

### 6.4.3. Coarse-Grain Parallelism and Multiple Reduction Statements

Even though our reduction recognition algorithm is not complicated, it is rather robust and capable of finding reductions that span many lines of code. The following is an excerpt of a 148-line loop which accounts for 68% of the execution time of the `mdljsp2` and `mdljdp2` programs in the SPEC92 floating point benchmark.

```
      SUBROUTINE FORCE
      DO 10 I = 1, 500 - 1
           JBEG = NBINDX(I)
           JEND = NBINDX(I+1) - 1
           IF (JBEG.GT.JEND) GO TO 20
           CALL JLOOPU (I,JBEG,JEND)
10    CONTINUE
```

```
      RETURN
      END

      SUBROUTINE JLOOPU(I,JBEG,JEND)
      DO 20 JX = JBEG,JEND
           J = NBTABL(JX)
           ...
           IF ... GOTO 20
           XFORCE(I) = XFORCE(I) + FIJX
           YFORCE(I) = YFORCE(I) + FIJY
           ZFORCE(I) = ZFORCE(I) + FIJZ
           XFORCE(J) = XFORCE(J) - FIJX
           YFORCE(J) = YFORCE(J) - FIJY
           ZFORCE(J) = ZFORCE(J) - FIJZ
20    CONTINUE
      RETURN
      END
```

First, the compiler invokes the symbolic analysis to analyze the scalar variables in the program, and determines that the value of `I` in the `JLOOPU` subroutine is between 1 and 499. Next, in the array analysis phase the compiler starts with analyzing the statements in the innermost loop `JX`. Our compiler easily determines that assignments to the `XFORCE`, `YFORCE`, and `ZFORCE` arrays are all commutative update operations. Since the value of `J` is unknown, the compiler calculates the reduction regions to be the entire "force" arrays. To summarize the statements within an iteration, the compiler finds the unions of the data regions for each array, and simply determines that there are summation reductions to all the "force" arrays. Note that neither the presence of multiple assignments to the same reduction region in a loop, nor the conditional flow present in the loop detracts from the analysis. Next, the compiler represents the entire loop `JX` using the closure of the summary of each iteration.

The compiler summarizes the entire subroutine by (1) performing parameter mapping and (2) reshaping the array from formal to actual parameter if necessary. No reshaping is nec-

essary in the case of the JLOOPU subroutine. The loop I in the FORCE subroutine is then summarized in a similar manner. This technique allows our compiler to parallelize the most important loop in the entire mdljsp2 and mdljdp2 programs at the outermost level possible. As we will show in Section 6.5, the parallelization of the loop I in the FORCE subroutine results in significant speedup.

It is interesting to note how our reduction recognizer allows outermost loops be parallelized, and how parallelizing the outermost loop also helps in amortizing the reduction overhead. If we were to parallelize the innermost loop, with a small iteration count, the overhead (discussed in detail in Section 6.3) associated with reductions per loop could easily overwhelm the advantage of parallelizing the inner loop.

The mdljsp2 example shows that the ability to recognize reductions to array variables is particularly instrumental to parallelizing large regions of code. This coarse-grain parallelism is important for multiprocessors because a significant amount of independent computations can be performed without any synchronization. Consider a larger application spec77, a program from the Perfect Club benchmark:

```
DO LAT = 1, 38
      ...
      DO K = 1, 12
          CALL FL22(...,Y(1,K),...)
      ENDDO
ENDDO

SUBROUTINE FL22(...,FLN,...)
DO LL = 1, 31
      DO I = 1, 31, 2
          FLN(I,LL) = FLN(I,LL) + ...
      DO I = 2, 30, 2
          FLN(I,LL) = FLN(I,LL) + ...
```

Even though the compiler can parallelize both loops in the subroutine FL22 via standard data dependence analysis, our analysis also notes that all the accesses to the array FLN

within FL22 are accumulation operations. Mapping FLN to the array Y at the procedure boundary, the compiler retains the reduction information across the procedure call. When the analysis considers the LAT loop, it determines that the loop carries a dependence on Y. The analysis examines whether all accesses corresponding to dependences on the loop are marked as potential reductions. They are, so the loop can be parallelized, by generating specialized reduction code. (Note that there are multiple such reductions in this loop.) By propagating reduction summaries across procedure boundaries, we are able to parallelize larger amounts of codes, with much lower parallelism overhead. This particular loop is composed of more than 1,000 lines of code from the original loop and its invoked procedures. The outer parallel loop contains 60 subroutine calls to 13 different procedures. Within this loop, the compiler found 5 interprocedural reduction arrays, 48 interprocedural privatizable arrays, and 27 other arrays which are accessed independently. If this loop is fully inlined, it would contain over 10,000 lines of code. Inlining is not an alternative for this program. Thus, having an interprocedural algorithm is important.

## 6.5. Experimental Results

Our reduction algorithm automatically parallelizes the reduction operations in sequential applications without relying on user directives. Parallel programs generated by our compiler are executed on cache-coherent shared address-space multiprocessors. We will first describe our experimental setup in Section 6.5.1, evaluate the frequency of commutative updates and reductions in Section 6.5.2 and 6.5.3, and present the performance results in Section 6.5.4.

### 6.5.1. Experimental Setup

The reduction recognition algorithm described above is implemented in the Stanford SUIF compiler. The following collection of results were obtained with the SUIF compiler, which takes as input sequential Fortran 77, and generates parallel SPMD (Single Program Multiple Data) code with calls to our own runtime thread package. The resulting C code is compiled using native C compilers. The parallel versions were generated automatically with no source code changes to the standard versions of these programs, with one notable exception: we have modified some type declarations in `spec77` that were invalid in Fortran 77.

#### 6.5.1.1. Target Architectures

Our runtime thread package supports parallel execution on a variety of machines, including the bus-based SMP (Symmetric Multi-Processors) such as the Silicon Graphics Challenge series[59], and the CC-NUMA (Cache-Coherent Non-Uniform Memory Access) architectures such as the Stanford DASH[75], the Stanford FLASH[72], and the Silicon Graphics Origin series[29]. While parallel speedups measure the overall effectiveness of a parallel system, they are also highly machine dependent. Since parallel reductions incur more overhead than simple parallelization, not only do speedups depend on the number of processors, they are sensitive to many aspects of the architecture, such as the cost of synchronization, the interconnect bandwidth, and the memory subsystem. Thus, we evaluate the effectiveness of our reduction algorithm on both an SMP and a CC-NUMA machine. A summary of the Silicon Graphics Challenge (an SMP) and the Silicon Graphics Origin (a CC-NUMA machine) multiprocessors is given in Figure 6-1.

The Challenge is a snoop-based shared-memory architecture. Both the first- and second-level caches are direct-mapped. The data interface to the off-chip cache in a Challenge is 128 bits wide and runs at a half or a third of the on-chip clock rate. Parallel reductions incur communication overhead between the processors. The multiprocessor interconnect used in the Challenge is called PowerPath-2. PowerPath-2 is a wide, split transaction bus that can sustain a transfer rate of 1.2 Gigabytes per second. The bus implements a write invalidate cache coherency protocol and has a 256-bit data bus and a 40-bit address bus. The independent data and address buses provide support for split transactions, and the PowerPath-2 can have up to eight outstanding read transactions[59].

The Origin system is a directory-based CC-NUMA. The Origin is composed of a number of processing nodes connected by a switch-based interconnection network. Each node contains two MIPS R10000 processors, each with first- and second-level caches. Unlike the MIPS R4400 processor, the Origin's MIPS R10000 processor is dynamically scheduled and does not stall on a read miss. It has two integer arithmetic-logic units, one load/store unit, and two floating-point units. Both the first- and second-level caches are two-way set associative. The cache line size for the second-level cache is 128 bytes. The Origin system consists of 780-MB/s SysAD bus (which is the memory bus of the two R10000 proces-

| Machine | Silicon Graphics Challenge | Silicon Graphics Origin |
|---|---|---|
| Architecture | bus-based SMP | CC-NUMA<br>(2 processors per node) |
| Processors | 8 superpipelined MIPS R4400 | 4 superscalar MIPS R10000 |
| Clock speed | 200 MHz | 195 MHz |
| On-chip cache | 16 KB Instruction (1-way)<br>+ 16 KB Data (1-way) | 32 KB Instruction (2-way)<br>+ 32 KB Data (2-way) |
| External cache | 4 MB (128 B/line, 1-way) | 4 MB (128 B/line, 2-way) |
| System bus bandwidth | 1.2 GB/s | 780 MB/s SysAD bus<br>780 MB/s node-to-network |
| Main memory | 768 MB | 640 MB |
| Operating system | IRIX 5.3 | IRIX 6.4 |

**Figure 6-1. Characteristics of the two multiprocessor systems used for the experiments**

sors), 670-MB/s bandwidth for local memory access, and 780-MB/s bandwidth for node-to-network access each way[29].

### 6.5.2. Commutative Updates

To evaluate the applicability of our reduction recognition algorithm, we apply our algorithm on all the programs in the SPEC92 floating point benchmark suite[105]. The suite is a set of 14 floating-point programs. Figure 6-2 provides the program description and the number of reduction operations for each SPEC92 program. Because our interprocedural analysis is available only for Fortran, we omit `alvinn` and `ear`, the two C programs, and `spice`, a program of mixed Fortran and C code. We also omit `fpppp` because it contains very little loop-level parallelism and has many type errors in the original Fortran source.

| Program | #lines | Description | Sum | Prod | Min | Max | Total |
|---|---|---|---|---|---|---|---|
| **SPEC92** | | | | | | | |
| `hydro2d` | 4461 | Navier-Stokes equation | 4 | 0 | 1 | 1 | 6 |
| `nasa7` | 1105 | NASA Ames kernels | 6 | 1 | 1 | 3 | 11 |
| `su2cor` | 2514 | quantum physics module | 51 | 1 | 0 | 0 | 52 |
| `tomcatv` | 195 | mesh generation kernel | 2 | 0 | 2 | 0 | 4 |
| `wave5` | 7628 | 2-D particle simulation | 83 | 15 | 0 | 0 | 98 |
| `swm256` | 487 | shallow water model | 3 | 0 | 0 | 0 | 3 |
| `doduc` | 5334 | Monte Carlo simulation | 35 | 1 | 10 | 0 | 46 |
| `mdljdp2` | 4316 | equations of motion | 50 | 3 | 0 | 0 | 53 |
| `mdljsp2` | 3885 | equations of motion (single precision) | 50 | 3 | 0 | 0 | 53 |
| `ora` | 373 | optical ray tracing | 1 | 0 | 0 | 0 | 1 |
| Total | 30298 | — | 285 | 24 | 14 | 4 | 327 |

**Figure 6-2. Numbers of reductions according to their operation types in SPEC92 benchmark**

These counts represent the static number of commutative update operations that appear in the programs; the operations may or may not occur within parallel loops. We see that commutative updates are very common and most are summations.

### 6.5.3. Role of Reduction in Parallelization

Now we present measurements on how often these commutative update operations must be converted to parallelized reductions in order to parallelize loops in the benchmark programs. To evaluate our reduction algorithm, we present two set of results on the SUIF compiler system, one without using reduction analysis and the other with reduction analysis.

| Program | No. of lines | Description |
|---|---|---|
| **NAS** | | |
| `appbt` | 4457 | block tridiagonal partial differential equation solver |
| `applu` | 3285 | parabolic/elliptic partial differential equation solver |
| `appsp` | 3516 | scalar pentadiagonal partial differential equation solver |
| `buk` | 305 | integer bucket sort of a random sequence |
| `cgm` | 855 | unstructured sparse solver using conjugate gradient |
| `embar` | 135 | parallel random number generator |
| `fftpde` | 773 | 3-D partial differential equation of fast fourier transform |
| `mgrid` | 676 | 3-D multigrid solver for computing potential field |
| **Perfect** | | |
| `adm` | 6105 | pseudospectral mesoscale air pollution hydrodynamics |
| `arc2d` | 3965 | 2-D fluid flow implicit finite difference algorithm |
| `bdna` | 3980 | molecular dynamics of DNA molecules |
| `dyfesm` | 7608 | 2-D dynamic finite element structural analysis |
| `flo52` | 1986 | transonic inviscid flow using unsteady Euler equations |
| `mdg` | 1238 | molecular dynamics simulation of water molecules |
| `mg3d` | 2812 | 3-D depth migration using fast fourier transforms |
| `ocean` | 4343 | 2-D grid simulation of fluid flow |
| `qcd` | 2327 | quantum chromodynamics using Monte Carlo method |
| `spec77` | 3889 | spectral analysis simulation of weather systems |
| `track` | 3735 | threat/sensor-based missile tracking system |
| `trfd` | 485 | two-electron integral transforms |

**Figure 6-3. Program information for the NAS Parallel benchmark and the Perfect Club benchmark**

The former is obtained by using the baseline system, which includes interprocedural data dependence analysis, interprocedural scalar analysis, and interprocedural array privatization analysis. The latter uses array reduction analysis, in addition to the analyses in the baseline system.

We experiment with the SPEC92 floating point benchmark, the NAS Parallel benchmark, and the Perfect Club benchmark. Figure 6-3 provides the program description and the number of lines of code for each NAS program and each Perfect Club program. The program information for the SPEC92 floating point benchmark is already provided in Figure 6-2.

The NAS parallel benchmark is a suite of eight programs used for benchmarking parallel computers[13]. NASA provides sample sequential programs plus application information, with the intention that they can be rewritten to suit different machines. We use all the NASA sample programs except for embar. We substitute for embar a version from Applied Parallel Research (APR) that separates the first call to a function, which initialize static data, from the other calls. The Perfect Club benchmark is a set of sequential code used to benchmark parallelizing compilers[91]. We present results on 12 of 13 programs here. `Spice` contains pervasive type conflicts and parameter mismatches in the original Fortran source that violate the Fortran 77 standard. This program is considered to have very little loop-level parallelism.

#### 6.5.3.1. Static Measurements

Figure 6-4 presents a count of the number of loops containing reductions that must be parallelized in order to parallelize the loop. The interprocedural and intraprocedural categories divide the reductions into those that span multiple procedures and those that do not. Note that some of the reductions classified as intraprocedural are in loops that contain procedure calls; a reduction is only classified as interprocedural if the commutative update operation and the loop in which it is a reduction are in different procedures. We also divide the loops into those containing only scalar reductions, only array reductions, or both types of reductions. The column labeled “number of parallel loops with reduction” gives the number of loops in all categories that require parallel reductions in order to be parallelized. Note that

| Program | No. of parallel loops with interprocedural reduction | | | No. of parallel loops with intraprocedural reduction | | | No. of parallel loops with reduction | No. of parallel loops w/o reduction | Total no. of parallel loops |
|---|---|---|---|---|---|---|---|---|---|
| | scalar | array | both | scalar | array | both | | | |
| **SPEC92** | | | | | | | | | |
| hydro2d | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 146 | 147 |
| nasa7 | 0 | 0 | 0 | 1 | 1 | 0 | 2 | 65 | 66 |
| su2cor | 0 | 0 | 0 | 7 | 3 | 0 | 10 | 60 | 69 |
| tomcatv | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 9 | 10 |
| wave5 | 0 | 0 | 0 | 6 | 0 | 0 | 6 | 193 | 198 |
| swm256 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 23 | 24 |
| doduc | 0 | 0 | 0 | 12 | 0 | 0 | 12 | 225 | 237 |
| mdljdp2 | 0 | 0 | 2 | 2 | 0 | 2 | 6 | 9 | 15 |
| mdljsp2 | 0 | 0 | 2 | 2 | 0 | 2 | 6 | 9 | 15 |
| ora | 1 | 0 | 0 | 0 | 0 | 0 | 1 | 7 | 8 |
| **NAS** | | | | | | | | | |
| appbt | 0 | 3 | 0 | 3 | 3 | 0 | 9 | 161 | 169 |
| applu | 0 | 3 | 0 | 3 | 4 | 0 | 10 | 126 | 136 |
| appsp | 0 | 3 | 0 | 3 | 3 | 0 | 9 | 157 | 166 |
| buk | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 3 | 4 |
| cgm | 0 | 0 | 0 | 4 | 2 | 0 | 6 | 13 | 19 |
| embar | 0 | 0 | 1 | 2 | 1 | 0 | 4 | 2 | 5 |
| fftpde | 0 | 0 | 0 | 4 | 0 | 0 | 4 | 21 | 25 |
| mgrid | 0 | 0 | 0 | 5 | 0 | 0 | 5 | 33 | 38 |
| **Perfect** | | | | | | | | | |
| adm | 0 | 0 | 0 | 6 | 0 | 0 | 6 | 170 | 176 |
| arc2d | 0 | 0 | 0 | 8 | 0 | 0 | 8 | 182 | 190 |
| bdna | 0 | 0 | 0 | 4 | 25 | 3 | 32 | 108 | 140 |
| dyfesm | 0 | 1 | 0 | 5 | 5 | 0 | 11 | 124 | 135 |
| flo52 | 0 | 0 | 0 | 6 | 0 | 0 | 6 | 150 | 156 |
| mdg | 0 | 0 | 0 | 3 | 0 | 0 | 3 | 35 | 38 |
| mg3d | 0 | 0 | 0 | 9 | 2 | 0 | 11 | 95 | 106 |
| ocean | 0 | 0 | 0 | 4 | 1 | 0 | 5 | 105 | 109 |
| qcd | 0 | 0 | 0 | 12 | 7 | 0 | 19 | 80 | 99 |
| spec77 | 0 | 1 | 0 | 10 | 13 | 0 | 24 | 294 | 314 |
| track | 0 | 0 | 0 | 1 | 3 | 0 | 4 | 52 | 55 |
| trfd | 0 | 0 | 0 | 0 | 5 | 0 | 5 | 16 | 21 |
| Total | 1 | 11 | 5 | 126 | 78 | 7 | 228 | 2673 | 2890 |

**Figure 6-4. Impact of reductions (static measurements)**

in Figure 6-4 we only count the outermost parallel loop in a loop nest, even if the inner ones may also be parallel.

The second column from the right end shows the number of loops that are parallelized without parallel reductions. The last column reports the total number of outermost parallel loops. Note that the number of parallel loops without reduction plus the number of loops requiring reduction does not necessary equal the number of parallel loops. This is because when an outer loop in a nest is parallelized, we only count the nest once, even if parallelizing its inner loops is also possible. Thus, sometimes parallelizing a reduction allows us to parallelize an outer loop in a nest; when the reduction is suppressed, parallelizing some inner loops may still be possible.

From this figure, we see that parallelizable reductions occur in almost all of the programs. Clearly, parallelized reductions are widely applicable. We note that most of the reductions are intraprocedural reductions on scalar variables. Array and interprocedural reductions occur less often. However, as we will see in subsequent results, the array and interprocedural reductions can have a tremendous impact on performance.

#### 6.5.3.2. Dynamic Measurements

The previous section presents static counts of the parallelizable loops found with and without reductions. Static loop counts, though, are not good indicators of whether parallelization is successful. Specifically, parallelizing just one outermost loop can have a profound impact on the performance of a program. Dynamic measurements provide much more insight into whether a program may benefit from parallel reductions. Thus, we present a series of results gathered from executing the program on two parallel machines. Figure 6-5 shows whether the reduction loops are ones in which the program spends its time. We use two dynamic measurements which we call *parallelism coverage* and *parallelism granularity*. Parallelism coverage, defined in Chapter 2, gives the percentage of the sequential execution time spent in parallelized regions of the code. Parallelism coverage gives us a first order approximation of how well the parallel program can be expected to perform; programs with low coverage do not perform well. By Amdahl's law, a program with a par-

| Program | Parallelism coverage | | | Parallelism granularity | | |
|---|---|---|---|---|---|---|
| | no reduction analysis (%) | use reduction analysis (%) | ratio of no reduc. vs. use reduc. (%) | no reduction analysis (msec.) | use reduction analysis (msec.) | ratio of no reduc. vs. use reduc. (%) |
| **SPEC92** | | | | | | |
| su2cor | 86.9 | 94.9 | 92 | 0.8 | 0.9 | 89 |
| tomcatv | 88.2 | 96.1 | 92 | 79.9 | 88.6 | 92 |
| mdljdp2 | 3.8 | 87.1 | 4 | 0.5 | 5.4 | 9 |
| mdljsp2 | 3.2 | 82.9 | 4 | 0.7 | 6.5 | 11 |
| ora | 0 | 100.0 | 0 | 0.02 | 64716.7 | 0 |
| **NAS** | | | | | | |
| appbt | 97.9 | 99.4 | 98 | 12.8 | 13.1 | 98 |
| cgm | 4.2 | 96.4 | 4 | 0.86 | 18.4 | 5 |
| embar | 0 | 100.0 | 0 | 0.009 | 8133.6 | 0 |
| **Perfect** | | | | | | |
| bdna | 30.1 | 87.4 | 34 | 4.1 | 7.3 | 56 |
| qcd | 18.1 | 37.5 | 48 | 0.005 | 0.008 | 63 |
| spec77 | 80.0 | 86.0 | 93 | 0.07 | 0.3 | 23 |
| trfd | 79.3 | 96.6 | 82 | 0.01 | 2.0 | 1 |

**Figure 6-5. Coverage and granularity information on the 12 SPEC92, NAS, and Perfect Club programs on which parallel reductions have an impact**

allel coverage of 80% can at most speedup by 2.5 on 4 processors. High coverage is indicative that the parallelizer is locating significant amounts of parallelism in the computation.

Parallelism granularity is the average length of computation between synchronizations in the parallel regions. Due to overheads of synchronization and data communication, programs with low granularity do not perform well. Figure 6-5 lists the programs for which reduction recognition is important to discover coarser granularity of parallelism.

We obtain our coverage and granularity data on a uniprocessor Silicon Graphics Challenge. We experiment with the SPEC92 floating point benchmark, the NAS Parallel benchmark, and the Perfect Club benchmark. Figure 6-5 reports *only* those programs for which parallel reductions increase more than 2% of the coverage or more than 2% of the granularity. Interested readers are referred to [51] for the detailed performance data for the other programs whose performance is not affected by parallel reductions.

We observe from these results that reductions are critical in extracting parallelism from 12 out of the 30 programs in these benchmark suites. The 12 programs are `su2cor`, `tomcatv`, `mdljdp2`, `mdljsp2`, `ora`, `appbt`, `cgm`, `embar`, `bdna`, `qcd`, `trfd`, and `spec77`. Coverage is above 80% for 11 of the 12 programs. Granularity is above 1 millisecond for 9 of the 12 programs. In our experience, granularities on the order of 1 millisecond are high enough to yield speedup.

Recall from Figure 6-4 that `mdljdp2`, `mdljsp2`, `ora`, `appbt`, `embar`, and `spec77` all contain interprocedural reductions. These interprocedural loops are the main reasons for the increased coverage and granularity. Despite the fact that interprocedural reductions are not all that common, when they do occur, because interprocedural loops often contain a significant amount of work, they can greatly impact performance. All of the 12 programs in Figure 6-5, except for `tomcatv` and `ora`, contain array reductions.

### 6.5.4. Performance Improvement

This section presents the performance results on both the Silicon Graphics Challenge[59] and the Silicon Graphics Origin[29]. Our reduction analysis substantially increases the coverage on 12 out of 30 programs in the three benchmark suites: the SPEC92 floating

point benchmark, the NAS Parallel benchmark, and the Perfect Club benchmark. Figure 6-6 compares the speedups of these 12 programs on a 4-processor SGI Challenge with and without parallelized reductions. The speedup data for SGI Origin will be presented later in Figure 6-7.

We observe from Figure 6-6 that *nine* programs, `mdljdp2`, `mdljsp2`, `ora`, `tomcatv`, `cgm`, `embar`, `bdna`, `trfd`, `spec77`, benefit from parallelized reductions. The speedups for `mdljdp2`, `mdljsp2`, `ora`, `cgm`, `embar`, and `bdna` are quite significant, as compared with speedups of approximately 1 without reduction. Figure 6-6 also explains the reasons for the increased speedups. The sparse reductions and the interprocedural reductions are the key to improving the performance of these six programs. The speedups for the remaining three programs, `tomcatv`, `trfd`, and `spec77`, also improve. Although little parallel speedup is observed on `spec77`, the improvement over the baseline system confirms the validity of our preference for interprocedural reduction loops. The program `spec77` contains a lot of input and output; their speedup also depends on the success of parallelizing I/O. The speedup of `tomcatv` improves by 18%. This program has poor memory behavior and its performance can be improved significantly via data and loop transformation to improve cache locality[9] and by using techniques to minimize synchronization[102]. In the case of `trfd`, the speedup increases by 12% due to array reductions. The original speedup increases from 0.9 to 1.7 due to array privatizations.

The difference in parallel coverage observed earlier for these programs translates into positive effects on parallelization. Experimental results indicate that reductions are common and are critical to the success of finding coarse-grain parallelism. Of the twelve programs with observed differences in parallel coverage due to reduction, only `su2cor`, `appbt`, and `qcd` did not benefit from the increased parallelism. In the case of `su2cor`, the coverage increases by 9%, but the speedup is still 1.9 due to the overhead of reduction. The coverage of `appbt` also increases slightly, and hence the performance is the same. The parallelization of `appbt` relies on the array privatization technique, not on the array reduction technique.

| Program | Execution time of sequential version (seconds) | Speedups | | | Reasons for improvement | | |
|---|---|---|---|---|---|---|---|
| | | no reduct. analysis | use reduc. analysis | relative improve-ment | sparse reduc. | inter-proc. reduc. | intra-proc. reduc. |
| **SPEC92** | | | | | | | |
| `su2cor` | 156.2 | 1.9 | 1.9 | 0% | | | |
| `tomcatv` | 19.8 | 1.7 | 2.0 | 18% | | | ✔ |
| `mdljdp2` | 45.5 | 1.0 | 2.0 | 100% | ✔ | ✔ | |
| `mdljsp2` | 40.5 | 1.0 | 1.8 | 80% | ✔ | ✔ | |
| `ora` | 89.6 | 1.0 | 4.0 | 300% | | ✔ | |
| **NAS** | | | | | | | |
| `appbt` ($12^3$x$5^2$ grid) | 10.1 | 2.9 | 2.9 | 0% | | | |
| `cgm` (1400 elements) | 5.4 | 1.0 | 3.5 | 250% | ✔ | | ✔ |
| `embar` (256 iterations) | 4.6 | 1.0 | 4.0 | 300% | ✔ | ✔ | |
| **Perfect** | | | | | | | |
| `bdna` | 63.7 | 0.9 | 2.0 | 122% | ✔ | | ✔ |
| `qcd` | 9.6 | 0.9 | 0.9 | 0% | | | |
| `spec77` | 124.6 | 0.7 | 1.2 | 71% | | ✔ | |
| `trfd` | 21.1 | 1.7 | 1.9 | 12% | | | ✔ |

**Figure 6-6. Performance improvement due to reduction analysis on a 4-processor SGI Challenge**

In the case of the quantum chromodynamics code `qcd`, the parallelism coverage increases from 18.1% to 37.5% due to reductions. But automatic analysis only detects low coverage (below 40%) and low granularity (below 10 us) parallelism, which leads to no speedup at run-time. However, it is indeed a highly parallel program. Even though our static analysis is not strong enough to determine that the main loop in `qcd` is parallel, the analysis forms the basis for the SUIF Explorer, an interactive parallelization system described in Chapter 2. The analysis results is used to isolate the problematic areas and to focus the user's attention on them. In the program `qcd`, the Explorer finds two over 600-line interprocedural loops (`update/1` and `update/2`) that would be parallelizable if not for a small procedure. Examination of that procedure reveals that it is a random number generator, which a user can potentially modify to run in parallel. By requesting very little help from the user, the SUIF Explorer can parallelize the loop and perform all the tedious reduction and privatization transformations automatically. As a result, `qcd` achieves a speedup of 2.5 on a 4-processor Challenge.

Figure 6-7 presents the performance data on the Origin, a CC-NUMA machine also from Silicon Graphics Inc. The speedups for `mdljdp2`, `mdljsp2`, `ora`, `cgm`, and `embar` are quite significant, as compared with speedups of approximately 1 without reduction. The performance improvement is due to the sparse reductions and the interprocedural reductions. However, the performance of the program `bdna` does not improve as much as that on the SGI Challenge. The main reason is that all the data sets for the Perfect Club programs are too small to obtain good speedups on the Origin. For the NAS programs, we are able to use the large data set to obtain good speedups. The running time is too small otherwise.

Reduction recognition increases little or none of the performance on the other six programs, `su2cor`, `tomcatv`, `appbt`, `qcd`, `spec77`, `trfd`, for the similar reasons on the SGI Challenge. In summary, nine programs, `mdljdp2`, `mdljsp2`, `ora`, `appbt`, `cgm`, `embar`, `bdna`, `spec77`, `trfd`, benefit from parallelized reductions on the Origin.

| Program | Execution time of sequential version (seconds) | Speedups | | | Reasons for improvement | | |
|---|---|---|---|---|---|---|---|
| | | no reduc. analysis | use reduc. analysis | relative improve-ment | sparse reduc. | inter-proc. reduc. | intra-proc. reduc. |
| **SPEC92** | | | | | | | |
| `su2cor` | 53.2 | 1.2 | 1.2 | 0% | | | |
| `tomcatv` | 6.6 | 1.1 | 1.1 | 0% | | | |
| `mdljdp2` | 16.0 | 1.0 | 1.7 | 70% | ✔ | ✔ | |
| `mdljsp2` | 17.1 | 1.0 | 1.7 | 70% | ✔ | ✔ | |
| `ora` | 29.7 | 1.0 | 3.9 | 290% | | ✔ | |
| **NAS** | | | | | | | |
| `appbt` ($34^3$x$5^2$ grid) | 939.4 | 3.6 | 3.7 | 3% | | | ✔ |
| `cgm` (14,000 elem.) | 87.8 | 1.0 | 2.9 | 190% | ✔ | | ✔ |
| `embar` (65,536 iter.) | 1009.7 | 1.0 | 4.0 | 300% | ✔ | ✔ | |
| **Perfect** | | | | | | | |
| `bdna` | 18.7 | 0.9 | 1.1 | 22% | ✔ | | ✔ |
| `qcd` | 6.0 | 0.9 | 0.9 | 0% | | | |
| `spec77` | 70.4 | 0.8 | 1.0 | 25% | | ✔ | |
| `trfd` | 17.4 | 1.6 | 1.7 | 6% | | | ✔ |

**Figure 6-7. Performance improvement due to reduction analysis on a 4-processor SGI Origin**

## 6.6. Related Work

Reductions have been an integral component in the study of vectorizing and parallelizing compilers for many years[89][90][109]. More recently, reduction recognition approaches have been proposed that rely on symbolic analysis or abstract interpretation to locate many kinds of complex reductions[6][49][50]. However, it is unclear whether the significant additional expense of these approaches is justified by the types of reductions that appear in practice.

Previous array reduction algorithms need to constrain the array index function to be affine. Our algorithm can perform reductions even when the compiler cannot predict the locations that are written. The formulation of our reduction recognition algorithm is different from that used in previous compilers, and is powerful enough to allow our compiler to parallelize more cases. For example, although important, sparse array reductions are not being sufficiently exploited as a source of parallelism in today's parallelizing compilers. A typical commercial parallelizing compiler can parallelize reductions on scalar variables, but not sparse array reductions. Furthermore, a typical commercial compiler recognizes reductions only if the associative operation and the loop in which it is a reduction are contained within a single procedure. This limitation means that the algorithm may be giving up opportunities to parallelize coarse grain computations that span multiple procedures. Coarser grain computations are particularly beneficial when parallelizing reductions because the overhead of the reduction can be amortized over a larger parallel computation.

Our reduction recognition is most closely related to recent research by Pottenger and Eigenmann[92] in the Polaris compiler[21]. Our reduction recognition, in conjunction with the scalar symbolic analysis, is capable of locating the reductions described by Pottenger and Eigenmann. However, our work is distinguished by its ability to parallelize interprocedural and sparse reductions. It is difficult to make direct comparison between the two systems. For example, optimizations such as unused procedure elimination in SUIF, which eliminate some loops, and selective procedure inlining in Polaris, which creates copies of some loops, make the parallel loop counts different. The latest results from the Polaris compiler can be found in [18].

Reductions on array variables are quite common in scientific programs, but the associated costs of privatization and synchronization for a reduction on a whole array are much greater than for a scalar variable. Parallelizing array reductions can potentially lead to a significant performance degradation, as was reported in an earlier study by Blume and Eigenmann[20][36]. We use rotating locks to parallelize the global accumulation into an array. It is sometimes advantageous to further parallelize the global accumulation when executing on larger numbers of processors than the 4 processor systems used for this experiment. Rather than each processor accumulating directly into the global copy of the array as the SUIF run-time system does, the transformed code could instead perform updates of local copies on pairs of processors in *binary combining trees*, as proposed by Blelloch[17].

## 6.7. Chapter Summary

It is important that parallelizing compilers recognize reductions in sequential programs and replace the sequential reduction code by a parallel implementation. We have presented an interprocedural region-based algorithm for detecting reductions and an implementation of reductions that avoids significant overhead, that are part of the Stanford SUIF compiler. Our algorithm is distinguished by its ability to parallelize interprocedural and sparse reductions.

We have shown through extensive measurements that parallelized reductions are an important component of an automatic parallelizer. Reduction operations commonly occur in scientific codes; failure to parallelize them significantly limits the effectiveness of a parallelizing compiler.

Reduction transformations can potentially introduce substantial overhead as compared to parallel loops without reductions. Our reduction transformation recognizes certain opportunities for reducing overhead of initialization and global accumulation. In particular, if the reduction computation is performed on only a single location of an array, the transformation promotes the reduction to be performed on a scalar temporary instead. The reduction transformation implementation also uses the rotating lock mechanism described in Section 6.3.4 to reduce contention for accessing memory locations during the global accumulation

Finally, our experimental results show that many of the parallelizable loops do not require interprocedural reduction analysis. However, the coarse-grained loops parallelized with our reduction analysis often contain a significant portion of the overall computation of the program and, as shown in Section 6.5, can make a substantial difference in overall performance. Nine of the thirty programs in the three benchmark suites obtain substantial improvement. Thus, parallelized reductions is an essential component in a compiler for obtaining excellent parallel codes.

# 7 Conclusion

To overcome many of the physical limitations of uniprocessor performance, computer architects have been striving to design fast computers by connecting many small ones. Flynn observed that such multiprocessors have not achieved the predicted performance gains for general purpose computing, mainly because of the inability to create parallel software, either explicitly by a programmer or automatically by a compiler[44]. Parallel programs are hard to develop, difficult to debug, and expensive to maintain. The current generation of parallelizing compilers cannot extract parallel performance from sequential programs even with extensive user intervention. Thus, the adoption of parallel computing has been much slower than that anticipated[59].

We developed the SUIF Explorer, an interactive and interprocedural parallelizer, to increase the productivity of parallel programming and to exploit the multiprocessors efficiently. This thesis represents a step towards making multiprocessors accepted as general purpose computers. This thesis makes four major contributions: design of an interactive parallelizer, application of slicing to interactive parallelization, design and application of an interprocedural array liveness analysis, and design of an interprocedural array reduction analysis.

## 7.1. Design of the SUIF Explorer

This thesis presents the design of the SUIF Explorer and shows that the system is effective in assisting a programmer in finding coarse-grain parallelism in sequential programs. The Explorer minimizes the number of lines of code requiring manual examination using three techniques: advanced interprocedural parallelization, sophisticated dynamic execution analyzers, and program slicing.

The key to the success of the Explorer lies in having sufficiently powerful analyses that can restrict the need for user assistance to a small number of lines of code. It is critical that the SUIF compiler parallelize many of the loops automatically and leave only a few unresolved dependences in the remaining sequential loops.

The effectiveness of the system has been demonstrated on real-world applications such as `arc3d` from NASA Ames Research Center, `hydro` from Los Alamos National Laboratory, and `flo88` from the Center for the Integrated Turbulence Simulations at Stanford. The programmers need only examine a small portion of the programs; the compiler automatically parallelizes most of the variables used. The slicing algorithm requires the programmer to read only about 10% of the code in the loop in order to resolve a dependence. Finally, we obtain substantial speedups on all the applications.

## 7.2. Slicing for Interactive Parallelization

We show that the concept of program slicing can be applied effectively to interactive parallelization. Our context-sensitive slicing algorithm is successful both in reducing the number of lines that need to be analyzed and in minimizing the likelihood of human error.

## 7.3. Design and Application of an Array Liveness Analysis

We propose a context-sensitive and flow-sensitive interprocedural array liveness algorithm that efficiently analyzes the program. We show that liveness can be used to eliminate the need to finalize a privatizable array, to separate live ranges of array variables so their layouts can be optimized independently, and to enable array contraction. As an enabler of several optimizations, our efficient array liveness analysis is a key analysis that should be included in any modern parallelizing compiler.

Experimental results on real-world applications show that the algorithm is effective in finding many dead array variables at loop boundaries. The precision in the algorithm is important as we show that simpler versions omitting differentiation or ignoring the control flow within regions in the top-down phase yield inferior results. Finally, liveness information is effective in speeding up most of the programs in our suite.

## 7.4. Design of an Interprocedural Reduction Analysis

Our reduction algorithm extends beyond previous approaches in its ability to locate reductions to array regions, even in the presence of arbitrarily complex data dependences. The algorithm can locate interprocedural reductions, reduction operations that span multiple procedures. We evaluate the reduction algorithm and show that it speeds up many programs.

## 7.5. Future Work

### 7.5.1. SUIF Explorer for Optimizing Memory Performance

This thesis demonstrates that the SUIF Explorer can both help the user develop parallel codes effectively and help the compiler researcher develop the next-generation parallelizer. The Explorer can be extended to optimize memory performance. Using the memory effectively is often the key to high performance computing because microprocessor speeds have increased steadily at an annual rate of 50% to 100% in the last decade, while memory access time has improved at an annual rate of only 7%[59]. The generic architecture of the Explorer can be re-used, but the Guru for optimizing memory performance will focus on the data layout and computation distribution instead of parallelism. The experience with the SUIF Explorer suggests the importance of coupling compiler analyses with dynamic program profilers. The same experience also suggests that the programmer needs guidance in choosing the proper program transformations. Thus, using the decomposition analysis[8] and the affine partitioning framework[80] in the SUIF compiler as well as the FlashPoint tool[81], an efficient memory profiler on the FLASH multiprocessors[72], to provide guidance is important.

The success of the SUIF Explorer relies on whether the static and dynamic analyses provide quality information that gives meaningful assistance to the programmer. Thus, our goal is to develop analyses that can focus the user's attention on the key conflicting data decompositions and non-contiguous data accesses. Experimenting with real-world applications that have poor memory performance is essential. The Guru for optimizing memory performance should make effective suggestions on how to change the data layouts and how to co-

locate the elements of the array and their computation to the same processor in order to maximize the parallelism and minimize the communication.

Our existing assertion checker does not check all the errors. A more complete assertion checking tool can help minimize the likelihood of human errors. Since the user's input is error-prone, some analyses to help the user come up with the missing program properties should be developed. For instance, a dynamic tool which automatically uncovers the promising run-time properties can feedback the information to the user. Finally, an effective tool for optimizing the memory performance should be able to explain the reasons for bad performance using easy-to-understand memory models. This model should suggest the effects of data and computation transformations on both the uniprocessor and the multiprocessor.

### 7.5.2. SUIF Explorer for Optimizing Sparse Computations

In addition, the SUIF Explorer can be extended to handle sparse matrix computations, which arise in many domains of science and engineering. Various special-purpose data structures are used to store only the non-zero elements in the sparse matrix; thus, the sparse computation code is much more complex than the counterpart in dense computation. Developing sparse codes is an error-prone process. A higher-level specialized programming environment, such as the SUIF Explorer, could ease the task of sparse matrix programming. There are two areas of research in such tools. First, the programming environment can focus on the programming aspect. Such tools can provide higher-level interfaces such as the language constructs and data structures for sparse computation and then convert the user's specification into the dense matrix code. There exist several *parallel* sparse programming tools such as CHAOS from the University of Maryland[112] and LPARX from the University of California, San Diego[70]. Our Explorer system can be extended to support the generation of parallel sparse codes. Thus, the Explorer system can automatically perform such tedious tasks as inserting explicit synchronizations and communications. Typical existing systems can successfully handle the simple kernels; for example, matrix-vector or matrix-matrix multiplications. Yet much room remains in representing more complex algorithms such as parallel matrix factorizations.

Second, the high-level programming environment can focus on the performance aspect. It can optimize the performance of the sparse codes via parallelization or data-structure transformations. For example, we want the system to automatically remove redundant synchronizations. In addition, we notice that some sparse codes compute in phases. Specifically, the index arrays for the sparse matrices typically never change in certain phases. We can exploit such behavior in optimizing the code. The current SUIF Explorer is already able to optimize sparse computations to a limited degree. For instance, our reduction algorithm can parallelize sparse reductions.

This dissertation demonstrates that the collaboration with scientists on real-world applications can help make the system more useful and practical for parallelizing dense computation. A usability study needs to be conducted to bring the programming environment to the level where it is actively used for programming sparse matrix computations. We can start with the collaboration with the scientists who use sparse codes in their research.

# Bibliography


[1] V. S. Adve, J. Mellor-Crummey, M. Anderson, K. Kennedy, J.-C. Wang, and D. A. Reed, "An Integrated Compilation and Performance Analysis Environment for Data Parallel Programs." In *Proceedings of Supercomputing* 1995, San Diego, CA, November 1995.

[2] H. Agrawal and J. R. Horgan. "Dynamic program slicing." In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation* (PLDI), pages 246-256, White Plains, NY, June 1990.

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

[4] S. P. Amarasinghe. *Parallelizing Compiler Techniques Based on Linear Inequalities.* PhD thesis, Stanford Technical Report No. CSL-TR-97-712, Department of Electrical Engineering, Stanford University, Palo Alto, CA, January 1997.

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

[6] Z. Ammarguellat and W. Harrison, "Automatic recognition of induction variables and recurrence relations by abstract interpretation." In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation* (PLDI). White Plains, NY, June 1990.

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

[8] J. M. Anderson. *Automatic Computation and Data Decomposition For Multiprocessors*. PhD thesis, Stanford Technical Report No. CSL-TR-97-719, Department of Computer Science, Stanford University, Palo Alto, CA. March 1997.

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

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

[11] Applied Parallel Research, Documentation for FORGE Programming Tools. http://www.apri.com/document.html.

[12] D. Bacon, S. Graham, O. Sharp, “Compiler transformations for high-performance computing.” In *Computing surveys*, 26(4):345-420, December 1994.

[13] D. Bailey, E. Barszcz, J. Barton, D. Browning, R. Carter, L. Dagum, R. Fatoohi, S. Fineberg, P. Frederickson, T. Lasinski, R. Schreiber, H. Simon, V. Venkatakrishnan and S. Weeratunga. “The NAS parallel benchmarks.” NASA Ames Technical Report RNR-94-007, March 1994.

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

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

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

[17] G. E. Blelloch. “Prefix sums and their applications.” Technical Report CMU-CS-90-190, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA. November 1990.

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

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

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

[21] W. Blume, R. Eigenmann, K. Faigin, J. Grout, J. Hoeflinger, D. Padua, P. Peterson, W. Pottenger, L. Rauchwerger, P. Tu and S. Weatherford. “Polaris: The next generation in parallelizing compilers,.” In *Proceedings of the Seventh Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY, August 1994. Springer-Verlag Lecture Notes in Computer Science.

[22] R. P. Bosch, C. Stolte, D. Tang, J. Gerth, M. Rosenblum, and P. Hanrahan. “Rivet: A Flexible Environment for Computer Systems Visualization.” In *Computer Graphics*, February 2000.

[23] Z. Bozkus, A. Choudhary, G. Fox, T. Haupt, and S. Ranka. “A compilation approach for Fortran 90/D and High Performance Fortran compilers on distributed memory MIMD computers.” In Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing (LCPC), Portland, OR, August 1993. Springer-Verlag Lecture Notes in Computer Science.

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

[25] D. Callahan, K. Kennedy. “Analysis of interprocedural side effects in a parallel programming environment.” In *Journal of Parallel and Distributed Computing*, 5:517-550, 1988.

[26] S. Carr, K. S. McKinley, C.-W Tseng. “Compiler optimizations for improving data locality.” In Proceedings of the International Conference on Architectural Support ofr Programming Languages and Operating Systems, pp. 252-262. San Jose, CA, October 1994.

[27] D. Cheng and D. Pase. “An evaluation of automatic and interactive parallel programming tools.” In *Proceedings of Supercomputing*, Albuquerque, NM, November 1991.

[28] K. Cooper, M. W. Hall, R. T. Hood, K. Kennedy, K. S. McKinley, J. M. Mellor-Crummey, L. Torczon, and S. K. Warren. “The ParaScope parallel programming environment.” In *Proceedings of the IEEE*, Volume 81, Issue 2, pages 244–263, February 1993.

[29] D. Culler, J. P. Singh, A. Gupta. *Parallel Computer Architecture: A Hardware/ Software Approach.* Morgan Kaufmann Publishers, San Mateo, CA, 1999.

[30] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, F. K. Zadeck, “Efficiently computing static single assignment form and the control dependence graph.” In *ACM Transactions on Programming Languages and Systems*, Volume 13, Issue 4, pages 451-490, September 1991.

[31] R. Cytron and R. Gershbein, “Efficient accommodation of may-alias information in SSA form.” In *SIGPLAN ‘93 conference on programming language design and implementation*, pp. 36-45, 1993.

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

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

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

[35] S. G. Eick, J. L. Steffen, and E. E. Sumner Jr., “SeeSoft: a tool for visualizing line oriented software statis tics.” In *IEEE Transactions on Software Engineering*, 18(11):957-968, November 1992.

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

[37] M. Ernst, “Practical fine-grain slicing of optimized code,” *Technical Report MSR-TR-94-14*, Microsoft Research, Redmond, WA, July 1994.

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

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

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

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

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

[43] J. Ferrante, K. Ottenstein, J. D. Warren, “The program dependence graph and its use in optimization.” In *ACM Transactions on Programming Languages and Systems*, 9(3), pp. 319-349, July 1988.

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

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

[46] E. D. Granston and A. Veidenbaum. “Detecting redundant accesses to array data.” In *Proceedings of Supercomputing ’91*, Albuquerque, New Mexico, pages 108-119, November 1991.

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

[48] R. Gupta, M. L. Soffa. “Hybrid slicing: an approach for refining static slicing using dynamic information.” In T*he Foundationsof Software Engineering*, pp. 29-40, September, 1995

[49] M. Haghighat and C. Polychronopoulos. “Symbolic analysis: A basis for parallelization, optimization, and scheduling of programs.” In *Proceedings of the Sixth Workshop on Languages and Compilers for Parallel Computing*, Portland, OR, August 1993. Springer-Verlag Lecture Notes in Computer Science.

[50] M. Haghighat and C. Polychronopoulos. “Symbolic analysis for parallelizing compilers.” In *ACM Transactions on Programming Language and Systems*, Volume 18, Issue 4. July 1996.

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

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

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

[54] M. W. Hall, T. J. Harvey, K. Kennedy, N. McIntosh, K. S. McKinley, J. D. Oldham, M. H. Paleczny, G. Roth, "Experiences using the ParaScope Editor: an interactive parallel programming tool." In *Proceedings of the Principles and Practices of Parallel Programming* '93, pp. 33-43, May 1993.

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

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

[57] W. L. Harrison. "The interprocedural analysis and automatic parallelization of Scheme programs." *Lisp and Symbolic Computation*, Volume 2, Issue 3, pages 179–396, October 1989.

[58] P. Havlak. *Interprocedural symbolic analysis*. PhD thesis, Department of Computer Science, Rice University, Houston, Texas, May 1994.

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

[60] High Performance Fortran Forum. "High Performance Fortran Language Specification," Version 2.0, ftp://softlib.rice.edu/pub/HPF. January 1997,

[61] High Performance Fortran Forum. "High Performance Fortran Language Specification." Version 1.0. In *Scientific Programming*, Volume 2, Issue 1 and 2, pages 1–170, January, 1993.

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

[63] S. Horwitz, T. Reps, D. Binkley, "Interprocedural slicing using dependence graphs." In *ACM Transactions on Programming Languages and Systems*, 12(1), pages 26-50, January 1990.

[64] S. Horwitz, T. Reps, M. Sagiv, “Demand interprocedural dataflow analysis.” In *Proceedings of the third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 104-115, 1995.

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

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

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

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

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

[70] S. R. Kohn. A Parallel Software Infrastructure for Dynamic Block-Irregular Scientific Calculations. PhD thesis, Department of Computer Science and Engineering, University of California, San Diego. June 1995.

[71] Kuck & Associates Incorporated, Documentation for the KAP/Pro Toolset. http://www.kai.com.

[72] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. “The Stanford FLASH Multiprocessor.” In *Proceedings of the 21st International Symposium on Computer Architecture (ISCA)*, pages 302-313, April 1994.

[73] W. Landi and B. Ryder. “A safe approximate algorithm for interprocedural pointer aliasing.” In *SIGPLAN ’92 Conference on Programming Language Design and Implementation*, pp. 235-248, June 1992.

[74] L. Larsen, M. J. Harrold, “Slicing object-oriented software,” *Technical Report 95-103*, Department of Computer Science, Clemson University, March 1995.

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

[76] E. C. Lewis, C. Lin, L. Snyder. "The implementation and evaluation of fusion and array contraction in array languages." In *Proceedings of the ACM SIGPLAN'98 Conference on Programming Language Design and Implementation*, pp. 50-59. Montreal, Canada, June 1998.

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

[78] S.-W Liao, A. Diwan, R. P. Bosch, Jr., A. Ghuloum, M. S. Lam. "SUIF Explorer: an interactive and interprocedural parallelizer." In *Proceedings of the 7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 37-48, Atlanta, Georgia, May 1999.

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

[80] A. W. Lim, G. I. Cheong, and M. S. Lam. "An affine partitioning algorithm to maximize parallelism and minimize communications." In Proceedings of the 13th ACM SIGARCH International Conference on Supercomputing, June 1999.

[81] M. Martonosi, D. Ofelt, and M. Heinrich. "Integrating performance monitoring and communication in parallel computers." In ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems, pp. 138-147, May 1996

[82] D. E. Maydan. *Accurate Analysis of Array References*. PhD thesis, Department of Computer Science, *Stanford Technical Report* CSL-92-544. Stanford University, August 1992.

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

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

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

[86] T. Munzner, “Exploring large graphs in 3D hyperbolic space.” In *IEEE Computer Graphics and Applications*. Volume 18 Number 4. IEEE Computer Society, July/ August 1998.

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

[88] J. Oplinger, D. Heine, S.-W. Liao, B. Nayfeh, K. Olukotun, M. Lam, “Software and Hardware for Exploiting Speculative Parallelism with a Multiprocessor,” *Stanford Technical Report* CSL-97-715.

[89] D. Padua and M. Wolfe, “Advanced compiler optimizations for supercomputers.” In the *Communications of the ACM*, Volume 29, Issue 12, ACM. December 1986, pages 1184-1201.

[90] S. S. Pinter and R. Y. Pinter. “Program Optimization and Parallelization Using Idioms.” In *Proceedings of Eighteenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, Orlando, Florida, January 1991, pages 79-92.

[91] L. Pointer, “Perfect: Performance evaluation for cost effective transformations report 2,” *Technical Report* 964, University of Illinois, Urbana-Champaign, Urbana, Illinois, March 1990.

[92] B. Pottenger and R. Eigenmann, “Parallelization in the presence of generalized induction and reduction variables.” In *Proceedings of the 1995 ACM International Conference on Supercomputing*, June 1995.

[93] T. Reps, S. Horwitz, M. Sagiv, G. Rosay, “Speeding up slicing.” In *Proceedings of the second ACM SIGSOFT Symposium on Foundations of Software Engineering*, pp 11 - 20, 1994.

[94] V. Sarkar, G. Gao, “Optimization of array accesses by collective loop transformations,” pp. 194-205, *International Conference on Supercomputing*, June 1991

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

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

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

[98] B. Steensgaard, “Points-to analysis in almost linear-time.” In *ACM Symposium on Principles of Programming Languages*, pp. 32-41, January 1996

[99] F. Tip, “A survey of program slicing techniques,” Centrum voor Wiskunde en Informatica, Report CS-R9438, July 1994.

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

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

[102] C.-W. Tseng. “Compiler optimizations for eliminating barrier synchronizations.” In *Proceedings of the Fifth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPOPP ’95)*, pages 144-155, Santa Barbara, CA, July 1995.

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

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

[105] J. Uniejewski. “SPEC Benchmark Suite: Designed for today’s advanced systems.” SPEC Newsletter Volume 1, Issue 1, SPEC, Fall 1989.

[106] M. Weiser, “Program slicing.” In *IEEE Transactions on Software Engineering*, 10(4), pp. 352-357, 1984

[107] R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy. "SUIF: A parallelizing & optimizing research compiler." Technical Report CSL-TR-94-620, Computer Systems Laboratory, Stanford University, May 1994.

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

[109] M. E. Wolf. *Improving Locality and Parallelism in Nested Loops*. PhD thesis, Department of Computer Science, *Stanford Technical Report* CSL-92-540, Stanford University, Palo Alto, CA. August 1992.

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

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

[112] J. Wu, R. Das, J. Saltz, H. Berryman, and S. Hiranandani. "Distributed Memory Compiler Design for Sparse Problems." In *IEEE Transactions on Computers*, pages 737-753, Volume 44, Number 6, June 1995.

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

[114] Z. Xu, J. Larus, B. P. Miller, "Shared-Memory Performance Profiling." In *Proceedings of the Principles and Practices of Parallel Programming* '97, pp. 240-251, Las Vegas, NV, June 1997

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

# Index