INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI
A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700  800/521-0600
Prescriptive Performance Tuning: The $R_\infty$ Approach

by

Ramakrishnan Rajamony

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:

Willy E. Zwaenepoel
Willy E. Zwaenepoel, Chair
Professor in Computer Science and Electrical and Computer Engineering

Alan L. Cox, Director of Research
Associate Professor of Computer Science

Sarita V. Adve
Assistant Professor in Electrical and Computer Engineering

John M. Mellor-Crummey
Faculty Fellow, Department of Computer Science

Houston, Texas
January, 1998
Copyright
Ramakrishnan Rajamony
1998
Prescriptive Performance Tuning: The $R_X$ Approach

Ramakrishnan Rajamony

Abstract

Programmers often rely on performance analysis tools to provide feedback about the execution of their applications. However, the nature of this feedback is far from satisfactory. Often the feedback is purely descriptive and at a very low-level, making it difficult for the programmer to rectify performance problems.

This dissertation demonstrates a new approach to performance tuning: prescriptive performance debugging. Our approach can greatly reduce the burdens imposed on the programmer compared to existing performance analysis tools. The basis of this approach is a set of requirements that must be satisfied by a performance analysis tool. In problem domains where these requirements can be met, a performance tool can prescribe source-level changes to improve performance.

$R_X$ is one such tool that we have developed to improve the performance of explicitly parallel shared memory programs. $R_X$ targets inter-process synchronization and data communication, two significant sources of overhead in shared-memory applications. $R_X$ automatically analyzes run-time data from program executions to prescribe transformations that reduce synchronization and some forms of data communication. This feedback is at the source-code level, eliminating the need for machine-level reasoning about the program. A correctness framework ensures that transformations obtained from one or a small set of executions will be applicable to all executions. In a few cases, feedback from $R_X$ has made a crucial difference, enabling applications that were originally slowing down on multiple processors to achieve a speedup.

In summary, this dissertation makes three contributions: (i) A new approach for designing performance tools, enabling the prescription of source-level changes to improve performance, (ii) a set of algorithms to detect excess synchronization and some forms of excess data communication in explicitly parallel shared memory programs, and (iii) a set of low-overhead techniques to collect run-time information for performance tuning.
Acknowledgments

This dissertation is the culmination of many years of research in the Systems group at Rice. I am indebted to Alan Cox for giving me the opportunity to be part of such a first-class research environment.

As my advisor, Alan has been an incredible font of guidance, advice, and encouragement. He has been a wonderful mentor, giving me the freedom to pursue my own research directions, while providing me with the benefit of his experience, insight and keen problem-solving skills. I will sorely miss his counsel when I leave Rice. Willy Zwaenepoel has been my other mentor. His ability to step back and look at the “big picture” has been extremely useful, and I hope I have learnt how to do that from him. Together, Alan and Willy have taught me what constitutes quality research and the importance of directing it towards a long-term goal.

Sarita Adve and John Mellor-Crummey, the other members of my committee, have also given me valuable advice and feedback. Their constructive comments and incisive questions have improved the overall quality of this work. John also suggested the term prescriptive feedback, which I have since appropriated from him.

Joe Cavallaro gave me a lot of support during my first year at Rice, and I am very grateful to him for that. I am also grateful to IBM for providing me with a generous fellowship during the last two years of my studies. Michael Scott provided me with access to an SGI multiprocessor in his research group when I needed it most. The results of my experiments on this machine, though not included in this dissertation, helped shape my early research. Iva Jean Jorgensen has been instrumental in enabling me submit this dissertation to the graduate school on time.

My friends: Karthick, Raghu, Mike, Cristiana, Mohit, Johnny, Honghui, Vivek, Weimin, Evan and others have made my life here very enjoyable. I am glad I had their company. Mohit and Honghui have also been wonderful sounding boards for some of my wackier ideas. I have also enjoyed discussing my work with Doug Moore, Mark Krentel, and Devika Subramanian.

Finally, I would like to express my sincere gratitude to my parents for enabling me to be what I am today. This degree is as much theirs as it is mine. To my wife Brinda: I could not have done this without you.
To Amma, Appa, and Brinda
Contents

Abstract ii
Acknowledgments iii
List of Illustrations ix

1 Introduction 1
   1.1 Thesis Statement ........................................... 2
   1.2 Contributions ............................................. 2
   1.3 Dissertation Overview ................................... 4

2 Prescriptive Performance Tuning 6
   2.1 Introduction ............................................... 6
   2.2 Requirements for Providing Prescriptive Feedback ............ 7
   2.3 Designing Prescriptive Tools ................................ 8
      2.3.1 Reducing Synchronization and Communication ............. 8
      2.3.2 Improving the Cache Behavior of Sequential Programs ... 9
      2.3.3 Optimizing Message Passing Programs .................. 10
      2.3.4 A Counter-Example ..................................... 11
   2.4 Related Work .............................................. 11
   2.5 Summary .................................................. 14

3 $R_X$: A Prescriptive Performance Debugger 15
   3.1 Introduction .............................................. 15
   3.2 The Motivation Behind $R_X$ ................................ 17
   3.3 Why Use a Run-Time Approach? ............................ 19
   3.4 The Drawback of a Run-Time Approach ...................... 19
   3.5 Extending the Scope of Transformations .................... 20
   3.6 $R_X$ Prescriptions: An Example ......................... 23
   3.7 Summary .................................................. 24
4 Reducing Synchronization and Communication

4.1 Introduction .................................................. 25
4.2 Assumptions .................................................. 26
4.3 The Partial–Order Graph .................................... 27
4.4 Detecting Unnecessary Synchronization ................. 27
  4.4.1 Barriers .................................................. 28
  4.4.2 Pairwise Synchronization ............................ 31
  4.4.3 Critical Sections ....................................... 34
4.5 Transforming Synchronization .............................. 37
  4.5.1 Aggregating Critical Sections ....................... 38
  4.5.2 Vectorizing Critical Sections ....................... 39
  4.5.3 Detecting ns–atomic Critical Sections .............. 39
4.6 Restructuring Computation ................................. 43
  4.6.1 Postponing Computation ............................... 46
  4.6.2 Relocating Computation ............................... 47
4.7 Related Work ................................................ 48
4.8 Summary ..................................................... 52

5 Preprocessing and Data Collection .......................... 54

5.1 Introduction .................................................. 54
5.2 High–Level Requirements .................................. 55
5.3 Conflict Analysis ............................................ 56
  5.3.1 Procedure Calls ....................................... 57
5.4 Moving Computation Outside Critical Sections ......... 57
  5.4.1 Processing the Source Line Information ............. 59
  5.4.2 Limitations ............................................. 60
5.5 Detecting ns–atomic Critical Sections .................... 62
5.6 Determining Computation Restructuring .................. 63
  5.6.1 Equivalence Classes .................................. 64
  5.6.2 Collecting Source Operands ......................... 66
  5.6.3 Collecting Address Traces for Basic Blocks ......... 69
  5.6.4 Converting Control to Data Dependences ........... 69
  5.6.5 Using Mirror Functions for the Run–time Processing 73
5.7 Perturbation .................................................. 75
6 Postprocessing and Data Analysis

6.1 Introduction ........................................... 78
6.2 Detecting Unnecessary Synchronization ................. 79
   6.2.1 Barriers ........................................ 79
   6.2.2 Pairwise Synchronization ...................... 80
   6.2.3 Critical Sections .............................. 81
6.3 Transforming Synchronization .......................... 81
   6.3.1 Aggregating and Vectorizing Critical Sections ... 82
6.4 Restructuring Computation ............................ 82
   6.4.1 Postponing Computation ....................... 82
   6.4.2 Relocating Computation ....................... 84
   6.4.3 Limitations .................................... 84
6.5 Extending the Scope of Transformations ............... 85
   6.5.1 Eliminating and Transforming Excess Synchronization ... 85
   6.5.2 Restructuring Computation .................... 86
6.6 Summary .............................................. 87

7 Results

7.1 Introduction ........................................... 88
7.2 Gross Application Characteristics .................... 89
7.3 Excess Synchronization Detected ..................... 90
   7.3.1 Barnes–Hut ..................................... 91
   7.3.2 Conjugate Gradient ......................... 92
   7.3.3 Gauss ........................................ 93
   7.3.4 Multigrid ..................................... 93
   7.3.5 Shallow ..................................... 94
   7.3.6 Water ....................................... 95
7.4 Performance Impact ................................... 95
   7.4.1 Hardware Shared Memory ...................... 96
   7.4.2 Software Shared Memory System ............... 98
7.5 Cost of Performance Debugging ....................... 101
   7.5.1 Experimental Platform ....................... 101
7.5.2 Performance Debugging Costs .......................... 101
7.5.3 Reducing the Performance Debugging Costs ............ 104
7.6 Summary ................................................. 105

8 Conclusions and Future Directions .......................... 107
  8.1 Conclusions ............................................. 107
  8.2 Future Directions ....................................... 109
    8.2.1 Prescriptive Tools for Other Domains ............... 109
    8.2.2 Augmenting Run-time Analysis with Compile-time Analysis 111

A Terminology for Chapter 3 .................................. 112
  A.1 Preliminary Definitions ................................ 112
  A.2 Final Definitions ...................................... 113

Bibliography .................................................. 115
Illustrations

3.1 A high-level overview of the $R_X$ performance debugging approach. . . . 16
3.2 Excess synchronization examples. ................................................. 17
3.3 Example illustrating the one-way implication. .............................. 20
3.4 Using system metaknowledge. ..................................................... 21
3.5 Illustrating the contract between $R_X$ and the programmer. ............. 22
3.6 Three $R_X$ prescriptions for Conjugate Gradient. ........................ 23

4.1 A portion of the partial-order graph for processes $P_a$ and $P_b$. ...... 28
4.2 A code fragment and its partial-order graph. ................................. 32
4.3 Reducing time spent inside a critical section. ............................... 34
4.4 Illustrating terminology. .......................................................... 35
4.5 Eliminating critical sections. ...................................................... 37
4.6 Two common ways of using critical sections. ............................... 38
4.7 The three conditions for a critical section to be ns-atomic. .............. 40
4.8 Testing for ns-atomicity in the presence of pairwise synchronization. . 11
4.9 Testing for ns-atomicity in the presence of multiple sets of critical sections. ........................................................................................................ 42
4.10 The partial-order graph for an execution of the code in Figure 4.9. . . 42
4.11 Eliminating synchronization by postponing computation. ............... 44
4.12 Eliminating communication and synchronization by relocating computation. ......................................................................................... 45
4.13 The two conditions that must be satisfied to postpone computation. 46
4.14 Relocating computation. ............................................................. 47

5.1 The run-time algorithm to determine if accesses can be moved outside critical sections. ............................................................. 58
5.2 Limitations of the run-time algorithm to move computation outside critical sections. ................................................................. 61
5.3 The run-time algorithm to check for the third condition from Figure 4.7. .......................... 62
5.4 Using equivalence classes to represent operand information. ........................................ 65
5.5 The SRCOPS run-time algorithm used for obtaining dependence information. .............. 67
5.6 A code fragment illustrating control dependences. ....................................................... 70
5.7 Algorithm executed after a controlling basic block. ............................................... 71
5.8 Algorithm executed before the immediate postdominator of a controlling basic block. .... 71
5.9 A control flow graph illustrating the working of the algorithms in figures 5.7 and 5.8. ........ 72
6.1 Rx algorithm to detect excess barriers. ................................................................. 79
6.2 Rx algorithm to detect excess pairwise synchronization. ............................................. 80
6.3 Rx algorithm to detect ns-atomic critical sections. ................................................... 81
6.4 Rx algorithm to determine whether computation can be postponed. ......................... 83
6.5 Rx algorithm to determine whether computation should be relocated. ....................... 83
Chapter 1

Introduction

Programmers often rely on performance analysis tools to improve the performance of their applications. Typically, these performance debuggers provide feedback about the program execution, such as the time spent in different routines, and the factors that slow down progress, such as cache misses. However, the nature of this feedback is far from satisfactory. Feedback, such as the time spent in different routines, is purely descriptive. Furthermore, the gap between machine-level feedback such as cache misses, and the source program, is considerable. Consequently, the cause-and-effect relationship between the source code and the tool feedback is difficult to ascertain. This makes it hard for the user to infer the true cause for poor performance.

This dissertation demonstrates a new approach to performance tuning: prescriptive performance debugging. This approach can greatly reduce the burdens imposed on the programmer compared to existing performance analysis tools. The basis of our approach is a set of requirements that must be satisfied by a performance analysis tool. In problem domains where these requirements can be met, a performance tool can prescribe source-level changes to improve performance.

$R_X^*$ is one such tool that we have developed to improve the performance of explicitly parallel shared-memory programs. $R_X$ targets inter-process synchronization and data communication, two significant sources of overhead in shared-memory applications. $R_X$ automatically analyzes run-time data from program executions to prescribe transformations. When incorporated into the source program, these prescriptions improve performance. The feedback is entirely at the source-code level, eliminating the need for machine-level reasoning about the program. A correctness framework ensures that $R_X$ prescriptions obtained from one or a small set of executions, will be applicable to all executions.

Our findings corroborate the usefulness of designing tools using our approach. $R_X$ was able to successfully performance tune complex, pointer-ridden applications, pro-

$^*$R$_X$ is a symbol used at the beginning of medical prescriptions and literally means “take”.
viding source-level changes that reduced synchronization and data communication. In a few cases, feedback from $R_X$ made a crucial difference, enabling applications that were originally slowing down on multiple processors to achieve a speedup.

1.1 Thesis Statement

The central hypothesis of this dissertation is that for several classes of problems, a performance tool can automatically analyze run-time data and prescribe source-level transformations that will improve performance.

In order to substantiate this thesis, we develop $R_X$, a prescriptive performance tool that targets synchronization and data communication in explicitly parallel shared-memory programs. By prescribing source-level transformations for reducing inter-process interactions, $R_X$ overcomes the problems of existing performance tools. Thus, it can be used easily by inexperienced parallel programmers.

1.2 Contributions

Contribution 1 A new approach for designing performance tools, enabling the prescription of source-level changes to improve performance.

Tools currently used by programmers to improve application performance suffer from two problems.

First, existing tools [BPR96, RAN+93] only describe performance information gathered at run-time, placing the onus on the user to deduce the cause for the performance problems. This requires users to understand how the program is mapped onto the parallel machine and also forces them to reason about machine-level execution statistics. Current performance debuggers thus do little beyond packaging raw information gathered at run-time.

Further, in many programming systems, the program executed by the hardware is very different from the one written by the programmer. For instance, software distributed shared memory systems present a shared memory abstraction to the programmer, even though the underlying hardware uses message passing. In such systems, it is difficult to correlate raw information and metrics gathered at the level of the hardware (such as message sizes and counts), with the source program. Without this correlation, a performance tool's output becomes difficult to interpret.
In several domains, performance tools developed using our approach can overcome these disadvantages. By satisfying a set of requirements that we have specified, a performance tool can prescribe source–level changes to improve performance. These requirements are:

- Automatically analyze run–time data to derive feedback
- Correlate the feedback with the source program
- Provide a framework to establish correctness of the feedback

By satisfying these requirements, prescriptive tools can be developed for a large set of problem domains, such as reducing inter–process interactions in concurrent programs, improving the cache behavior of sequential programs by changing the data layout, and indicating the best communication primitives to use in message–passing programs. The advantage of basing a tool on this approach is that it can be used by novice programmers to rectify performance problems, while requiring only source–level, as opposed to architecture–related, reasoning about the program.

**Contribution 2** A set of algorithms to detect excess synchronization and some forms of excess data communication in explicitly parallel shared–memory programs.

In a shared–memory program, if conflicting accesses (read–write or write–write accesses to the same memory location) [SS88] are not ordered by synchronization, time–dependent bugs may be introduced. Consequently, the program behavior can be unexpected or even incorrect. Faced with the possibility of these race conditions [Net91], most inexperienced parallel programmers add more synchronization than necessary to their applications. By over–synchronizing, the programmer can often ensure that the application is free of data races without an in–depth understanding of its sharing pattern. This is especially true when the programmer is not writing the application from scratch, but is instead parallelizing or porting it. Because of the overheads imposed by synchronization, the performance of such applications can be improved, often substantially, by removing unnecessary synchronization.

We present a set of algorithms to detect excess synchronization ranging from pairwise synchronization constructs such as flags, to global synchronization constructs such as barriers. The parallelizing compiler community has been addressing the problem of reducing synchronization in parallel programs [OB95, PH95, Tse95]. However,
most of their algorithms are tailored to detect and handle excess barriers introduced during compilation.\textsuperscript{1} In contrast, we present algorithms to detect and transform barriers, pairwise synchronization, and critical sections. We explain these algorithms using a framework introduced by Adve [Adv93] for reasoning about memory models.

We also present new algorithms to reduce synchronization and data communication by restructuring computation. These algorithms are based on two new computation transformations we have developed: \textit{postponing} and \textit{relocating} computation.

\textbf{Contribution 3} A set of low-overhead techniques to collect run-time information for performance tuning.

Several challenges need to be surmounted in order to prescribe transformations to reduce synchronization and data communication using a run-time approach. Two of the most important issues are what information to collect at run-time, and dynamically finding out the source operands of writes to memory. We have developed new methods to tractably address these problems.

In order to reduce the amount of data collected at run-time, we collect information in terms of \textit{equivalence classes}. Subsequently, we use this information to regenerate the data required to determine the prescriptive feedback. We also use a novel method for computing the source operands of writes at run-time. At run-time, we could have interpreted information about the program obtained by analyzing the program source. Instead, when analyzing the program source, we generate specialized functions that when executed at run-time, provide the necessary information. This considerably speeds up the collection of run-time data. This method can also be used to speed up program analysis tools such as ATOM [SE94], which instrument an application to collect run-time information.

\subsection{1.3 Dissertation Overview}

This dissertation is organized as follows.

Chapter 2 describes our new approach for designing performance tools. We provide three examples to illustrate how this approach can be used to develop prescriptive tools. Chapter 3 then introduces \textit{Rx}, a prescriptive tool for reducing synchronization

\textsuperscript{1}The only exception to this that we are aware of is the work by Diniz and Rinard [DR97], which we describe in Chapter 4.
and data communication in shared-memory parallel programs. \( R_X \) demonstrates our prescriptive approach to performance tuning.

Chapter 4 describes a set of algorithms for detecting excess synchronization in shared-memory programs. We also present algorithms for reducing synchronization and some forms of data communication by restructuring computation. These algorithms are based on our observations of excess synchronization and data communication in a large set of shared-memory applications. Although we implement these algorithms in a run-time tool, they can also be used in a compiler.

Several challenges need to be surmounted before a run-time tool can prescribe transformations that reduce synchronization and data communication. Two of the most important issues are what information to collect at run-time, and dynamically finding out the source operands of writes to memory. Chapter 5 describes the techniques we use to overcome these and other problems. Chapter 6 then describes the algorithms used by \( R_X \) to process the information gathered at run-time. These algorithms produce the source-level, prescriptive transformations that are then supplied to the programmer.

In Chapter 7, we provide results demonstrating the feasibility of using our approach to design a prescriptive performance tool. First, we present the feedback provided by \( R_X \) for a broad class of applications derived from benchmarks, course programming projects and an industry source. Then, we judge the usefulness of the feedback by evaluating the performance of the transformed applications on both hardware and software shared memory platforms. We also discuss the overheads associated with the performance tuning process.

Finally, our conclusions and plans for future work are discussed in Chapter 8.
Chapter 2

Prescriptive Performance Tuning

2.1 Introduction

Programmers often rely on performance analysis tools to improve the performance of their programs. Typically, these tools provide feedback about the program execution, such as the time spent in different routines, and the factors that slow down progress, such as cache misses. However, the nature of this feedback is far from satisfactory.

First, existing tools only describe the performance information gathered at runtime [BPR96, RAN+93]. For example, a performance tool applied to a program that suffers from poor cache behavior could say:

70% of all cache misses occur in function foo()

The onus is then on the user to infer the cause for the performance problems. This requires users to understand how the program is mapped onto the machine and also forces them to reason about low-level execution statistics. Current performance debuggers thus do little beyond packaging raw information gathered at run-time.

Further, in many programming systems, the program executed by the hardware is very different from the one written by the programmer. For instance, software distributed shared memory systems present a shared memory abstraction to the programmer, even though the underlying hardware uses message passing. In such systems, it is difficult to correlate raw information and metrics gathered at the level of the hardware (such as message sizes and counts), with the source program. Without this correlation, a performance tool's output becomes difficult to interpret.

This chapter presents a new approach for building performance tools that can overcome these problems in several domains. The basis of our approach is a set of requirements that must be satisfied by a performance analysis tool. In problem domains where these requirements can be met, a performance tool can prescribe source-level changes to improve performance. Such a tool, when applied to the application that suffers from poor cache behavior, would prescribe:

Change data structure bar as follows ...
Our approach can be used to design prescriptive tools for a wide variety of purposes. The advantage of such a tool is that it can be used by novice programmers to rectify performance problems, while requiring only source-level, as opposed to architecture-related, reasoning about the program.

The rest of this chapter is organized as follows. In section 2.2 we describe the basic requirements that when satisfied, enable a performance tool to provide prescriptive feedback. Next, in section 2.3, we provide three examples illustrating how our approach can be used to develop prescriptive tools. We discuss related work in section 2.4, and summarize the chapter contributions in section 2.5.

2.2 Requirements for Providing Prescriptive Feedback

A performance tool must satisfy three requirements in order to provide prescriptive feedback. These requirements are:

1. Automatically analyze run-time data to derive feedback.
   This is the first step that a tool must perform. Automatically analyzing data from the execution enables a tool to detect performance problems. Once a performance problem has been detected, a prescriptive tool must also determine a solution to it.

   The exact methods used to carry out this step depend on the problem domain in which the tool is being applied. Consider a tool being developed to reduce false sharing [TLH94] in shared-memory programs. False sharing can be detected by analyzing the memory access trace from the application. Once false sharing has been detected, a possible solution is to rearrange the layout of the data structures in the address space, enabling better use of the cache hierarchy.

2. Correlate the feedback with the source program.
   Once the solution for the performance problem has been determined, the next step is to correlate it with the source program. This step is extremely important, for it enables the programmer to reason about the program purely at the source level. Correlation can be especially problematic when the run-time system and the programmer see different views of the program. Software distributed shared memory systems are an example: the programmer writes a shared-memory program, but the hardware executes a message passing program.
3. Provide a framework to establish correctness of the feedback.

The prescriptive feedback derived by the tool is based on one or a small set of executions. Hence, in some domains, it may not be applicable to all executions of the program. In such cases, the performance tool must provide a framework that will allow the programmer to reason about the correctness of the feedback.

Satisfying these requirements will only enable a tool to prescribe feedback. Applying the prescribed transformations to the program is the responsibility of the programmer. In specific cases, a tool may be able to automatically apply the transformations (see Section 2.3.2 for instance). In general, this is not possible because programmer intervention may be required in order to ensure that the transformations, obtained by analyzing run-time data, are applicable to all executions.

2.3 Designing Prescriptive Tools

The requirements described in section 2.2 can be used as the basis for designing prescriptive tools for a wide variety of purposes. We provide three examples in this section to illustrate the general approach.

2.3.1 Reducing Synchronization and Communication

In a shared-memory program, time-dependent bugs may be introduced if conflicting accesses (read-write or write-write accesses to the same memory location) [SS88] are not ordered by synchronization. In the presence of these race conditions [Net91], the program behavior can be unexpected or even incorrect. Faced with this possibility, most inexperienced parallel programmers add more synchronization than necessary to their applications. This is especially true when the programmer is not writing the application from scratch, but is instead parallelizing or porting it. By over-synchronizing, the programmer can often ensure that the application is free of data races without an in-depth understanding of its sharing pattern. Because of the overheads imposed by synchronization, the performance of such applications can be improved, often substantially, by removing unnecessary synchronization.

The requirements specified in section 2.2 can be used to design a performance tool to determine instances of excess synchronization. Such a tool can prescribe source-level changes to reduce synchronization.

Excess synchronization can be detected by analyzing the accesses made by different processes. Synchronization is necessary only in the presence of conflicting
accesses. Once excess synchronization is detected, performance can be improved by just removing it. Automating this analysis satisfies the first requirement. The second step is to ensure that the feedback provided to the programmer is in terms of lines in the source program. The final step is to provide a framework that allows the programmer to determine that program correctness will not be affected by removing the synchronization.

In the rest of this dissertation, we develop this example in detail. We describe $R_X$, a tool we have developed that prescribes transformations to reduce synchronization and some forms of data communication in explicitly parallel shared-memory programs. In a few cases, feedback from $R_X$ has made a crucial difference, enabling applications that were slowing down on multiple processors to achieve a speedup.

### 2.3.2 Improving the Cache Behavior of Sequential Programs

It is well known that the memory performance of a sequential program is affected by the placement of its data structures in memory. Changing this layout can reduce conflict misses and also increase implicit prefetching by improving spatial locality. Consequently, the application performance can be improved.

Existing tools that target memory performance [GH93, LW94, MGA95, SB94] present only descriptive information, such as cache statistics, to the user. In contrast, a prescriptive tool can treat this as an optimization problem where the goal is to optimally arrange the program data structures and directly specify the best layout to use.

The first step is to analyze an access trace from the application and determine an alternate arrangement of the program data structures. The problem of finding a suitable arrangement that best utilizes the cache can be cast as a variant of the graph coloring problem. Nodes in the graph correspond to the program data structures. Edges represent temporally interleaved accesses to data. The problem is to assign addresses (i.e., cache lines) to the nodes, such that two data structures connected by an edge fall on distinct cache lines. An optimal solution can be found by formulating this as an integer programming problem and solving it using one of the many solvers (such as CPLEX [Bix92]) that are currently available.

The second step is to correlate this feedback with the source program, so that it can be provided in terms of the data structure names used by the programmer. In the case of rearrangements to statically allocated data structures, the tool can directly
provide a mapfile to the loader, obviating the need for programmer involvement. This is an instance where the tool can itself apply the transformations it prescribes. Dynamic data structures need to be handled differently, possibly by specifying the creation of different memory "pools", and informing the programmer of the particular pool from which to allocate each heap-based data structure. Each memory pool can be made to map to a particular portion of the cache by also specifying a mapping between virtual and physical memory to the operating system.

If the programming language permits variable aliasing*, program correctness may be affected by changing the internal arrangement of data structures. The debugger must then provide the means to determine (possibly through a dialogue with the programmer) that program correctness will not be affected. An approach similar to the one we take for establishing the correctness of $R_\alpha$ transformations (see Section 3.5) can be used for this purpose.

2.3.3 Optimizing Message Passing Programs

Our approach can also be used to design tools that facilitate the development and porting of message passing programs. With the increasing complexity of message-passing interfaces, the task of choosing the appropriate communication primitive is not easy. For instance, MPI [Mes95], a widely implemented message-passing interface standard, specifies around 100 primitives for communication, including several complex ones for group communication.

The large selection of communication primitives makes it difficult for a novice programmer to choose the best primitive from the communication library. A prescriptive tool can automatically analyze run-time data to specify the best primitive to use from among those available.

The first step is to analyze messages sent by the application, and potentially, their contents. The goal of the analysis is to determine opportunities for using less expensive collective communication primitives in place of the point-to-point primitives being used. This can be cast as a least-cost isomorphism problem on a communication graph, representing the communication structure of the application. Each communication primitive provided by the library can be viewed as a subgraph with

---

*Variable aliasing refers to the ability to access a variable without using its name. C permits practical use of variable aliasing by specifying the internal placement of elements of a data structure. This specification allows the different elements of a data structure to be accessed through one pointer.
an assigned cost. The problem is to find a least-cost set of subgraphs that covers the communication graph. Again, integer programming solvers can be used to obtain the optimal set of communication primitives.

The second step is to correlate the results of the analysis with the source program. Correlation requires pointing out the specific communication calls in the program that would need to be changed: information that can be easily obtained from the symbol table of the executable.

Finally, a framework would have to be provided to ensure that program correctness will not be violated by the substitution. An approach similar to the one we take for establishing the correctness of $R_X$ transformations (see Section 3.5) can be used for this purpose.

2.3.4 A Counter-Example

In the previous sections, we have specified the requirements for prescriptive performance tuning and also indicated three domains where it can be applied. This raises the question of whether there are situations where our approach may not be applicable. We answer this question in this section.

Consider a problem domain where some (or all) of our requirements cannot be met. For instance, implicit in our first requirement: “Automatically analyze run-time data to derive feedback”, is the condition that the derived feedback be useful. In other words, the feedback must improve performance. If it is not possible to automatically obtain the desired (useful) feedback for a problem domain, a prescriptive tool cannot be developed.

A specific instance of this case can easily be constructed. Consider a finite element simulation that is controlled by a number of parameters. A user who monitors the progress of the simulation, perhaps using a visualization tool, can “steer” the computation by changing its control parameters. Explicit control by a user enables many such complex simulations to be tractable. It is difficult to see how this process can be automated, i.e., function in the absence of user intervention. A prescriptive tool can be developed only if this process can be automated.

2.4 Related Work

Techniques for run-time performance tuning focus primarily on two aspects. A number of performance tools concentrate on the presentation of collected infor-
mation using differently formatted displays [LSV+89, DBKF90, RAN+93], animation [FLK+91, PU89], and auralization [FJ93]. These tools perform none or very little analysis of the data and leave the reasoning required to detect the performance problem and its solution entirely to the programmer. Other performance debuggers, such as the ones listed below, perform at least a partial analysis of run-time information in order to detect performance problems.

Quartz [AL90] uses the normalized processor time metric to rank the contribution of individual procedures in the program to the overall execution. This metric normalizes the execution time of a procedure to the concurrent parallelism present during its execution. A listing of the “importance” of different procedures to the execution is then produced. The programmer uses this listing a la gprof for performance tuning.

Paradyn [MCC+95] is a performance measurement tool that dynamically instruments the program being traced, and searches the execution for bottlenecks. Paradyn uses the time spent in various operations, such as synchronization or the time wasted blocking on message receives, as a metric. Trace data is collected from potential bottleneck sites. The dynamic instrumentation reduces the perturbation caused by the instrumentation. The tool builder creates a search model consisting of a set of hypotheses which are refined into more precise ones by testing for their validity during the execution. By starting from a broad set of hypotheses, the goal is to narrow the search down until one hypothesis (or a few) accurately reflects the performance bottleneck in the program. The problem with this approach is that performance bottlenecks cannot often be attributed to any one point in the code. For instance, the performance degradation might be caused by accesses to a particular data structure whose references are distributed through the program.

Mtool [GH93] analyzes memory and synchronization bottlenecks in parallel programs. Basic blocks and loops are instrumented to collect timing information. The difference between the actual time spent in a code section and that spent when the memory subsystem imposes no delays is reported as the memory overhead of that section. Mtool’s main drawback is that it provides only descriptive feedback. In addition, the level of detail at which information is provided does not easily explain the performance problems.

Cprof [LW94] is a cache profiling system for sequential programs that presents statistics in terms of both code and data structures. It categorizes cache misses obtained from a simulation of the program into compulsory, capacity or conflict misses and presents them the programmer. MemSpy [MGA95] also presents data-oriented
statistics for tuning the memory performance of parallel programs. At compile-time, instrumentation is added into the program which calls an event simulator at interesting events. The program is then simulated using Tango Lite [Go93], an address reference generator. MemSpy measures three types of cache misses viz. replacement misses (typically caused by cache interference), first-reference misses (typically caused by poor spatial locality) and invalidation misses (potentially caused by false sharing). It then correlates this information to code and data structures in the program. MemSpy leaves the programmer with the task of identifying the performance problems based on the statistics and displays it presents. For instance, the presence of a large number of invalidation misses to a data structure could indicate the presence of false sharing, or that the sharing is true and cannot be avoided. This deduction is left to the programmer.

ParaView [SB94] performs some analysis on trace information collected from a cycle-level simulation of the program and displays it in various formats to the programmer. Time spent in computation, synchronization and the memory hierarchy along with cache access information is presented. Miss rates are broken into categories as in MemSpy. Consistency misses that do not bring in information that is used subsequently are classified as false sharing misses. The user has to interpret this data.

The issue of correlating low-level information with the source program has been examined for the parallelizing compiler environment. For instance, Adve et al. describe an integrated compilation and performance analysis environment that provides source-level performance analysis of data-parallel programs [AMCA+95]. Their compiler provides information to the run-time system to enable it to correlate execution statistics with the source code. In contrast, our focus is on prescribing source-level transformations that address performance problems.

In contrast to the tools and methods outlined above, our goal is very different. Our focus is on automatically analyzing run-time information to prescribe solutions to performance problems.

Our approach also differs on a number of other points from those described above. Unlike Mtool, MemSpy and ParaView, we gather run-time information from an instrumented version of the program that executes in parallel on a parallel computer. Thus, compared to using a simulation, our data gathering phase takes significantly less time. Our approach also differs from tools like Quartz and Paradyn that use architecture dependent metrics (based on resource profiles). Instead, by using archi-
tecture independent information from the program execution, we are able to pinpoint sources of program inefficiency independent of the target architecture. Using architecture independent information also allows us to conduct our analysis on a system other than the one on which the performance debugging is taking place. A user can obtain program transformations that will improve performance, without actually carrying out the performance debugging on the target machine. Finally, to the best of our knowledge, no existing performance debugger looks at detecting or eliminating excess synchronization in shared-memory parallel programs.

2.5 Summary

In this chapter, we presented a new approach for developing performance tools. The basis of our approach is a set of requirements that must be satisfied by a performance analysis tool. In problem domains where these requirements can be met, a performance tool can prescribe source-level transformations to improve performance. In contrast, existing tools only provide descriptive feedback.

Three requirements must be satisfied to enable a tool to provide prescriptive feedback. These are (i) automatic analysis of run-time data to derive feedback, (ii) correlation of the feedback with the source program, and (iii) provision of a framework that allows the programmer to reason about the correctness of the feedback.

Next, we illustrated how these requirements can be satisfied in three very different domains to yield prescriptive tools. The advantage of these tools is that they can be used by novice programmers to rectify performance problems, while requiring only source-level, as opposed to architecture-related, reasoning about the program.

In the remainder of this dissertation, we demonstrate how we use this approach to develop $R_X$, a prescriptive tool that targets synchronization and some forms of data communication in explicitly parallel shared-memory programs. $R_X$ prescribes transformations to reduce these overheads.
Chapter 3

$R_X$: A Prescriptive Performance Debugger

3.1 Introduction

In the previous chapter, we described a new approach to performance tuning: prescriptive performance debugging. The basis of this approach is a set of requirements that must be satisfied by a performance analysis tool. Our approach can be used to design prescriptive tools for a wide variety of purposes.

$R_X$ is one such tool that we have developed to improve the performance of explicitly parallel shared-memory programs. $R_X$ targets inter-process synchronization and data communication, two significant sources of overhead in shared-memory applications, prescribing transformations to reduce these overheads.

Figure 3.1 explains the different steps involved in using $R_X$. First, the explicitly parallel source program is compiled using a standard compiler. Next, we analyze the assembly output from the compiler and instrument it to collect dependence information at run-time. The executable is then run on a shared-memory parallel computer. The instrumentation we insert produces trace data, which is written to log files. $R_X$ then automatically analyzes this data, and prescribes transformations that reduce synchronization and communication.

The responsibility for applying the transformations resides with the user. Specifically, $R_X$ does not itself automatically apply the transformations. $R_X$ does not do this because programmer intervention may be required in order to ensure that the transformations, obtained by analyzing run-time data, are applicable to all executions.

This chapter mainly serves as an introduction to $R_X$. We describe the motivation behind $R_X$ in Section 3.2, explaining how excess synchronization can arise in shared-memory programs and why removing it can improve performance. In Section 3.3, we discuss our rationale for choosing a run-time performance debugging approach. However, a run-time approach suffers from some drawbacks. Specifically, the transformations we determine based on one execution may violate program correctness in
another execution. We explain this drawback in Section 3.4. We describe how $R_X$ handles this problem in Section 3.5. Our approach is to use metaknowledge from the programmer and the run–time system, enabling $R_X$ to make guarantees about the correctness of the transformations it prescribes. In Section 3.6, we use Conjugate Gradient, a shared–memory application, to illustrate the nature of the transformations prescribed by $R_X$. We summarize this chapter in Section 3.7.

![Diagram](image)

**Figure 3.1** A high–level overview of the $R_X$ performance debugging approach.
3.2 The Motivation Behind $R_X$

In the shared memory programming paradigm, there is a global address space that can be accessed from anywhere in the system. Different processors can access locations anywhere within this global address space. Processors communicate by reading and writing locations in the shared memory.

Since the same location can potentially be accessed simultaneously by multiple processors, a mechanism to order accesses is required. Synchronization is used for this purpose. Specifically, conflicting accesses (read–write or write–write accesses to the same memory location) [SS88] need to be ordered using synchronization. If this ordering is not enforced, time-dependent bugs known as race conditions [Net91] may be introduced into the program. Consequently, the program behavior can be unexpected or even incorrect. By using synchronization to order accesses, the program can be made to behave as expected.

Faced with the possibility of race conditions, most inexperienced parallel programmers add more synchronization than necessary to their applications. By over-synchronizing, the programmer can often ensure that the application is free of data races without an in-depth understanding of its sharing pattern. This is especially true when the programmer is not writing the application from scratch, but is instead parallelizing or porting it. Erring on the side of caution results in programs containing excess or unnecessary synchronization.

Figure 3.2 illustrates four different situations with unnecessary synchronization. The examples are simplified versions of excess synchronization that we have encountered in various shared-memory applications.

<table>
<thead>
<tr>
<th>Example A</th>
<th>Executed on Pa</th>
<th>Executed on Pb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barrier B0</td>
<td>Barrier</td>
<td>Barrier</td>
</tr>
<tr>
<td>Write X</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Barrier B1</td>
<td>Lock L0</td>
<td>Lock L0</td>
</tr>
<tr>
<td>Read Y</td>
<td>$x = x + 1$</td>
<td>$y = y + 6$</td>
</tr>
<tr>
<td>Barrier B2</td>
<td>Unlock L0</td>
<td>Unlock L0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example B</th>
<th>Executed on Pa</th>
<th>Executed on Pb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barrier</td>
<td>Barrier</td>
<td>Barrier</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example C</th>
<th>Executed on Pa</th>
<th>Executed on Pb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barrier</td>
<td>Barrier</td>
<td>Barrier</td>
</tr>
<tr>
<td>Lock L0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$x = x + 4$</td>
<td>$y = y + 6$</td>
<td></td>
</tr>
<tr>
<td>Unlock L0</td>
<td>Unlock L0</td>
<td>Unlock L0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Example D</th>
<th>Executed on Pa</th>
<th>Executed on Pb</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lock L0</td>
<td>if (condition)</td>
<td></td>
</tr>
<tr>
<td>Unlock L0</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 3.2 Excess synchronization examples.
Example A: Here, barrier \textit{B1} serves no purpose as none of the accesses before and after it are to the same set of locations. The barrier can therefore be removed.

Example B: Here, the critical section made up of the lock and unlock to \textit{L0} enforces exclusive access to \textit{x}. However, as only process 0 can access \textit{x}, the enclosed computation can be moved outside and the critical section removed.

Example C: Here, the two program snippets shown are executed by processes \( P_a \) and \( P_b \). Again, the critical sections controlled by \textit{L0} ensure exclusive access to data by the enclosed computation. However, since \( P_a \) and \( P_b \) are accessing different data structures inside the critical sections, mutual exclusion is not required. The lock and unlock can therefore be removed.

Example D: Here, unless the accesses made in evaluating the condition need to be ordered with respect to the other accesses, the critical section can be nested inside the conditional. If this can be done, the synchronization overhead is incurred only in those cases where the conditional evaluates to \textit{true}.

Excess synchronization often degrades performance because synchronization imposes several forms of overheads. First, by requiring processes to interact, it imposes a communication overhead. This can be significant in loosely coupled systems. Second, synchronization can reduce parallelism by causing a process to block, forcing it to wait for another process. Third, executing the synchronization protocol imposes an overhead, which can be large in programs that synchronize frequently. Finally, synchronization inhibits compiler optimizations that reorder computation. Thus, by removing excess synchronization, the performance of a program may be improved considerably.

Data communication can also be a significant source of overhead in shared-memory programs. Ideally, computation should be performed by the process that has computed most of the source operands, and uses or reads most of the results. Provided load balance is not degraded, it may be possible to improve program performance by restructuring computation.

We have detected opportunities to reduce synchronization and data communication in applications drawn from a wide variety of sources. By applying \textit{Rx}, a prescriptive tool that targets these overheads, we have been able to improve program performance, in some cases by several orders of magnitude. From a programmer's
perspective. A significant advantage of $R_X$ over other tools is that it prescribes source-level transformations.

### 3.3 Why Use a Run-Time Approach?

At the heart of the methods to detect excess synchronization and data communication is the detection of conflicting accesses (read-write or write-write accesses to the same location) [SS88]. A prerequisite for detecting conflicting accesses is the accurate determination of *access sets*, i.e., the set of data read or written in different parts of the program. Hence, the more precisely we can determine the access sets, the greater the effectiveness of our techniques for reducing excess synchronization and data communication.

Compile-time methods for determining access sets are not always precise. In some cases, typically programs with regular accesses, static analysis can precisely determine the access sets. However, in the presence of unknown loop bounds, procedure calls, indirection arrays, and pointers, only a conservative approximation (i.e., a superset) of the true access sets can be determined. Lacking the precise access sets, transformations obtained from compile-time analyses are often conservative.

On the other hand, by working with precise access sets gathered from a program execution, we can expose more opportunities for reducing synchronization. The access sets for a particular execution can be precisely determined, just by keeping track of the memory locations accessed by the program. This allows us to successfully performance tune pointer-ridden applications with complex data structures; applications for which the access sets cannot often be determined precisely at compile-time.

### 3.4 The Drawback of a Run-Time Approach

In the previous section, we explain the rationale for implementing our methods in a performance debugger. Nevertheless, a run-time approach also has drawbacks. Consider a case where $R_X$ determines that a particular synchronization is required for an execution. This implies that there is at least one execution of the program where this synchronization is required. However, the implication is only one-way. In other words, by observing that a synchronization operation is not required for an execution, the debugger *cannot* conclude that the synchronization will not be required for any execution.
Barrier B0
ptra[pid] = ...
Barrier B1
for (i = 0; i < nprocs; i++) {
    ... = ptrb[i]
}
Barrier B2

Figure 3.3 Example illustrating the one-way implication. pid refers to the process identifier, and nprocs the number of processes.

This is illustrated by the example in Figure 3.3. If the accesses before and after barrier B1 do not conflict, the barrier can be removed. Let R_X observe this to be the case in one execution. We cannot therefore conclude that this will always be the case. That depends on whether the nprocs locations starting from ptra will ever overlap with the nprocs locations starting from ptrb.

From a programmer's perspective, R_X is most useful if, in addition to providing a transformation, it also tells the programmer whether the transformation will be valid for all executions. The programmer can then rest easy. However, a performance debugger can observe only a small number (possibly just one) of executions. Its findings are hence based on, and restricted to, these executions. To extend the scope of these findings, the debugger analysis must be based on the set of all possible executions. The static analysis used in compilers accomplish precisely this.

3.5 Extending the Scope of Transformations

The discussion in the previous sections leaves us with a dilemma. While we use a run-time approach to determine the access sets precisely, analysis carried out with run-time data can be correctly applied only to the execution(s) from which the data was obtained. To extend the scope of the findings requires analyzing the set of all executions. Static analysis can do this, but the access sets it arrives at can be imprecise.

Our solution is to rely on the programmer to provide us with additional information. When determining the transformations that reduce synchronization and
communication, we also determine the conditions under which these transformations will be applicable to all executions. If we could prove or disprove these conditions, we would have achieved our goal. Since the limitations of static analysis may prevent us from doing this, we make use of metaknowledge about the run-time system and the program.

We proceed in two directions. First, we provide $R_X$ with invariants due to the run-time system. For example, these “system-wide” invariants could specify that the process identifier is different on each process. The invariant could perhaps specify that all shared data is allocated from the heap. We also allow the programmer to specify properties of the program to the debugger. These would be in the form of control-flow invariants and descriptions of the data access patterns. Using this metaknowledge, $R_X$ attempts to infer the correctness of the transformation. Currently, we only permit the metaknowledge to be provided in terms of a propositional calculus. Hence, the debugger actions to carry out the inference are straightforward.

Supplied Invariant: pid is unique to each process

```
Barrier
if (pid == 0) {
  Lock L1
  *ptra = ...
  Unlock L1
}
Barrier
```

Figure 3.4 Using system metaknowledge.
pid refers to the process identifier.

We illustrate this with the example in Figure 3.4. Here, we have supplied the debugger with the invariant that the process identifier, pid, is unique to each process. Based on this, $R_X$ can determine that only one process can execute the critical section protected by $L1$. As a result, it can determine that the access to ptra need not ever be protected by the critical section.

This form of programmer involvement is not new. For instance, the SGI Power compiler provides pragmas based on which the compiler parallelizes loops [Bau92].
Sun Microsystem's lock lint tool [Sun95] uses programmer annotations extensively to check for data races and improperly used locks in multithreaded programs. Static debugging (see Bourdoncle's work [Bou93] for example) also has a history of using programmer supplied invariants.

When the supplied information is insufficient for the debugger to infer the correctness of the transformations, we proceed in the second direction. The debugger presents the programmer with the transformations, and the conditions under which they would be universally applicable. The debugger then makes a contract with the programmer. If the programmer can prove that the conditions supplied by the debugger will always hold, then the debugger guarantees that the transformations will be correct. Otherwise, the transformations should not be applied.

```
Barrier B0
ptra[pid] = ...
Barrier B1
for (i = 0; i < nprocs; i++) {
    ... = ptrb[i]
}
Barrier B2
```

\( R_X \) contract: If the accesses through ptra and ptrb do not conflict, then, the barrier \( B1 \) can be safely removed.

**Figure 3.5** Illustrating the contract between \( R_X \) and the programmer.

We illustrate this situation with the example in Figure 3.5. The program fragment shown is the same as in Figure 3.3. As before, let \( R_X \) have observed that barrier \( B1 \) is not required for an execution. Since there are no invariants supplied that will let \( R_X \) infer that \( B1 \) is not required for all executions, we fall back on the programmer. \( R_X \) states that if the programmer can guarantee that the accesses made through ptra and ptrb will not conflict, then the barrier \( B1 \) can be safely removed.

Interestingly, the amount of programmer involvement we require is a lot less than would be required by a tool that uses static analysis to prove properties about a program. The difference stems from the fact that we execute the program to prune away the uninteresting cases; cases where the synchronization is found to be necessary.
Thus, we need more information about the program only at the places where we find synchronization and communication to be excessive. In essence, the feedback we provide greatly reduces the state space of transformations the programmer has to reason about in order to improve performance.

### 3.6 $R_X$ Prescriptions: An Example

So far, we have only described the kind of feedback that $R_X$ prescribes. In this section, we provide a concrete example illustrating the nature of this feedback. We applied $R_X$ to Conjugate Gradient, a program written by first-year graduate students as part of a course programming project. Figure 3.6 illustrates three transformations prescribed by $R_X$ for this program. A description of this application and a detailed explanation of the $R_X$ transformations are presented much later in Section 7.3.2.

<table>
<thead>
<tr>
<th>cg.c: Critical section starting at line 352 may be removed using vectorization.</th>
</tr>
</thead>
<tbody>
<tr>
<td>This transformation is valid for all executions provided:</td>
</tr>
<tr>
<td>1. Reads to *x and *y in line 349 do not conflict with Write to *shared.ddotr in line 353</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cg.c: Computation in line 386 can be placed outside enveloping critical section.</th>
</tr>
</thead>
<tbody>
<tr>
<td>This transformation is valid for all executions provided:</td>
</tr>
<tr>
<td>1. Critical section is preceded in all executions by the barrier at line 356</td>
</tr>
<tr>
<td>2. Read to *x in line 386 does not conflict with Write to *shared.q in line 386</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cg.c: Critical section synchronization starting at line 386 can be removed.</th>
</tr>
</thead>
<tbody>
<tr>
<td>This transformation is valid for all executions provided:</td>
</tr>
<tr>
<td>1. The computation in line 386 can be moved outside</td>
</tr>
</tbody>
</table>

**Figure 3.6** Three $R_X$ prescriptions for Conjugate Gradient.

An informal discussion of these transformations is as follows. The first prescription specifies *vectorization*, a transformation that reduces the number of critical sections.
The second and third transformations are linked. The second transformation specifies that computation currently taking place inside a critical section can be moved outside. The third transformation states that if the computation inside the critical section is moved outside, the critical section can be removed. After each transformation, $R_X$ specifies the conditions under which the transformation can be applied to the program. The set of conditions for applying each transformation make up the $R_X$ contract for that transformation. Note that the $R_X$ prescriptions require only a source-level reasoning of the program.

At present, the prescriptions are in plain text form. Our plan is to provide an auxiliary user interface tool with which the user can interact while interpreting the feedback. The purpose of this tool will be to simplify the task of the user when checking the conditions making up the $R_X$ contract. For instance, once the user verifies that the $R_X$ stipulations for the second prescription in Figure 3.6 are valid, the auxiliary tool can determine that the condition for the third prescription has been satisfied, obviating the need for further user intervention.

### 3.7 Summary

We introduced $R_X$ in this chapter. Unlike existing performance debuggers that provide only descriptive feedback, $R_X$ prescribes a set of transformations for the program that will improve its performance.

We began by explaining the motivation behind detecting excess synchronization and data communication. We then explained our rationale for using a run-time approach in $R_X$. Although static analysis can precisely determine the access sets for regular programs, it is less effective in the presence of unknown loop bounds, procedure calls, indirection arrays, and pointers. In such cases, transformations based on static analysis must be conservative in order to be correct. By implementing our methods in a run-time tool, we have been able to successfully performance tune pointer ridden applications with complex data structures. A correctness framework ensures that $R_X$ prescriptions obtained from one or a small set of executions, will be applicable to all executions.

In the next three chapters, we describe the details of $R_X$. We begin with the algorithms in Chapter 4 for detecting excess synchronization and reducing data communication. The automatic data analysis in $R_X$ is based on these algorithms.
Chapter 4

Reducing Synchronization and Communication

4.1 Introduction

We introduced $R_X$ in the previous chapter. In order to prescribe transformations, $R_X$ must automatically analyze run-time data. In this chapter, we describe a set of algorithms for detecting excess synchronization and reducing some forms of data communication. These algorithms form the basis for the automatic data analysis in $R_X$, permitting it to prescribe transformations that improve performance.

Before developing these algorithms, we examined a large number of shared-memory applications, including several standard benchmarks.* We were able to detect several instances of excess synchronization in these applications, as also possibilities for reducing synchronization and data communication. Our observations motivated the algorithms presented in this chapter. These algorithms may not uncover all possible cases of excess synchronization and data communication. However, the methods we outline in this chapter can be extended to handle other cases that may arise.

Excess synchronization can arise due to several reasons. Programmers often oversynchronize, forcing more processes to interact than needed or forcing the processes to interact more frequently than needed. Another common error is to use a synchronization construct that imposes more ordering constraints than required. A synchronization operation may also be redundant, either because all the dependences it enforces are satisfied by other synchronization operations, or because no dependences need to be enforced.

There are two main forms of excess synchronization. In the first case, a synchronization operation is strictly unnecessary and removing it does not alter the correctness of the program. Along the same lines, a synchronization operation may impose more ordering than necessary. So replacing it with a weaker form of synchronization is possible. In the second case, the excess synchronization is a pattern of

*A subset of these applications is showcased in Chapter 7.
synchronization operations that can be eliminated by transforming the program. In contrast to the first case, the individual synchronization operations are essential to the correctness of the program as written. However, a modest change to the program, particularly the data structures, may allow some synchronization to be eliminated.

Synchronization and data communication in a program may also be reduced by restructuring the computation. Specifically, by moving computation around synchronization points (when it does not affect correctness), we may be able to weaken or eliminate other some synchronization. The data locality can also be improved by restructuring the computation, resulting in reduced data communication.

The chapter organization is as follows. In Section 4.2, we describe the conditions required by the algorithms in this chapter to work correctly. We describe the partial-order graph, a construct used in the remainder of this dissertation, in Section 4.3. In Section 4.4, we describe the techniques for detecting unnecessary (or unnecessarily strong) synchronization. Next, in Section 4.5, we discuss how to aggregate and vectorize critical sections, two transformations that reduce synchronization without reducing parallelism. We also describe the conditions under which these transformations can be applied without affecting program correctness. In Section 4.6, we discuss the conditions under which synchronization and data communication can be reduced by postponing and relocating computation, and also when these restructuring can be applied without affecting program correctness. We discuss related work in Section 4.7, and summarize the chapter in Section 4.8.

4.2 Assumptions

The algorithms described in this chapter have two requirements. If these requirements are not satisfied, incorrect transformations may result.

- All synchronization accesses must be distinguished (i.e., made visible) to the algorithms.
  Since most programs make use of system-supplied libraries for synchronization, this requirement is met by most programs without taking any special steps.
- The program must be free of data races.
  In the absence of synchronization between conflicting accesses, a program is said to exhibit a data race [PK96]. Data races may cause the behavior of a program to be time-dependent. Their presence may therefore cause programs to execute
incorrectly. Therefore, it is not unreasonable to require that the program being analyzed be free of data races.

The programming model assumed by our algorithms is sequential consistency [Lam79]. Since we require programs to be free of data races, our algorithms can also be applied to programs written under weaker consistency models such as release consistency [GLL+90], lazy release consistency [K CZ92], and data-race-free-0 [AH90]. In the absence of data races, these weaker models guarantee sequential consistency.

It must be noted that the algorithms described in this chapter only reduce synchronization and communication. In particular, they do not guarantee that the ensuing synchronization will be minimal.

4.3 The Partial-Order Graph

During the execution of a program, the program order and the inter-process synchronization impose a partial ordering on all accesses. This partial ordering is used to create a partial-order graph, made of as many disjoint subgraphs as there are processes. The nodes of a subgraph represent the synchronization operations issued by a process. The program order imposes a total ordering on all nodes in a subgraph. Between subgraphs, synchronization operations are used to create directed edges. An edge is drawn from the release of a synchronization variable to its subsequent acquire, as observed during the execution. A release is a synchronization write and an acquire, a synchronization read [GLL+90]. The synchronization operations divide the execution on a process into intervals. Let $S_0$ and $S_1$ denote two synchronization operations on one process. Then we denote the interval between them by $(S_0, S_1)$.

This graph enables us to evaluate the happens-before-I relation ($\rightarrow_{hb}$) [AH93]. Given accesses $x$ and $y$, $x \rightarrow_{hb} y$ iff there is a path in the partial-order graph from the release immediately succeeding $x$ to the acquire immediately preceding $y$. This relation allows us to determine the earliest interval in which a process can become aware of a write on another process.

4.4 Detecting Unnecessary Synchronization

Synchronization is needed only to enforce interprocess data dependences. In other words, it is required only between conflicting data accesses (read-write or write-
write accesses to the same memory location) [SS88]. In this section, we discuss how to determine whether a particular synchronization operation is required or not.

![Diagram of partial-order graph for processes Pa and Pb.]

**Figure 4.1** A portion of the partial-order graph for processes Pa and Pb.

Consider Figure 4.1 which shows a portion of the partial-order graph for an execution on two processes, Pa and Pb. Let R_{a}^{before} and W_{a}^{before} refer to the read and write sets on Pa, preceding the current synchronization and following a previous synchronization between the two processes. Let R_{b}^{after} and W_{b}^{after} refer to the read and write sets on Pb, following the current synchronization and preceding a subsequent synchronization between the two processes. A read or write set of a portion of the program execution is the set of reads or writes made during that portion. Then, the acquire and release making up the synchronization in the middle can be removed if:

\[
W_{a}^{before} \cap R_{b}^{after} = R_{a}^{before} \cap W_{b}^{after} = W_{a}^{before} \cap W_{b}^{after} = \emptyset
\]  

Equivalently, synchronization is necessary only if there exists an interprocess flow, anti- or output data dependence. These are an adaptation of Bernstein's conditions for determining whether the concurrent execution of a set of tasks is determinate [Ber66].

### 4.4.1 Barriers

The beginning and end of the program execution, and the barriers in between, divide a program into what we call b-intervals. If all the read and write accesses made in two adjacent b-intervals are known, we can determine whether the intervening barrier is necessary, or if we can replace it with weaker synchronization. Consider the barrier between two adjacent b-intervals. Let R_{i}^{after} denote the read set on process Pi in the b-interval following the barrier, and W_{i}^{before}, the write set on process Pi in the
<table>
<thead>
<tr>
<th></th>
<th>Condition</th>
<th>Synchronization</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$R_{i}^{after} \cap \bigcup_{i,j} W_{j}^{before} \neq \emptyset$, $\forall i,j \in P$</td>
<td>Barrier needed</td>
</tr>
<tr>
<td>2</td>
<td>$R_{i}^{after} \cap \bigcup_{i,j} W_{j}^{before} \ll \bigcup_{i,j} W_{j}^{before}$, $\forall i,j \in P$</td>
<td>Barrier probably underutilized</td>
</tr>
<tr>
<td>3</td>
<td>$R_{i}^{after} \cap \bigcup_{i,j} W_{j}^{before} \neq \emptyset$, $\forall i,j \in PS$, $\equiv \emptyset$, $\forall i,j \in P - PS$</td>
<td>Partial barrier (over $PS$)</td>
</tr>
<tr>
<td>4</td>
<td>$R_{i}^{after} \cap \bigcup_{i,j} W_{j}^{before} \neq \emptyset$, specific $i$, $\forall j$, $\equiv \emptyset$, all other $i$, $\forall j$, $i,j \in P$</td>
<td>Pairwise Sync</td>
</tr>
<tr>
<td>5</td>
<td>$R_{i}^{after} \cap W_{j}^{before} \neq \emptyset$, $\forall i$, specific $j$, $\equiv \emptyset$, $\forall i$, all other $j$, $i,j \in P$</td>
<td>Pairwise Sync</td>
</tr>
<tr>
<td>6</td>
<td>$R_{i}^{after} \cap \bigcup_{i,j} W_{j}^{before} \equiv \emptyset$, $W_{i}^{after} \cap \bigcup_{i,j} W_{j}^{before} \neq \emptyset$, and/or $\forall i,j \in P$</td>
<td>Control barrier</td>
</tr>
</tbody>
</table>

**Table 4.1** Conditions under which a barrier can be replaced by other synchronization ($j \neq i$ for all cases). $P$ refers to the set of processes, and $PS$ to a subset.

preceding b-interval. $R_{i}^{before}$ and $W_{i}^{after}$ are defined analogously. The appropriateness of the barrier for the purpose of synchronization is given by the following equations:

$$\forall i,j \in \{\text{processes}\}, \ j \neq i \quad R_{i}^{after} \cap \bigcup_{i,j} W_{j}^{before}$$  \hspace{1cm} (4.2)

$$W_{i}^{after} \cap \bigcup_{i,j} R_{j}^{before}$$  \hspace{1cm} (4.3)

$$W_{i}^{after} \cap \bigcup_{i,j} W_{j}^{before}$$  \hspace{1cm} (4.4)

If all three equations evaluate to the null set for all processes, there are no dependencies between the accesses before and after the barrier. The barrier is not required and can therefore be removed outright. The barrier can also be removed or replaced with cheaper synchronization under a variety of conditions. For instance, when the equations evaluate to the null set for only some processes, the barrier can be replaced with weaker synchronization. Even if no equations evaluate to the null set, it may still be possible to remove the barrier or weaken it, by modifying the program. Table 4.1 is a fairly exhaustive list of the several cases that can arise.
Two forms of program modifications may permit the removal of a barrier or its weakening even when not all the equations evaluate to the null set. When only equation (4.2) is equal to the null set for all processes, the odd–even solution may permit the barrier to be removed. Computation postponement, which we describe in Section 4.6.1, may permit the barrier to be removed or weakened.

The odd–even solution works by removing the anti- and output dependences that necessitate the barrier in the first place. This is done by providing multiple locations for the data involved in the dependence. By using a different location for the final write than the ones being used for the earlier read or write, the anti- or output dependence can be eliminated.\footnote{One replica suffices for every datum involved in the dependence, if it occurs in a loop that already has a barrier. In the absence of this barrier, the space overhead depends on the amount of data involved in the dependence and the number of iterations of the loop.} Consequently, the barrier can be removed.

When only a small fraction of the data produced on the other processes is read after the barrier, the barrier is probably overkill (case 2 in Table 4.1). It may be possible to weaken or remove the barrier by restructuring the computation. Water and Barnes–Hut (see Section 7.3) are two applications where this situation arises. Here, the computation that uses data produced before a barrier can be postponed to the following b–interval. By deferring this computation, the first barrier can be removed. A full treatment of restructuring computation to reduce synchronization is deferred until Section 4.6.

Qualitatively, a barrier is most utilized when in the following b–interval, a process reads data written in the preceding b–interval on all the other processes. When this is not true, the barrier can be weakened by replacing it with cheaper synchronization. Thus, when only a subset of processes share data between themselves (Table 4.1. case 3), synchronizing all the processes with a barrier is a waste. A partial barrier, which synchronizes the process subset alone, suffices.

A barrier can also be weakened by replacing it with pairwise synchronization such as flags (Table 4.1, cases 4 and 5). For instance, consider a manager process operating on data produced by a set of workers. When only the dependences between the data produced by the workers and that used by the manager need be enforced, pairwise synchronization suffices. Flags can also be used to enforce the true dependence between the production of data by a producer process and its consumption by consumer processes. A barrier clearly constitutes excess synchronization in this case.
Many programs have phases where a manager process uses the results from one phase of the computation to repartition the work among the worker processes. The workers then use this new partition in the next phase of the computation. This also corresponds to cases 4 and 5 in Table 4.1. Using pairwise synchronization as explained above provides the same benefits as using a fuzzy barrier [Gup89]. In a fuzzy barrier, the barrier releases and acquires are separated, permitting computation to be carried out between the two.

Finally, when a barrier is present only to enforce anti-dependences, it can be replaced with a weaker form of synchronization that does not make the shared address space consistent. Here, the barrier only serves as a notification to a later writer that the reads preceding the barrier (by program order) have been performed. In particular, the barrier is not being used to communicate values written preceding it (by program order). Hence, the barrier can be replaced by a special synchronization construct that ensures that all reads preceding it (by program order) will be performed before the synchronization access will be performed. We call this construct a control barrier.

Software DSM systems such as TreadMarks [KDCZ94], which make the address space consistent lazily, can take advantage of control barriers. In such systems, by using a control barrier, substantial benefits can be obtained by eliminating the false sharing [TLH94] that may be caused by making the address space consistent eagerly.

### 4.4.2 Pairwise Synchronization

Acquires and releases constitute the general form of synchronization that can be used to impose an access ordering. In this section, we examine how to eliminate excess synchronization edges due to pairwise synchronization (such as flags), while preserving the ordering among conflicting accesses.

As described in Section 4.4, we can use the presence of conflicting accesses to infer that a particular pairwise synchronization construct is required. However, a synchronization edge can enforce more than one dependence. Therefore, by looking at all the dependences together, we can reduce synchronization better than we can by looking at each dependence in isolation. We illustrate this with an example.

Consider the code fragment shown in Figure 4.2 along with its partial-order graph. Here, the only conflicting accesses are by \(P_a\) and \(P_d\) to \(t\), and by \(P_b\) and \(P_c\) to \(y\). These constitute the only dependences that need to be synchronized. We could consider the
Figure 4.2  A code fragment and its partial-order graph. An edge denoted by (⃝) represents the release and subsequent acquire of the synchronization variable c. The bold boxes indicate conflicting accesses.

two dependences independently, and determine the synchronization that will enforce each. A possible choice of synchronization edges would then be:

- a, b, and e for the accesses to t,
- d for the accesses to y.

Note that no edge can be removed from this set with the remainder still being capable of enforcing the dependences. In other words, no edge in the set is redundant. Yet, we do not need four synchronization edges to enforce these two dependences. A better choice would be the three edges a, c, and e. By choosing these edges, we can eliminate all excess synchronization in the example.
Finding the Minimal Synchronization from Among Those Present

Consider the dependences in Figure 4.2 along with the synchronization that can enforce them. We have:

- \( t \): Edges \( a, b, \) and \( e \) or Edges \( a, c, \) and \( e \)
- \( y \): Edge \( c \) or Edge \( d \)

We can represent this as:

\[
\begin{align*}
\text{Dep } t \xrightarrow{\text{enf by}} & (a \ b \ e) + (a \ c \ e) \\
\text{Dep } y \xrightarrow{\text{enf by}} & c + d
\end{align*}
\]

(4.5)

where \( \xrightarrow{\text{enf by}} \) indicates "enforced by".

Each dependence can be represented in this form, with the right hand side representing the set of possible synchronization edges that can enforce the dependence. This can be interpreted in a purely boolean algebraic sense, where every variable on the right hand side takes on values from the set \( \{0,1\} \), with the requirement that each formula representing a dependence evaluate to \text{true}.

As all dependences need to be enforced, we can consider these equations collectively. The minimal synchronization set (from those already in the program) can then be determined as the minimal set of edges required to enforce all dependences.

Thus, our first step is to find the paths between the source and sink of each dependence. From these, we directly obtain equations similar to equation (4.5). Once we have these equations, we can consider them collectively. Let \( \delta_i \) represent the \( i \)th dependence that needs to be enforced. We then have a single equation with a conjunction of the dependences on the left side, and a set of \textit{terms} on the right side.

\[
\prod_{i=1}^{\text{total deps}} \delta_i \xrightarrow{\text{enf by}} \prod_{i=1}^{\text{total terms}} t_i
\]

Each term, \( t_i \), represents the enforcement of dependence \( \delta_i \) by the appropriate synchronization.

For simple cases, we expand the right hand side of the equation into its disjunctive normal form (DNF). Although the expansion can take time exponential in the size of the right hand side, for small problem instances, this brute-force approach suffices. Then, we obtain the minimal set of synchronization to enforce \textit{all} the dependences as the term in the DNF expansion with the smallest number of synchronization edges.
When the equation has a large number of terms, we resort to integer programming. A description of the integer programming formulation can be found in our earlier work [RC97a].

For the example in Figure 4.2, we obtain the following equation:

\[
\prod_{i=1}^{\text{total}} \delta_i \xrightarrow{\text{cnf by}} (abc) + (abde) + (ace) + (acde)
\]

The minimal set of synchronization is \(a\), \(c\), and \(e\). Note that this is only the minimal synchronization from among those already present in the program.

### 4.4.3 Critical Sections

Although locks and unlocks are just acquire and release operations, programmers typically use them in pairs to implement mutual exclusion. When used in this restrictive fashion, we can treat them differently from the pairwise synchronization described in Section 4.4.2. Here, we describe how we can detect operations that do not have to be performed inside critical sections. By doing so, we can reduce the time for which a process "holds" a critical section. Figure 4.3 gives an example, in which the print statements can be moved outside the critical section, resulting in the program on the right-hand side. As the black bars indicate, the processes will hold the critical section for lesser time in the transformed program.

```
repeat
{ local = foo ()
  Lock L0
  printf ("Whats up Doc?")
  printf ("local = %d", local)
  global = global + local
  Unlock L0
} until (computation over)

repeat
{ local = foo ()
  Lock L0
  printf ("Whats up Doc?")
  printf ("local = %d", local)
  global = global + local
  Unlock L0
} until (computation over)
```

**Figure 4.3** Reducing time spent inside a critical section.
We define a critical section (recursively) as follows. A lock and unlock operation that access the same location and are executed by the same process constitute a critical section provided:

- No other program memory operations access the locations accessed by the lock and unlock.
- The lock operation is always ordered before the unlock.
- The only other synchronization operations between the lock and unlock in any execution are other locks and unlocks that also make up critical sections. Thus, while nested critical sections are okay, other synchronization (such as flags or barriers) are not permitted inside critical sections.

For convenience, we refer to a location accessed by the lock and unlock operations as a *synchronization variable*. We also use the term "same synchronization variable" to imply that the locations accessed by two lock or unlock operations are the same.

The following discussion makes use of the terms "critical path", "sender", and "receiver". These terms have been defined by Adve [Adv96]. The definitions are reproduced in Appendix A. However, for the class of programs that we consider (see Section 4.2), these terms can be explained as follows.

**Figure 4.4** Illustrating terminology. An edge denoted by $a$ represents the release and subsequent acquire of the synchronization variable $a$. $R_x$ and $W_x$ denote a read and write to memory location $x$.

Consider the partial-order graph for an execution. An *ordering path* is a path in this graph between two conflicting accesses. In Figure 4.4, $R_x \rightarrow a \rightarrow W_x$ is an
ordering path. Now consider two conflicting accesses to memory location $x$, such that there is no ordering path between the two which includes a write to $x$. These two operations are called consecutive conflicting operations. In Figure 4.1, the read to $x$ and the write to $x$ on $P_k$ are consecutive conflicting operations. A counter-example is the read to $x$ and the write to $x$ on $P_c$. The ordering paths between two consecutive conflicting operations are called candidate critical paths. Thus, $R_x \rightarrow a \rightarrow W_x$ and $R_x \rightarrow b \rightarrow W_x$ are candidate critical paths. A critical set of paths is now constructed as follows. For each pair of consecutive conflicting operations, one ordering path between the two (i.e., one candidate critical path) is placed in the critical set. Each path in the critical set is termed a critical path. In the figure, there are two critical sets of paths. One set is \( \{ R_x \rightarrow a \rightarrow W_x, \ W_x \rightarrow c \rightarrow W_x \} \). The only other set for this execution is \( \{ R_x \rightarrow b \rightarrow W_x, \ W_x \rightarrow c \rightarrow W_x \} \). Releases on cross-process synchronization edges that are part of a candidate critical path are called the senders in the path. The corresponding acquires are termed receivers. Together they are known as communicators.

The restrictions on when accesses can be moved with respect to the lock and unlock of the critical section can be stated as follows:

1. Take all the critical paths that originate with accesses made inside the critical section, and for which the unlock is the sender. These accesses must precede the unlock. Refer to these accesses as $A$.

2. Take all the critical paths that terminate with accesses made inside the critical section, and for which the lock is the receiver. These accesses must follow the lock. Refer to these accesses as $B$.

Informally, these conditions check for an access inside the critical section that conflicts with another access (by a different process), and for which the lock or unlock lies on the synchronization edge that orders the conflicting accesses. If the lock lies on the synchronization edge, the access must follow the lock. If the unlock lies on the synchronization edge, the access must precede the unlock.

Only the accesses in \((A \cap B)\) need to be inside the critical section. All other accesses can be placed outside the critical section, provided that all uniprocess control and data dependences are obeyed.

This transformation allows us to detect several common cases of programmer over-synchronization, two of which are shown as examples C and D in Figure 3.2.
4.5 Transforming Synchronization

Often, it is possible to apply transformations to the program that eliminate some inter-process dependences. This can be done by observing the global access and synchronization pattern across all processes. By changing the accesses and computation, these false dependences can be removed, eliminating the conflicting accesses and the need for synchronization. Figure 4.5 shows how a “reduction” operation across the processes can be transformed to use less synchronization. Here, the lock and unlock operations are only needed to enforce dependences between the accesses to `glob` by the different processes. Using aggregation and vectorization, two transformations we describe in this section, the critical sections can be eliminated.

A critical section is typically used in two situations. As in Figure 4.6(a), it may be used to ensure that updates to a data structure are carried out atomically. In this case, the synchronization is used only to ensure atomicity. These are the non-synchronizing `atomic` (ns-atomic) operators described by Adve et. al. [ACDZ94].

The lock and unlock operation of a critical section may also be used to impose an ordering on accesses made outside the critical section. Consider a task queue where processes execute the following actions in a loop (see Figure 4.6(b)). A process dequeues a task from the queue and executes it. While executing the task, the process can potentially produce data for more tasks and enqueue the newly generated tasks.

<table>
<thead>
<tr>
<th>Original</th>
<th>Aggregated</th>
<th>Aggregated &amp; Vectorized</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barrier</td>
<td>Barrier</td>
<td>Barrier</td>
</tr>
<tr>
<td>loop {</td>
<td>tmp = 0</td>
<td>tmp = 0</td>
</tr>
<tr>
<td>priv = ...</td>
<td>loop {</td>
<td></td>
</tr>
<tr>
<td></td>
<td>priv = ...</td>
<td></td>
</tr>
<tr>
<td></td>
<td>tmp = priv</td>
<td></td>
</tr>
<tr>
<td></td>
<td>}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Lock L0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>glob += priv</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unlock L0</td>
<td></td>
</tr>
<tr>
<td>Barrier</td>
<td>Lock L0</td>
<td>gtmp[pid] += tmp</td>
</tr>
<tr>
<td>Use glob</td>
<td>glob += tmp</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unlock L0</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Barrier</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Use (gtmp[0] + gtmp[1] + ...) in place of glob</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 4.5** Eliminating critical sections. The only shared variables are `glob` and `gtmp[]`. 
repeat
{
    Perform some computation
    priv = result of computation
    Lock L0
    global = global + priv
    Unlock L0
} until (computation not over)

repeat
{
    taskptr = DequeueTask()
    Process task using taskptr
    if (new tasks created)
        EnqueueTask (newtaskptrs)
} until (no tasks left)

(a) (b)

Figure 4.6 Two common ways of using critical sections. Except for priv, all other variables are shared. The code for DequeueTask() and EnqueueTask() is shown in Figure 4.9.

The enqueue and dequeue operations (see Figure 4.9 for the code) are carried out inside critical sections to ensure that the task queue is atomically updated. However, these critical sections also ensure that the dequeuer of a task sees the writes to the task data by the enqueuer. Thus, the accesses to the task data are ordered by the enqueue and dequeue critical sections. However, unlike accesses that are ordered using ns-atomic critical sections, the accesses to the task data take place outside these critical sections.

The distinction between these types becomes important when determining whether the synchronization making up a critical section is redundant. Before discussing how we detect ns-atomic critical sections, we describe aggregation and vectorization, two transformations that we can carry out on ns-atomic critical sections.

4.5.1 Aggregating Critical Sections

Consider the updates to a location that are made inside a set of ns-atomic critical sections, protected by a particular lock variable. If these updates are commutative and associative, they can be carried out in any order without affecting the final result. In this case, the synchronization in the program can be reduced by first aggregating the updates made by the processes into local variables, and then applying the aggregated updates to the original (shared) location. Figure 4.5 illustrates this. In the second
column. the program is transformed by accumulating the updates into a local variable \texttt{tmp}, which is then applied to the shared variable \texttt{glob} using a critical section.

4.5.2 Vectorizing Critical Sections

Another transformation can be used to further reduce synchronization when ns-atomic critical sections are used for updates that are commutative and associative. The third column in Figure 4.5 shows how the critical section introduced during aggregation can be removed completely by \textit{vectorizing} the update. This is done by having each process record its contribution to the update in a global vector (\texttt{gtmp[ ]} in the figure). As the critical sections are ns-atomic, the value being updated will be accessed only after the current b-interval. Hence, after the barrier marking the end of the b-interval, each process can compute the total update by including the contributions of the other processes.

In such cases, vectorization can completely eliminate the synchronization due to a set of critical sections. However, the elimination is achieved at the expense of increased computation. If there are \( n \) processes and each originally performs \( x \) operations to update a location, after vectorization, each process performs \( n \times x \) operations. Hence, vectorization should only be used when the benefits of reducing synchronization outweigh the cost of the increased computation. In particular, if each process is updating a large number of locations, the overheads due to vectorization may outweigh the gains from eliminating synchronization.

4.5.3 Detecting ns-atomic Critical Sections

We check for ns-atomic critical sections as follows. Consider all critical sections in a b-interval that are protected by the same synchronization variable. These critical sections are ns-atomic provided the conditions in Figure 4.7 are true for all of them. An informal definition of the terms "receive", "send", and "critical path" was presented in Section 4.4.3.

The first two conditions in Figure 4.7 ensure that the lock and unlock of the critical sections are not acting as communicators for accesses outside the critical sections. Thus, they ensure that the critical sections are not being used to order data accesses that lie outside them. For example, these conditions would exclude critical sections used to update a task queue. These two conditions have been specified by Adve [Adv96]. The third condition excludes cases where a value obtained from an-
1. The lock does not receive for any operations following the critical section.

2. The unlock does not send for any operations preceding the critical section.

3. Consider a critical path that terminates in a read inside the critical section, for which the lock is the receiver. Then, no operation that follows the critical section (by program order) must be control or data dependent on the value returned by the read.

**Figure 4.7** The three conditions for a critical section to be ns-atomic.

other process is used in computation following the critical section. Our use of critical paths in this condition is deliberate, and prevents an over specification compared to a case where only the uniprocess control and data dependences are considered. Together, the three conditions ensure that removing the lock and unlock operations of these critical sections do not affect the results of operations that are outside the critical sections.

Informally, the first two conditions state that a ns-atomic critical section must be a "critical section that is used only for atomicity" [Adv96]. This is because they ensure that the lock and unlock of the critical sections are not acting as communicators for accesses outside the critical section. The third condition excludes cases where a value obtained from another process (by means of the critical section lock) is used in computation following the critical section.

**Applying the Conditions Optimally**

We provide examples to illustrate that the conditions specified in Figure 4.7 can be applied non-optimally in the presence of pairwise synchronization, and/or when there is more than one set of critical sections, each protected by a different synchronization variable.

Figure 4.8 illustrates the case where the b-interval has pairwise synchronization. This figure shows the execution of two processes in a b-interval. In addition to the critical sections, pairwise synchronization is also present. From the figure, it is clear that in the presence of this synchronization, the two critical sections are not needed
to order the conflicting accesses to \( x \). This is because although there is a candidate critical path from the read of \( x \) to its write that has \( U0 \) as the sender and \( L1 \) as the receiver, there is another candidate critical path in which these are not the sender and receiver.

Figure 4.9 illustrates a case when the \( b \)-interval has two sets of critical sections, each protected by a different synchronization variable. While the critical sections protected by \( L0 \) impose an ordering on accesses to the tasks, the one protected by \( L1 \) is clearly ns-atomic. However, the conditions described in Figure 4.7 may not be able to detect the distinction. To see why, consider an execution of this code where process \( P_a \) dequeues a task, works on it, enqueues the two new tasks created, and updates \text{process\_tasks}, after which process \( P_b \) dequeues each task enqueued by \( P_a \), and processes it, updating \text{process\_tasks} twice. The partial order graph for this execution is shown in Figure 4.10.

Here, one candidate critical path for the conflicting accesses to task \( T0 \) has the unlock and lock of the critical section updating \text{process\_tasks} as the sender and receiver. These are marked as \( U \) and \( L \) in the figure. Using the conditions in Figure 4.7, we would classify this critical section as not ns-atomic. However, note that there is another candidate critical path between the conflicting accesses, that has the unlock and lock of the enqueue and dequeue critical sections as the sender and receiver.

The conditions in Figure 4.7 must be applied intelligently because there is a lot of \textit{choice} in forming a critical set of paths. Of all the candidate critical paths for a pair of conflicting accesses, only one needs to be in the critical set. Hence, if we do not choose the critical set of paths carefully, we could end up classifying a critical section as not ns-atomic, when it really is.
Barrier
repeat
{
    taskptr = DequeueTask ()
    Process task (use taskptr)
    if (new tasks created)
        EnqueueTask (newtaskptrs)
    Lock L1
    processedtasks++
    Unlock L1
} until (no tasks left)
Barrier

DequeueTask ()
{
    Lock L0
    localptr = queue[--head]
    Unlock L0
    return localptr
}

EnqueueTask (taskptrs)
{
    Lock L0
    for (each ptr in taskptrs)
        queue[++head] = ptr
    Unlock L0
}

**Figure 4.9** Testing for ns-atomicity in the presence of multiple sets of critical sections. The change bar indicates the difference between this code and that in Figure 4.6(b).

**Figure 4.10** The partial-order graph for an execution of the code in Figure 4.9. The shaded critical sections represent the updates to `processedtasks`. $U$ and $L$ are described in the text.
We solve this problem using the same approach we used in Section 4.4.2. For each interval outside a critical section, we find the critical sections and acquires (due to pairwise synchronization) that can receive for the accesses in the interval. We can do this by checking for conflicts between the accesses in the interval outside the critical section, and those on the other processes that precede it (by execution order). Similarly, for each interval outside a critical section, we find the critical sections and releases (due to pairwise synchronization) that can act as the sender for the accesses in the interval. Then, we set up an “enforced-by” equation similar to equation 4.5, in which the send-receive requirements on the left hand side are satisfied by the terms on the right hand side. In the equation, variables corresponding to synchronization that is required are set to true. These correspond to the releases and acquires due to pairwise synchronization that are required, and the critical sections that cannot be ns-atomic because they fail the conditions in Figure 4.7. We then use the methods of Section 4.4.2 to obtain the minimal set of critical sections, from among the critical sections in the program, that can enforce the dependences. This gives us the maximal number of ns-atomic critical sections, from among the critical sections present in the program.

We do not expect the “enforced-by” equation to have a large number of terms on the right hand side. The reason for this is as follows. If a synchronization that enforces a dependence is required (because it corresponds to pairwise synchronization that is required or to a critical section that is not ns-atomic), the dependence and the corresponding term in the equation can be excluded. This brings down the size of the right hand side.

If the equation does have a large number of terms, we resort to integer programming. A description of the integer programming formulation can be found in our earlier work [RC97a].

### 4.6 Restructuring Computation

Even if a program contains no excess synchronization as written, it may be possible to reduce synchronization and data communication by changing its computation structure.

---

1 Before we solve the equation, we weight each critical section term with the number of times it is invoked in the program.
If the results of some computation are not needed immediately and the source operands are still available later, the computation can be postponed. Postponement will be profitable if the dependences due to the computation can be enforced by other synchronization. Synchronization that is otherwise necessary may then be weakened or even eliminated, resulting in improved performance.

Figure 4.11 explains this with an example. On both left and right sides, process $P_b$ reads data written by the other processes after synchronizing with them. This true dependence necessitates synchronization $S0$. After reading $x$, $y$ and $z$, $P_b$ writes $t$. The result of this computation, $t$, is not read by the other processes after synchro-

![Diagram showing synchronization and computation]

**Figure 4.11** Eliminating synchronization by postponing computation. The execution of four processes is shown, with computation indicated in the boxes. In the left side, synchronization $S0$ can be eliminated by postponing the computation of $t$. In the right side, postponement is not possible. $S1$ is required due to dependences not shown in the figure. Shaded computation is not relevant to the discussion.
nization $S1$. On the left side of the figure, the source operands of the computation ($x$, $y$, and $z$) are still available unmodified in the interval $(S0, S1)$ and in the interval beginning with $S1$. Hence, the computation of $t$ can be postponed to after $S1$, eliminating the need for synchronization $S0$.

On the right side of Figure 4.11, $x$, $y$ and $z$ are modified after $S1$. The source operands will therefore not be available if the computation is postponed. Hence, the computation of $t$ cannot be postponed, as it would affect program correctness.

The amount of data communicated is also an important factor in determining program performance. Ideally, computation should be performed by the process that has computed most of the source operands, and uses or reads most of the results. By attributing data movement to the computation whose memory accesses induce the movement, we can determine if it would be better for the computation to be performed on another process. If this relocation eliminates all the data dependences enforced by some synchronization, the synchronization can also be removed.

In Figure 4.12, process $P_b$ performs some computation that uses data created by $P_a$. The results of this computation are again used by $P_a$. Hence, as it stands, there is data communication between the processes, and synchronization $S0$ and $S1$ are required. Both the data communication and the synchronization can be removed by relocating the computation to $P_a$.

Figure 4.12 Eliminating communication and synchronization by relocating computation. Arrows indicate data dependences.
Two pieces of information are required in order to restructure computation. First, we need to know the source operands of the computation. Second, we need to know the cross-process data and control dependences. We describe how we obtain this information only later, in Chapter 5. In the next two sections, we discuss the conditions under which computation can be postponed or relocated.

4.6.1 Postponing Computation

Consider four adjacent barriers $B0$, $B1$, $B2$, and $B3$ that create three adjacent $b$-intervals in the computation. Let $B1$ be required to enforce true dependences. Our goal is to see if $B1$ can be weakened or removed by postponing computation in the second $b$-interval to the third.

We start by identifying the computation in the second $b$-interval on a process $P_a$, that has source operands that were written in the first $b$-interval on another process $P_c$. Figure 4.13 illustrates this for a three-process case. If this computation can be postponed to beyond $B2$, $P_c$ does not have to synchronize with $P_a$. To see if this is possible, we ask two questions:

1. Are the results of this computation used on any process in the interval $(B2, B3)$?
2. Are the source operands of this computation modified in the interval $(B1, B3)$?

![Diagram of process synchronization](image)

Figure 4.13 The two conditions that must be satisfied to postpone computation. The black circle denotes computation, and the shaded ones, its source operands.
The first question can be answered by determining if the writes produced by the computation are accessed in the interval \((B2, B3)\). To answer the second question, we first determine all the source operands of the computation. Then, we check to see if any of these locations are written on any process in the interval \((B1, B3)\).

If both these questions are answered in the negative, the computation can be postponed. Once this is done, the synchronization from \(P_c\) to \(P_a\) in barrier \(B1\) is not required to enforce any true dependences. If there are no anti- or output dependences between these intervals, the synchronization can be removed. Otherwise, it can be weakened (see Section 4.4.1).

Special care is required when postponing I/O statements. Specifically, in addition to the requirement that the values being printed remain unchanged, we also require that the order of I/O operations remain unchanged.

### 4.6.2 Relocating Computation

We use a simple approach to detect cases where synchronization and communication can be reduced by relocating computation. Our approach checks if part or all of the computation done by a process in a \(b\)-interval is better done on another process.

As before, consider four adjacent barriers \(B0\), \(B1\), \(B2\), and \(B3\) that create three adjacent \(b\)-intervals in the computation. Our goal is to see if the synchronization and communication due to barriers \(B1\) and \(B2\) can be reduced by relocating the computation in the interval \((B1, B2)\) from one process to another.

![Figure 4.14 Relocating computation. The computation indicated by the black circle should be relocated to the process that supplies most of the source operands (shaded circles) and uses most of the results (white circles).](image-url)
We start by identifying the computation in the second b-interval on a process \( P_a \) that has source operands that were written in the first b-interval on another process \( P_c \). Figure 4.14 illustrates this for a three-process case. Next, we determine two pieces of information:

1. The source operands for this computation, that were created in the interval \((B0, B1)\) on each process. The identity of the source operands is not required; all we need is the amount of source operand data that each process "contributes" to the computation.

2. The results of the computation, that are used (i.e., read) by each process in the interval \((B2, B3)\). Again, the identity of the results that are used is not required; all we need is the amount of result data that is read by each process in \((B2, B3)\).

For each process, we add these two amounts together. We relocate the computation to the process for which this sum is the maximum, if it is different from the one on which the computation is currently performed.

Relocating computation with the aim of reducing synchronization and communication is at cross-purposes with parallelization. For instance, reducing communication by relocating computation greedily, could result in all the computation being collapsed onto a single process. Therefore, load balancing also needs to be taken into account. We use the amount of shared data written during the computation as a rough metric of the time spent in the computation. Hence, after determining computation that is to be relocated, we check if the relocation will skew the distribution of writes to shared data by the processes in that b-interval. If the distribution will be skewed, we do not carry out the relocation.

4.7 Related Work

To the best of our knowledge, we are the first to address the detection and elimination of excess synchronization from explicitly parallel shared-memory programs [RC96].

Parallelizing compilers have long examined the issue of reducing the amount of synchronization inserted during the parallelization process. Typically, synchronization is inserted conservatively during parallelization, to preserve dependences. A final pass then examines the program and removes unnecessary synchronization. Excess
synchronization is detected by analyzing the data access behavior of the program. Data dependence analysis [PW86] forms the core of all compile-time data transformations.

Li and Abu-sufah have presented a technique to remove redundant synchronization interlocks (for instance, enforced with a post and wait condition) from doacross loops [LAS85]. An interlock is required in the presence of loop-carried flow- and anti-dependences. They detect simple cases where synchronization inserted to satisfy a dependence satisfies another as well, and implement their technique as a pass in the Parafrase source-to-source restructurer. Midkiff and Padua extend this to eliminate redundant synchronization operations from loops with arbitrary control flow [MP87]. Both algorithms can be applied to reduce the synchronization added when parallelizing loops. However, we have shown that the synchronization added by these algorithms is not optimal [RC97a].

Quinn, Hatcher and Seevers describe an algorithm to insert the minimal number of barriers when compiling a synchronous data parallel program [QHS91].

Prakash et al. address reducing synchronization costs when compiling UC, a synchronous data parallel extension of C [PDB93]. They describe a set of transformations to reduce synchronization such as eliminating barriers and replacing it with point-to-point communication. They also describe how definition variables⁵ array renaming, and array alignment can be used to replace barriers with explicit communication. In order to arrive at these transformations, they require data dependences to be known at compile time. The drawback of this work is that they just stop at describing the transformations. Specifically, they do not give any algorithms for arriving at these transformations. In addition, their evaluation is based on a simplistic analytical model.

Philippsen and Heinz present an algorithm [PH95] for eliminating redundant barriers from synchronous FORALL loops in Modula-2*, a parallel extension of Modula-2. They first construct a graph representing both dependence information and expression trees (as used in intermediate representations during compilation). The minimum number of synchronization barriers required is given by the maximum number of dependences encountered along any path in the graph. For nodes that are not on the maximal path, there is some freedom in terms of where the corresponding

⁵A definition variable [CT91] (DV) is a data structure associated with a queue. Requests are queued until the data structure is initialized, and are then satisfied. A DV can be used in producer–consumer situations permitting a producer to write ahead and a consumer from reading data not yet produced.
expression can be evaluated. They use heuristics based on a topological sort of the graph to determine this position, thereby decreasing the storage needed for intermediate variables. Their base compiler inserts a barrier after each expression evaluation. Hence, programs analyzed by their implementation perform very well compared to the original compiler.

Tseng has implemented a barrier elimination algorithm [Tse95] in the SUIF compiler [AALT95]. Two optimizations are used to reduce overhead and synchronization. By combining adjacent SPMD regions, the overhead of starting up parallel tasks is reduced. By augmenting dependence analysis with communication analysis [HKT92], the compile-time compute partitioning is taken into account when checking whether a barrier is needed. In addition, the communication that needs to occur at a barrier (obtained using communication analysis) is matched against some simple patterns corresponding to nearest neighbor, scatter or gather operations. The barrier is replaced by counters or flags in this case. The performance of the implementation is compared against that of the original compiler; modest improvements are reported.

O’Boyle and Bodin describe a similar approach for reducing the number of barriers when compiling into SPMD code [OB95]. Once loop and data partitioning have been completed, they use dependence analysis to insert the minimal barriers required to protect inter-processor dependences. This translates into the maximal cut problem [GJ79]: they hence use a heuristic solution to insert the barriers. They also examine whether barriers within loops can be replaced with wait/post synchronization. They compare their compiler implementation against KAP and provide results for two kernels on a 32-processor KSR.

None of the above approaches look at optimizing critical sections. Furthermore, since all the approaches are based on data dependence analysis, the amount of synchronization they can reduce is affected by the accuracy of the static analysis. As we explain in the next chapter, by basing our analysis on run-time information, our approach can provide feedback on how to reduce synchronization and communication in complex, pointer–ridden applications.

Krishnamurthy and Yelick present compiler optimizations for explicitly parallel shared–memory programs [KY95]. In such programs, code motion optimizations require consideration of inter–process dependences. In order to guarantee sequential consistency after reordering transformations have been applied, analysis introduced by Shasha and Snir known as cycle detection [SS88] is required. Krishnamurthy and Yelick improve the accuracy of cycle detection by incorporating additional informa-
tion available from the explicit synchronization in the program. Consequently, scalar optimizations may be applied around synchronization primitives.

Diniz and Rinard present three transformations that allow a compiler to decrease the overhead of critical sections used for mutual exclusion in object-based programs [DR96, DR97]. Lock cancellation eliminates acquires and releases that access the same lock and have no intervening computation. Lock movement moves acquire and release nodes across computation. Together, these two transformations permit lock elimination, which coarsens a critical section by increasing the computation performed inside it.

Their approach has two drawbacks.

First, their transformations assume that critical sections will only be used for mutual exclusion, thereby restricting the semantics of the lock and unlock operations. Therefore, their techniques cannot, in general, be applied to sequentially consistent programs. For instance, consider a task queue based program, such as that shown in Figure 4.6. In a sequentially consistent version of this program, only the accesses to the task queue need to be protected using a critical section. In particular, accesses to the tasks themselves need not be protected using critical sections. If applied to such a program, the transformations implemented by Diniz and Rinard will violate program correctness. In contrast, our algorithms do not restrict the synchronization semantics.

Second, their compile-time analysis is able to arrive at the critical section transformations primarily because the programs being considered are object-based [DR97]. It is unknown how much the efficacy of this approach will be affected when applied to non-object-based programs, for which the static analysis is likely to be less precise.

Several of the algorithms presented in this chapter have been explained using a framework for reasoning about memory models introduced by Adve [Adv93]. Adve has used this framework to explore system optimizations that do not violate sequential consistency, based on information supplied by the programmer [Adv96]. In contrast, we use her framework to determine excess synchronization or synchronization that is stronger than necessary.

Adve et. al. have looked at the benefits of relaxing the strict semantics of the system-provided lock and unlock synchronization constructs in a lazy release consistent system [ACDZ94]. They look at three constructs using which a programmer can communicate the intended use of a critical section. Their run-time system then uses this information to implement a weaker and more efficient form of the critical
section. In contrast, our focus is on inferring the intentions of a programmer in using a particular critical section. When we detect that the synchronization that is used can be relaxed, we provide transformations that eliminate the need for the critical sections.

4.8 Summary

The first requirement for designing a prescriptive tool is the ability to automatically analyze run-time data to derive feedback. In this chapter, we describe three methods for detecting excess synchronization and reducing data communication in shared-memory parallel programs. These algorithms form the basis for the automatic data analysis in \( R_X \).

First, we examined how to eliminate synchronization that is not required for a program to execute correctly. We detect such unnecessary synchronization by checking for conflicting accesses when processes synchronize. We looked at barriers, presenting the conditions under which they can be weakened to partial barriers, control barriers, and finally to just pairwise synchronization. For pairwise synchronization, we considered cases where a dependence may be enforced by more than one synchronization operation. We obtain the minimal set of synchronization (from among those present in the program) that enforces all dependences by expanding into disjunctive normal form, an equation that represents the enforcement of all dependences. In earlier work, we have shown that a similar problem is encountered in a compile-time context, when minimizing synchronization that must be added while parallelizing loops [RC97a]. We also described how to reduce the time spent inside critical sections by moving outside those operations that do not have to be performed inside the critical section.

Our second method for reducing synchronization and communication involves transforming critical sections in barrier-separated intervals of the program. We described two transformations, aggregation and vectorization. These transformations can be correctly applied only to non-synchronizing atomic critical sections. We describe how these critical sections can be identified. In the presence of pairwise synchronization, or when there are multiple sets of critical sections protected by different synchronization variables, we may have a choice in determining the maximal set of ns-atomic critical sections. As in the case of pairwise synchronization, we arrive at the maximal set by expanding the equation representing the enforcement of all dependences.
Our final method for reducing synchronization and communication involves computation restructuring. Postponing computation allows synchronization to be weakened or removed. Relocating computation can reduce data communication, and may also permit synchronization to be weakened or removed.

We arrived at this set of algorithms by examining a large number of shared-memory applications, including several standard benchmarks. We were able to detect several instances of excess synchronization in these applications, as also possibilities for reducing synchronization and data communication. Our observations motivated the algorithms described in this chapter. We do not claim that these algorithms will uncover all possible cases of excess synchronization and data communication. However, the methods we have outlined in this chapter can be extended to handle other cases that may arise.

These algorithms can be implemented in a compiler. However, their efficacy will then depend on the accuracy of static dependence analysis. Our primary goal was to successfully performance tune complex, pointer-ridden applications. Hence, our focus in this dissertation is on a run-time approach.

In the following chapters, we describe how we implemented these methods in RX. Several challenges need to be overcome to make our run-time approach *tractable*. Two of the most important issues are what information to collect at run-time, and dynamically finding out the source operands of writes to memory. In subsequent chapters, we describe the methods we have developed to tractably address these problems.
Chapter 5

Preprocessing and Data Collection

5.1 Introduction

In Chapter 3 we gave a broad overview of $R_X$. Several challenges need to be surmounted in order to reduce synchronization and data communication using a run-time tool. Two of the important issues are what information to collect at run-time, and dynamically determining the source operands of writes to memory. In this chapter, we present the techniques we use to tackle these and other issues.

The basic technique we use to identify excess synchronization is conflict analysis. By collecting the set of locations accessed in each interval during program execution, we can later determine if the accesses in two intervals need to be synchronized. This enables us to detect excess barriers and excess pairwise synchronization such as flags.

By itself, conflict analysis does not provide the information needed to detect the other forms of excess synchronization described in Chapter 4. We need to perform three specific operations in order to prescribe computation restructurings, and transformations involving critical sections. These are:

- Correlate line numbers of specific accesses with the source program.
- Determine if a read uses values written by an earlier write.
- Determine the source operands of each write.

All of this information can be obtained by recording an augmented access trace of the execution. For instance, the correlation can be achieved by also recording the line number associated with each access. The source operands of a write (i.e., the set of memory locations on which the write depends) can be obtained in two steps: (i) instrument the program to link each write with its source operands, and (ii) record the source operands at run-time.

The drawback of this approach is that it is not tractable. A program that executes for even a few seconds can produce large quantities of information that must not only be stored, but also processed. This would significantly restrict our ability to
performance tune long-running applications. A more careful approach is required in order to tractably collect the information required for the automatic analysis.

This chapter focuses on the methods we have developed for tractable data collection. We use a two-step process to generate the information. First, we instrument the source program to gather certain information at run-time. This information is stored as state associated with the memory locations of the program. Then, when the program executes, we process this state using a set of run-time algorithms, producing the information required for the analysis. Partially processing run-time data on-the-fly, enables use to significantly reduce the amount of information that must be stored for subsequent analysis. We also use a novel approach for collecting source operands tractably: one that groups writes together and collects operand information in terms of equivalence classes. This can be decoded later to obtain the source operands of writes to memory.

The chapter organization is as follows. In Section 5.2, we restate the requirements of our performance debugging approach. These requirements are necessitated by the algorithms we use to reduce synchronization and data communication. In Section 5.3, we describe how we collect the read and write sets of each interval. These sets are used to detect conflicting accesses.

The next three sections describe the details of the two-step process that we employ to tractably collect the information needed to prescribe some transformations. In Section 5.4, we explain how to correlate computation that can be moved outside critical sections with the source program. In Section 5.5, we explain how to gather information required to detect ns-atomic critical sections. In Section 5.6, we explain our techniques for collecting information required to restructure computation.

The instrumentation we add could significantly perturb the run-time behavior of the instrumented program. We address this issue in Section 5.7. In Section 5.8, we provide details of our run-time system. Finally, in Section 5.9, we summarize the contributions of this chapter.

5.2 High-Level Requirements

Our approach uses information about the synchronization accesses in the program to both collect and analyze information about the program execution. Hence, we require that the programmer make all the synchronization operations in the program known to $R_x$. This can be done by using the synchronization primitives provided by the run-
time system. Since most programs make use of run-time libraries for synchronization anyway, this requirement is not very restrictive. If the synchronization accesses are not all distinguished, the feedback provided by $R_c$ may be incorrect.

In addition, the program under consideration must be free of data races. Data races may cause the behavior of a program to be time-dependent. Hence, their presence is often indicative of incorrect program execution. Therefore, it is not unreasonable to require that the program being analyzed be free of data races. Several techniques can be used to assist in the detection of data races; the work of Perkovic and Keleher [PK96] is an example.

### 5.3 Conflict Analysis

The presence or absence of conflicting accesses is a key factor that determines whether synchronization can be removed. Synchronization is needed between two intervals only if the accesses made within them conflict. This can be used to detect excess pairwise synchronization (Section 4.4.2), and barriers (Section 4.4.1). The absence of conflicts also determines whether computation can be moved outside critical sections (Section 4.4.3), and is also instrumental in determining whether critical sections can be aggregated and vectorized (Section 4.5). Finally, postponing computation, which checks for the availability of source operands and whether results are used, also requires analyzing for conflicting accesses (Section 4.6.1).

The information required for conflict analysis is recorded as follows. We instrument the loads and stores of the program. Only potential accesses to shared locations need to be instrumented, as only these can conflict. Our run-time system allocates all shared memory from the heap (see Section 5.8). This allows us to avoid instrumenting accesses that are clearly to private data, such as those through the stack and frame pointers. To check for conflicts, the order in which accesses are made within intervals is irrelevant. Therefore, we aggregate the accesses and record them as access ranges. Aggregation permits a very compact representation in the log files. An exact description of the run-time algorithm we use is provided later in Figure 5.5.

We treat I/O to and from shared memory as writes and reads respectively. Since the order in which I/O operations are performed is important, we also record the order in which all I/O operations take place on a per-descriptor basis. Later, when restructuring computation, we make sure that the order of I/O operations (on each file descriptor) will be preserved.
5.3.1 Procedure Calls

Consider a situation where a synchronization operation is encapsulated inside a function foo(). Let this synchronization operation be unnecessary when foo() is called from one procedure, but required when foo() is called from other places. In this case, Rx will only be able to determine that the synchronization inside foo() is not always required. To make this feedback more useful to the programmer, Rx must inform when the synchronization is not required. This can be done by collecting a procedure call trace. Our prototype implementation does not collect this information.

5.4 Moving Computation Outside Critical Sections

By itself, conflict analysis can only identify which shared accesses can be moved outside a critical section. To be of use to the programmer, we need to specify the lines in the program that correspond to these accesses.

The straightforward methods for collecting information that enable shared accesses to be associated with source code also tend to drastically increase the amount of collected information. For instance, along with each access, we could record the program counter (or line number) to which it corresponds. Alternately, we could collect a full access and basic block trace of the program. However, both these methods would severely reduce the ability to aggregate accesses, which is essential to compact logging.

We prevent an explosion in the amount of data collected, by associating state with each shared memory location and processing it at run-time. The state keeps track of when and where a location was last accessed. The run-time processing uses this state to determine accesses that can be moved outside critical sections.

With each shared memory location, we associate the following state:

READLINE: Indicates source code line from which this location was last read.
READPROC: Indicates the process that last read this location.
WRITELINE: Indicates the source code line from which this location was last written.
{W|R}ICS: These stand for {Read|Wrote} Inside Critical Section, and indicate whether the accesses made at READLINE and WRITELINE were from inside critical sections.

The algorithm for processing this state at run-time is given in Figure 5.1. The abbreviations MFL and MPU stand for Must Follow Lock and Must Precede Unlock respectively. We explain each step in the algorithm below.
1. On every load, if the location is shared
   (a) If the location was last written on another process:
       • If WICS is set, mark WRITELINE of the location as MPU.
       • If the load is inside a critical section, mark its source line as MFL.
   (b) Set READPROC to the current process id.
   (c) If the load is inside a critical section, set RICS and READLINE to the current line.

2. On every store outside a critical section, if the location is shared:
   (a) If the location was last read on another process
       • If RICS is set, mark READLINE of the location as MPU.
   (b) If the location was last written on another process
       • If WICS is set, mark WRITELINE of the location as MPU.
   (c) Reset READPROC, RICS, and WICS.

3. On every store inside a critical section, if the location is shared:
   (a) If the location was last read or written on another process
       • Mark the source line of the store as MFL.
       • If RICS is set, mark READLINE of the location as MPU.
       • If WICS is set, mark WRITELINE of the location as MPU.
       • Reset READPROC and RICS.
   (b) Set WICS and WRITELINE to the current line.

4. On reaching a barrier:
   (a) Log all marked lines, with the marks (MPU, MFL). Reset the marks.
   (b) Reset WRITELINE and READLINE for locations accessed in the b-interval.

Figure 5.1 The run-time algorithm to determine if accesses can be moved outside critical sections.
1. When a shared location is read, we check to see if it has been written in this b-interval on a different process. How we determine this is described later, in Figure 5.5. If so, we check if the conflicting write was performed inside a critical section. We then consider the unlock following the write to be a sender, and require the write to precede the unlock. If the shared read is inside a critical section, we consider the lock preceding the read to be its receiver. The read must therefore follow the lock.

Finally, if the read is inside a critical section, we mark it so and record its source line.

2 & 3. When a shared location is written, we check to see if it was read and/or written on another process in this b-interval. Note that both reads and writes can conflict with a write. If the shared write is inside a critical section, we consider the lock preceding the write to be a receiver for the write. The write must hence follow the lock. We also check if an access that conflicts with the write was performed inside a critical section. If so, we consider the unlock following the access to be a sender, requiring the access to precede the unlock.

Finally, if the write is inside a critical section, we mark it so and record its source line.

4. At the end of a b-interval, we record the source lines corresponding to accesses that must follow the lock or precede the unlock. We discuss the performance debugger actions to process this information in Section 5.4.1.

As all our critical section analysis is carried out in b-intervals, we need to reset state before beginning a new b-interval. Thus, on reaching a barrier, we reset WRITE_LINE and READ_LINE for all locations accessed in the b-interval ending with the barrier. We also reset the source line marks.

5.4.1 Processing the Source Line Information

The result of the run-time processing in Figure 5.1 is a set of source lines marked MPU and/or MFL. Computation corresponding to the source lines marked both MPU and MFL must be performed inside the critical section. Subject to uniprocess control and data dependences, unmarked source lines can be placed outside the critical section. Again, subject to the uniprocess dependences, source lines marked only as MPU or MFL can be placed before or after the critical section respectively.
Only the accesses in \((A \cap B)\) need to be inside the critical section. However, even if the accesses from a line in the program belong to \(A\) and \(B\) at different times, we report that the line be left inside the critical section.

5.4.2 Limitations

Our prototype implementation has two limitations in its ability to identify computation that can be moved outside critical sections. We discuss them in this section.

Ideally, we would like to resolve the control and data dependences, and precisely identify the computation that can be moved outside the critical section. However, a limitation in the information that we can tractably collect at run-time makes the identification difficult. The problem is not in resolving control and data dependences at run-time. In fact, we have to do this to determine the source operands of computation. We explain how we do this later, in Section 5.6.2. The difficulty arises because we have to preserve the dependence information. Consider an access to a shared location inside a critical section. It is a subsequent conflicting access to this location that determines whether the first access must precede the unlock. If we want to identify the computation on which the first access is dependent, we must preserve the dependence information involving the first access until the second conflicting access occurs. Preserving a potentially large amount of information in order to resolve the dependences is impractical.

We can consider two solutions to this problem. The simple solution is to require the programmer to ensure that the uniprocess control and data dependences are preserved before moving computation outside the critical section. We take this approach in our prototype implementation. Alternately, after recording the marked accesses, we can carry out a second performance debugging pass. During the second pass, we will have the information to process the dependence data as it is generated. This run-time processing can ensure that any computation that is data or control dependent on the marked accesses will also be marked appropriately. A requirement of this approach is that the second execution be a replay of the first. To handle the perturbation effects of the added instrumentation, we have to do this anyway (see Section 5.7).

The second limitation of our prototype is more involved. Our prototype only preserves the last source line at which a write or read is performed. Since an access is marked MPU only when it conflicts with a later access, preserving only the last source
line precludes us from marking previous accesses to the same location. Figure 5.2 illustrates the problem with three examples. In examples (a) and (b), our run-time algorithm will mark only the access just before the unlock as MPU. In example (c), only the read to \( x \) before the second unlock will be marked MPU.

Again, we can consider two solutions to this problem. The first solution is to provide a sufficient number of READLINE and WRITELINE states for each location. As the problem arises only when the same location is accessed from multiple source lines, a case that is unlikely to happen often in practice, a couple of extra source line states per location should suffice. We take this approach in our prototype implementation. In fact, for all the applications we consider in Chapter 7, no location was ever read or written twice inside critical sections without an intervening write on another process. The second solution is more complicated, and involves a second performance debugging pass. In the first pass, for each access that gets marked MPU, we can record two attributes. These are (i) the location accessed, and (ii) the type of the access with which this access conflicts. Then, we can replay the program execution, as we describe in Section 5.7. Within a critical section, when a recorded location is accessed, we can check the type of the access with which it conflicts (read and/or write). Since we now know the type of the conflicting access, if the current access will also conflict with it, we can mark it also as MPU. Thus, in Figure 5.2(c), on encountering the first read to \( x \), we would mark it as MPU because we know that it will conflict with the subsequent write to \( x \).
5.5 Detecting *ns*-atomic Critical Sections

Detecting critical sections using the specification given in Section 4.4.3 is fairly straightforward. Inside a b-interval, we look for lock-unlock pairs that are protected by the same synchronization variable, and that encapsulate only synchronization making up critical sections. Our run-time system ensures that the locations accessed by the lock and unlock operations cannot be accessed by other (data) accesses.

Conflict analysis can be used to check for the first two conditions of Figure 4.7. Checking for the third condition requires the additional run-time support shown in Figure 5.3.

With each location (shared and private), we associate the following state:

**WRITECSID**: If the corresponding location was last written inside a critical section, and depends on data created on another process, this field specifies the synchronization variable which protects the critical section.

**WRITEBINT**: This indicates the b-interval in which the location was written.

The explanation of the algorithm in Figure 5.3 is as follows. If a write inside a critical section depends on data created on another process, we mark the loca-

<table>
<thead>
<tr>
<th>1. On every load, if WRITECSID for the location is set to $S$, record that the set of critical sections protected by $S$ is not <em>ns</em>-atomic provided:</th>
</tr>
</thead>
<tbody>
<tr>
<td>- The load is not from inside a critical section protected by $S$. and</td>
</tr>
<tr>
<td>- The current b-interval is the same as WRITEBINT.</td>
</tr>
<tr>
<td>2. On every store</td>
</tr>
<tr>
<td>- If the store is from within a critical section protected by $S$, and if it is dependent on data created on another process,</td>
</tr>
<tr>
<td>- Set WRITECSID to $S$</td>
</tr>
<tr>
<td>- Set WRITEBINT to the current b-interval</td>
</tr>
<tr>
<td>- If the store is outside any critical section, reset WRITECSID.</td>
</tr>
</tbody>
</table>

---

**Figure 5.3** The run-time algorithm to check for the third condition from Figure 4.7.
tion with the synchronization variable S, that protects the enclosing critical section. Subsequently, if this location is read anywhere other than inside a critical section protected by S, we mark the set of critical sections protected by S as not ns–atomic. WRITEBINT ensures that we only mark a critical section as not ns–atomic if there is no intervening barrier between the write inside the critical section and the subsequent read. Since our run–time system constrains b–interval id’s to increase monotonically, using WRITEBINT enables us to avoid clearing state for data, especially private data.

The missing piece of the algorithm is how we determine if a write is dependent on data created on another process. We explain how to do this in the Section 5.6, on restructuring computation.

5.6 Determining Computation Restructuring

Two pieces of information are needed to reduce synchronization and communication by restructuring computation, as described in Section 4.6. First, to determine the legality of postponing computation, we have to check if accesses in different intervals conflict. We use conflict analysis, described in Section 5.3, to do this. Second, we need to know the source operands of computation. We do this tractably, by grouping writes into equivalence classes, and collecting the source operands of a write in terms of the equivalence classes of its operands. This approach allows us to performance tune long–running applications.

The static and run–time support needed to do this is fairly involved. We again require state to be to be associated with memory locations and source lines. In order to determine the source operands of computation, we need to know the memory locations making up the source operands of each write. Instrumentation that we add keeps track of the source operands of the data in each machine register. When this is combined with the locations accessed on memory loads, we are able to determine the operands of each store.

Control dependences pose a special problem. For instance, a store buried inside a function could be control dependent on a load performed in the calling context. To handle this, we statically analyze the control flow graph of the program and instrument the basic blocks that can change the control flow. Then at run–time, we use this information to convert control dependences into data dependences.

To improve efficiency, we combine the source operand generation and its run–time processing. For each basic block in the program, our static analysis creates
a specialized \textit{mirror} function that first generates the source operands in terms of memory locations, and then processes them on-the-fly to compactly represent them in terms of the equivalence classes. All of the other run-time processing described in this chapter is incorporated into functions that are then invoked from the mirror functions.

All of the static analyses we describe below is carried out with the assembly code output from the compiler.

5.6.1 Equivalence Classes

As a program executes, it writes values to memory that are derived from the contents of other memory locations. Thus, during execution, a write can be expressed in terms of earlier writes to the memory locations on which it depends at that point. These are the source operands of the computation. However, collecting operand information at such fine detail presents several problems. The amount of memory required at run-time to keep track of this information quickly becomes very large. Storing the collected data for analysis also becomes impractical. Data analysis in the performance debugger typically requires (costly) combinatorial operations such as set intersections and unions, which with detailed operand information, can become time-bound. All of these factors combine to make this approach impractical.

We can avoid these problems by grouping writes to memory locations into \textit{equivalence classes} and collecting the operand information for later writes in terms of these equivalence classes. By referring to the group instead of to the individual writes, a smaller amount of data gets collected. Different criteria can be used to form equivalence classes: we describe the method we use below.

As we described in Section 4.3, synchronization operations divide the execution of a process into intervals. Consider the writes made by a process in a particular interval. We can group these writes into an equivalence class characterized by a \texttt{[process,interval]} tuple. This tuple specifies the process and the interval (on that process) in which the writes were performed. Then, at a later write, instead of recording the actual locations on which the write depends (i.e., the source operands), we use the equivalence classes corresponding to the last writes to these locations. Figure 5.4, showing the execution of processes $P_a$, $P_b$ and $P_c$ illustrates this.

Here, the source operands of the write to $t$ by $P_a$ are $q$, $s$, $u$, $v$, $x$, $y$ and $z$. These locations were last written by $P_a$ and $P_c$ in interval 2 and $P_b$ in interval 3. The
operand information can hence be represented compactly as

\[ \text{Writes in } [P_a, 2], [P_b, 3] \& [P_c, 2] \quad \text{Write to } t \]

For brevity, in the following discussion, we represent this just as

\[ [P_a, 2], [P_b, 3], [P_c, 2] \quad \text{Write to } t \]

where this is understood to mean that the source operands for the write on the right hand side come from the writes in the intervals on the left hand side.

Defining equivalence classes in this manner enables us to reduce the amount of operand data collected. Although a write may have a large number of source operands, the number of distinct \([\text{process}, \text{interval}]\) tuples to which its operands belong is much smaller.

For analysis by the performance debugger, we need to regenerate the memory locations making up the source operands. We can do this using the set of shared locations read and written in each interval. We already record this information to carry out conflict analysis. By combining this with the operand information, we can determine a superset of the writes that make up the source operands.
5.6.2 Collecting Source Operands

We now explain the run-time algorithm used to collect operand data at the level of [process,interval] tuples. Our algorithm works by associating certain state with all memory locations and source lines that correspond to shared stores. This state is used to record operand information as well as the shared locations accessed in an interval. We use the symbol table information to map computation (i.e., writes) to lines in the source code.

A private location has only the PROCINT state. Source lines that could correspond to shared stores also have WRITE associated with them. Locations in shared memory have READ associated as well.

PROCINT: The PROCINT field of a location keeps track of the [process,interval] tuples of the source operands of the data in the location. It specifies when and where the operands were created. For a source line, the PROCINT field keeps track of the PROCINT fields of all stores in this interval, that correspond to the line.

WRITE: Let \( P_{\text{curr}} \) refer to the process executing the algorithm. Then, if a shared location has WRITE set to \( P_{\text{curr}} \), it was written by \( P_{\text{curr}} \) in this interval. If a source line has WRITE set to \( P_{\text{curr}} \), a store corresponding to the line wrote a shared location in this interval.*

READ: If a shared location has READ set to \( P_{\text{curr}} \), it was read by \( P_{\text{curr}} \) in this interval.*

The state for private locations and source lines is maintained in private memory. For shared locations, PROCINT and WRITE are maintained in shared memory, but READ is maintained in private memory.

Figure 5.5 specifies the SRCOPS algorithm that uses this state to arrive at the dependence information. \( P_{\text{curr}} \) signifies the process executing the algorithm. We explain each step in the algorithm below.

1. When a shared location is read, we set READ to \( P_{\text{curr}} \). This permits us to later determine the shared locations read by \( P_{\text{curr}} \) in this interval, which we need for conflict analysis. Note that READ is maintained in private memory.

2. When a shared location is written, we set WRITE to \( P_{\text{curr}} \). This permits us to later determine the shared locations written by \( P_{\text{curr}} \) in this interval, which we

*WRITE and READ subsume CREAT and D in our earlier work [RC97b].
(a) On every load to a shared location, set READ for the location to \( P_{\text{curr}} \).

(b) On every store

   i. If the location is in shared memory, set WRITE for the location to \( P_{\text{curr}} \).
   ii. Set the PROCINT field of the location to the merged result of the PROCINT fields of all its operands.
   iii. If the location is in shared memory, merge the PROCINT information of the line corresponding to the store with the values already there.

(c) On reaching a synchronization operation:

   i. Write to the log file
      - All locations in shared memory that have WRITE set to \( P_{\text{curr}} \), along with their PROCINT fields.
      - All locations in shared memory that have READ set to \( P_{\text{curr}} \).
      - All source lines marked WRITE, along with their PROCINT fields
   ii. For all shared locations, reset READ if it is set to \( P_{\text{curr}} \).
   iii. If a shared location has WRITE set to \( P_{\text{curr}} \)
      - Reset WRITE
      - Set PROCINT to \([P_{\text{curr}}, \text{current interval}]\).
   iv. If a source line has WRITE set, reset WRITE and PROCINT

**Figure 5.5** The SRCOPS run-time algorithm used for obtaining dependence information. \( P_{\text{curr}} \) is the process executing the algorithm.

need for conflict analysis. Locations in private memory are not marked as they cannot be involved in cross-process dependences. Although a private location cannot be read by other processes, it can be read by the writing process in a later interval. Consider a location \( x \) in private memory, that depends on shared data. When \( x \) is read in a later interval, we want to know the locations on which \( x \) depends. Hence, for private as well as shared writes, we keep track of the equivalence classes of the operands on which the write depends.

A single line in the source code may correspond to stores to different locations. Hence, in contrast to the stores, we merge the PROCINT information of the operands of the store, with the existing PROCINT information for the line.
3. For every interval, we need to know two pieces of information. First, we need the identity of shared locations written in the interval, along with the equivalence classes of their operands. Second, we need the set of shared locations read in the interval. While we need the location addresses, we do not need the order in which the locations were accessed. Since most programs exhibit some degree of temporal and spatial locality, this information can aggregated, and represented very compactly. This in turn enables the log files to which we write this information to grow slowly.

Accesses to private memory are not recorded. Since these locations can be read only by the writing process, we effectively short-circuit them when recording operand information.

Once this information has been written to the log files, we clear the state associated with the shared locations in preparation for the next interval. By clearing READ and WRITE, we are able to pinpoint the interval in which shared accesses occur. For shared locations written in an interval, we also set the PROCINT information. This is the equivalence class to which the write belongs, and is used by a subsequent write for which this data is a source operand. We also clear the PROCINT information associated with the source lines. Note that the existing synchronization in the program orders the read–write accesses to PROCINT and WRITE.

We do not clear the PROCINT information for private locations. This enables us to short-circuit these locations when recording operand information.

For both logging the state of shared locations and for clearing them, we need to scan the shared state information. We show how this can be done efficiently, without scanning all the shared state, in Section 5.8.

Relating Computation to Source Lines

Once we identify computation that can be postponed or relocated, we need to map these writes to source lines in the program. Similar to memory locations, we also maintain operand information for source code lines. The operand information of all the stores corresponding to a source line is merged and attributed to the line. Given a set of stores that need to be postponed or relocated, we first determine the PROCINT information corresponding to these stores. Let this be \( P \). Then, we find the source
lines for which $P$ is a subset of the PROCINT field. These are the lines corresponding to the stores.

### 5.6.3 Collecting Address Traces for Basic Blocks

We instrument the loads and stores in each basic block of the assembly code. The instrumentation we insert collects the address trace for the basic block in a statically allocated area. At the end of the basic block, we insert a call to a function (described in Section 5.6.5) that processes this trace completely. Hence, the space required for collecting the trace during execution is very small.

We reserve four registers from the application for the instrumentation, using the `-ffixed-register` option of the GNU C compiler. One register each is used for computing the access address and for holding the pointer to the location in which to store the address trace. We use another two registers to handle instructions that modify the condition codes. The hardware platform we use is SPARC v8 based. In this architecture, user-level code cannot read the integer condition code registers except through branch instructions. Hence, when we insert an instrumentation call between the setting of the condition code (say, by a compare instruction), and its subsequent use (say, by a branch instruction), we use these registers to temporarily store the operands of the instruction that sets the condition code. In the SPARC, only ALU instructions affect the condition code. Hence, just prior to the branch, we use the saved values to re-execute the instruction that set the condition codes, discarding the result. The instrumentation procedures do not contain any floating point code. Hence, we do not have to take similar measures when inserting calls between the setting of a floating point condition code and its subsequent use.

### 5.6.4 Converting Control to Data Dependences

During execution, control dependences must be converted to data dependences. To see why, consider the code segment shown in Figure 5.6. Here the write to $x$ depends not only on $y$ and $z$, but also on $v$. Moreover, all accesses made inside doit(), and those inside the functions doit() might call, depend on $v$ too.

The first step in converting control to data dependences is to compute the control flow graph (CFG) [CFR+91] of each function in the program. This is a directed graph with nodes representing basic blocks, and edges, the possible flow of control. We use the assembly output to compute this graph. Next, we use the graph to com-
Figure 5.6 A code fragment illustrating control dependences.

```plaintext
if (v == 0) {
    x = y + z
    t = u + x
    doit (t)
}
```

pute the *immediate postdominator* of each basic block in a function. The immediate postdominator is defined as follows [CFR+91]:

If X and Y are CFG nodes, and X appears on every path from Y to the exit node, X *postdominates* Y. If X postdominates Y, but X \( \neq Y \), then X *strictly postdominates* Y. The immediate postdominator of Y is the closest strict postdominator of Y on any path from Y to the exit node.

The exit node is the possibly artificial node through which all basic blocks exit a function.

We use the statically computed immediate postdominator information to determine the run-time scope of a *controlling* basic block. A controlling basic block is a basic block that can change the flow of control. Within this basic block, we refer to the instructions causing control-flow as the controlling instructions. A store between a controlling basic block B and its immediate postdominator depends on the accesses in B that cause the change in control flow. For instance, in Figure 5.9, stores in basic block F depend on the values accessed by the controlling instructions of basic block A. The stores in C and D depend on the values accessed by the controlling instructions of B as well.

At any point in the execution, we maintain the equivalence classes of the operands on which a store at that point will be control dependent. We use the term *ccd* (current control dependence) to refer to this information. The ccd needs to be updated only after executing a controlling basic block or its immediate postdominator. After executing a controlling basic block we first save the current ccd in some state associated with the immediate postdominator for that basic block. We then update the ccd with the operands of the controlling instruction in the just executed basic block. On reaching an immediate postdominator for any controlling basic block, we first restore the saved ccd. As multiple controlling basic blocks may have the same
1. If cinfo[ipostdom].flag is EMPTY
   • Save ccd in cinfo[ipostdom].ccd
   • Set cinfo[ipostdom].flag to FULL
2. Merge data dependences of controlling instruction with ccd

**Figure 5.7** Algorithm executed after a controlling basic block.

1. Restore ccd from cinfo[this basic block].ccd
2. Set cinfo[this basic block].flag to EMPTY

**Figure 5.8** Algorithm executed before the immediate postdominator of a controlling basic block.

 immediate postdominator. We save the ccd only if a previously executed controlling basic block has not already saved the ccd. We use a flag associated with the ccd to determine this.

Figures 5.7 and 5.8 show respectively, the pseudo-code for the instrumentation added at the end of controlling basic blocks, and at the beginning of their immediate postdominators. In both figures, cinfo refers to the area in memory where the ccd information is stored. ipostdom refers to the immediate postdominator. flag refers to the flag used to determine if the ccd information has already been set.

Figure 5.9 illustrates the working of these two algorithms with an example. Each node represents a basic block in the CFG and is annotated with the controlling instruction (in italics) and the operations carried out to convert the control dependence to a data dependence. Note that basic block $G$ is the immediate postdominator of both $A$ and $E$. Hence, if control reaches node $E$, it does not save the ccd in the state associated with $G$. 
Figure 5.9 A control flow graph illustrating the working of the algorithms in figures 5.7 and 5.8. ccd $\rightarrow$ ccd, p indicates that the dependence information of p is merged with ccd. Thick circles indicate the control flow.

Converting control to data dependences requires us to maintain the ccd through function calls. Thus a callee inherits the ccd at the call site. At the exit basic block, the ccd is restored to that in effect when the call was made.

Handling control dependences as described above allows us to compactly represent the ccd information. The maximum number of concurrently saved ccd’s in a function is bounded by the number of immediate postdominators for its controlling basic blocks. This number is independent of the run-time control flow behavior of the function. Each ccd takes a small amount of space for representing the [process, interval] pairs of the operands on which it depends. Thus, on entry to a function, we allocate the space needed for saving the ccd’s in a separate stack. This stack grows as the program makes calls, much like the data stack.
Our approach has some similarities with "if-conversion", developed by Allen et. al. for vectorizing loops with control flow [AKPW83]. The difference is that instead of explicitly converting control dependences into data dependences statically (as Allen et. al. do), we do the conversion dynamically, as required for a particular execution.

**Procedure Calls and Indirect Jumps**

The technique we have described above works even if the program makes calls through function pointers. We construct the control flow graphs on a procedure-by-procedure basis. Consequently, the presence of procedure calls whose identity is not known statically does not affect us. Our run-time processing also ensures that all stores in a procedure called via a function pointer depend on two entities: (i) the ced in effect at the time the procedure is called through the pointer, and (ii) the function pointer itself.

At present our scheme does not handle indirect jumps, since their presence precludes the construction of a control flow graph and the determination of the immediate postdominator information.

**5.6.5 Using Mirror Functions for the Run-time Processing**

A trace of the memory locations accessed by the program is not sufficient to determine the source operands of a store. For instance, in Figure 5.6, if the compiler re-orders memory accesses, the code for the two assignments before the call to doIt() could produce the following run-time trace:

```
load y, load z, load u, store x, store t
```

The source operands of a store can be determined as follows. A basic block specifies dependences between registers and memory. By scanning each basic block, we can determine the register-register, register-memory and memory-memory dependences that it specifies. We can use these dependences to determine the operand information for both stores in the basic block, and the registers that are live on exiting the basic block. This operand information is in terms of the loads in the basic block and the registers live on entering the basic block. Both registers and memory have no inter-dependences at the beginning of the program. Hence, by keeping track of the operand information as each basic block executes, we can determine the memory locations of the source operands for each store.
Here is one way to do this. After a basic block executes, we could interpret it to determine the register and memory inter–dependences. By combining this with the access trace produced during execution, we would have the exact memory locations involved in the dependences. These locations could then be used in step 2 of the SRCOPS algorithm in Figure 5.5 to obtain the source operand information in terms of equivalence classes. This process could be speeded up by annotating the basic blocks with the register and memory inter–dependences before execution.

As interpreting the basic block information can be slow, we take a different approach. After scanning the basic blocks in the assembly code, we use the register–register, register–memory and memory–memory dependences to generate a mirror function. The mirror takes the run–time address trace from the original basic block as its argument to determine the source operands. Now, as the program executes, we also need to execute SRCOPS. Therefore, inside the mirror function, we insert calls to functions that execute this algorithm.

The advantage of this approach is that it allows us to specialize the mirror function to perform only the actions required by a particular basic block. The disadvantage is an increase in the code size. We can address this problem by first profiling the code to find out the basic blocks that are executed most often. Tools such as pixie [Smi91] can be used to do this. Then, we can use the mirror functions only for the most frequently executed basic blocks, leaving the others to be handled by a run–time interpreter. We have not yet implemented this. Thus, in our implementation prototype, we generate and use mirror functions for all program basic blocks.

Our prototype currently generates the mirror code in the form of C functions, which we compile and link with the application program. At the end of each basic block, we insert a call to its mirror function, which then maintains the operand information for the basic block just executed. The mirror functions also set all assignments to registers and memory to depend on the current control dependence (ccd) information.

The SPARC architecture uses register windows, causing a function to get some new registers and share some of its registers with its caller (a caller’s output and a callee’s input registers are the same). Hence, like the current control dependence information, we keep the register information also in a stack that grows as the program makes function calls. Instrumentation that we insert at the entry and exit of every function ensures that the mirror functions operate on the proper register set.
Using mirror functions to carry out the run-time processing shares some similarities with abstract execution [Lar90], which expands a small set of events recorded at run-time into a full trace. However, while our mirror functions can be considered abstract versions of the original basic blocks, we execute them in conjunction with the original program to efficiently compute operand information.

Although we could have used an instrumentation system such as ATOM [SE94] for collecting the per-basic block address trace. or EEL [LS95] which also provides control flow information, we would need to extend them to convert control to data dependences and to examine the basic blocks to obtain the operand information. As we could not trivially use these existing instrumentation systems, we implemented all the analyses described here from scratch.

5.7 Perturbation

The instrumentation we discuss in this chapter is highly intrusive and could significantly distort the run-time behavior of the program. Here, we show how the effects of this perturbation can be reduced so that it has negligible impact on the performance debugging.

As our goal is to pinpoint sources of program inefficiency that are independent of the target architecture, we need only architecture independent information from the execution, such as the memory accesses and source operands of writes. Hence, we can first execute the program, tracing only the relative order in which synchronization operations take place. Only a small amount of perturbation is introduced when doing this. The recorded trace of synchronization events can then be used to “replay” the program, forcing the synchronization operations to take place in the same order. As long as the program is deterministic and free of data races, this results in the same access ordering as during the “record” phase. This technique and the perturbation it introduces has been researched by Roosse and Zwaenepoel [RZ97].

We have not yet implemented this technique. The applications we discuss in this dissertation use only barriers and locks. In these applications, the locks are used only to enforce mutual exclusion when computing reductions. Hence, for these applications, the instrumentation did not distort the synchronization order.
5.8 Run–time System

After instrumenting the assembled source program, we compile it and link with the mirror functions. Our run–time system is the TreadMarks software distributed shared memory system [KDCZ94]. TreadMarks provides a shared memory abstraction on distributed memory (message–passing) machines.

In our implementation, we limit the number of [process,interval] tuples used for recording operand information. When a location depends on more [process,interval] tuples than we can accommodate, we discard the oldest interval and keep the rest. For each location, we currently limit this number to four intervals per process. During the data analysis phase, we can therefore encounter a write for which the operand information has “overflowed”. In this case, we conservatively assume that the source operands for this write could have been last written in any interval preceding the last recorded interval.

As the parallel program executes, it generates log information. We compress this data on–the–fly using the Gnu gzip utility and write it to disk in the form of per–process log files. After the program has finished executing, we analyze these files using Rx.

Before we conclude this section, we describe two aspects of the implementation of the SRCOPS algorithm in Figure 5.5.

First, we mentioned there that for shared locations, PROCINT and WRITE are placed in shared memory. Actually, we place the state in private memory and maintain consistency ourselves. We do this because the state can be kept up-to-date with lesser overhead by snoop ing the TreadMarks software consistency actions. than by having TreadMarks maintain the consistency. For a hardware shared memory implementation of Rx, we would just keep the state in shared memory.

Second, in step 3 of SRCOPS, we have to scan the PROCINT, WRITE, and READ fields of the state, record some data into the log files, and then reset state information. We use the virtual memory protection mechanism to do this efficiently. At first, we protect all shared pages from read and write accesses. On the first access to the page, we record the page as having been accessed and unprotect the page. On reaching a synchronization point, we scan the state for only those pages that have been accessed. Before proceeding, we again protect these pages from all accesses. When the accesses demonstrate spatial locality, this method has a low overhead. TreadMarks already
performs these actions for maintaining coherence. Hence, we use TreadMarks’ internal data structures to determine the virtual memory pages that have been accessed.

5.9 Summary

A recurrent theme throughout this chapter is the need to compactly represent information collected at run-time. This stems from the necessity to avoid collecting and dealing with voluminous amounts of data. The fairly involved run-time processing that we use in this chapter is a direct consequence of this requirement. By judiciously deciding what kind of information to collect and by partially processing collected data at run-time, we have been able to keep the performance debugging process tractable.

To detect excess synchronization, we need to analyze shared accesses for conflicts. We do this by collecting the shared read and write sets in each interval of the program execution. By aggregating accesses, we can represent them compactly in the log files. To keep our performance debugging approach tractable, we need to partially process the collected data at run-time. We explain the static analysis and instrumentation required to support the run-time processing.

Our implementation is integrated with TreadMarks, a state-of-the-art software distributed shared memory system. The instrumentation we insert is quite intrusive, and could significantly perturb the execution of the program being performance tuned. We explain how to reduce the effects of this perturbation by using a “record and replay” approach.

Although this chapter addresses the pragmatic subject of implementation, it still makes a few interesting contributions. We present an efficient method for dynamically converting control dependences to data dependences. We also use a novel method for carrying out the run-time processing. As is done in most simulators, we could have interpreted the statically pre-computed information. Instead, we generate code which, when executed, directly carries out the desired run-time processing. This significantly speeds up execution. This technique can also be used to speed up program analysis tools such as ATOM [SE94], which instrument an application to collect run-time information.

In the next chapter, we describe how R$_X$ analyzes the gathered information to prescribe transformations that reduce synchronization and communication.
Chapter 6

Postprocessing and Data Analysis

6.1 Introduction

In the previous chapter, we described the implementation of $R_X$, and the run-time processing required to collect the information needed for performance debugging. In this chapter, we present the algorithms used by $R_X$ to process this information. These algorithms produce the source-level prescriptive transformations that are supplied to the programmer.

A central component of all the algorithms is a check for conflicting accesses. When reducing synchronization, the algorithms for detecting ns-atomic critical sections and excess pairwise synchronization may also require solving a dependence enforcement equation (see section 4.4.2). Finally, for restructuring computation, we make use of the source operand information, collected in terms of equivalence classes.

The first half of this chapter has, more-or-less, the same structure as Chapter 4. We apply the algorithms in the order shown below. The section where an algorithm is described appears within brackets.

1. Detect excess barriers, and barriers that can be weakened (6.2.1)
2. Detect excess pairwise synchronization (6.2.2)
3. Check whether accesses can be moved outside critical sections (6.2.3)
4. Check whether critical sections can be removed, aggregated, vectorized (6.3)
5. Check whether computation can be postponed (6.4.2)
6. Check whether computation can be relocated (6.4.1)

When we introduced our performance debugging approach in Chapter 3, we also described the drawback of a run-time approach. Specifically, as $R_X$ uses data gathered from a program execution, the transformations it determines may violate program correctness in another execution. Our solution to this problem uses metaknowledge
about the program and the run-time system. We describe this in the latter part of this chapter, in Section 6.5.

We conclude with a summary of this chapter.

6.2 Detecting Unnecessary Synchronization

6.2.1 Barriers

Figure 6.1 shows the algorithm used by $R_X$ to detect unnecessary or unnecessarily strong barriers. We consider each barrier in the execution, and check for conflicting accesses in the $b$-intervals preceding and following the barrier. The absence of a conflict between the accesses by processes $P_a$ and $P_b$ indicates that the synchronization from $P_a$ to $P_b$ in the barrier can be removed.

If the accesses in the intervals conflict, the barrier can still be removed if the conflict is due to anti- or output dependences. In this case, we inform the user of this possibility, suggesting the transformations described in Section 4.4.1. These include replacing the barrier with weaker synchronization, or removing it outright by applying the odd-even fix.

In all subsequent operations on the partial-order graph, we exclude excess barriers and replace unnecessarily strong barriers with weaker pairwise synchronization.

For each barrier in the program execution:

1. Identify the $b$-intervals preceding and following this barrier.

2. For each pair of processes $P_a$ and $P_b$, check for conflicts between accesses in the preceding $b$-interval on $P_a$, and the following $b$-interval on $P_b$.

3. If there are no conflicting accesses, synchronization from $P_a$ to $P_b$ can be removed from the barrier.

4. If there are conflicting accesses that are due to anti- or output dependences, the barrier can be weakened using the methods of Section 4.4.1.

Figure 6.1 $R_X$ algorithm to detect excess barriers.
6.2.2 Pairwise Synchronization

Figure 6.2 shows the algorithm used by $R_X$ to detect excess pairwise synchronization. The cross-process edges in the partial-order graph correspond to the synchronization order observed during the execution. Therefore, if they are not already present, we add the edges due to pairwise synchronization by joining each such release to the subsequent acquire that reads the write of the release. We derive this ordering from the information collected at run-time. Since our goal is to find the minimal set of pairwise synchronization from among those present in the program, we ignore all the other synchronization at this point. We refer to the resulting intervals in the augmented partial-order graph as mega-intervals. For each pair of conflicting mega-intervals $I_1$ and $I_2$, either $I_1 \xrightarrow{h} I_2$, or vice versa. We find all the paths that can order them. Then, we set up the “enforced-by” equation and solve it as described in Section 4.4.2.

In all subsequent operations on the partial-order graph, we only include the releases and acquires that are in the resultant set.

<table>
<thead>
<tr>
<th>For each b-interval in the program execution:</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Make explicit all the release-acquire edges in this b-interval of the partial-order graph.</td>
</tr>
<tr>
<td>2. Check each pair of mega-intervals for conflicting accesses.</td>
</tr>
<tr>
<td>3. For each pair of conflicting mega-intervals $I_1$ and $I_2$, where $I_1 \xrightarrow{h} I_2$, find all paths from $I_1$ to $I_2$ made up of only release-acquire edges.</td>
</tr>
<tr>
<td>4. Set up and solve the dependence enforcement equation as discussed in Section 4.4.2. This gives the minimal set of releases and acquires from among those present in the program.</td>
</tr>
</tbody>
</table>

**Figure 6.2** $R_X$ algorithm to detect excess pairwise synchronization. The term “mega-interval” is described in the text.
6.2.3 Critical Sections

All the work involved in identifying computation that can be moved outside critical sections is performed at run-time. As described in Section 5.4.2, in our prototype implementation, we require the programmer to ensure that the uniprocess control and data dependences are preserved when moving all computation not marked as MPU or MFL outside the critical section.

6.3 Transforming Synchronization

Figure 6.3 shows the algorithm used by $R_X$ to detect ns-atomic critical sections. We consider each b-interval in the execution. First, we add the edges joining the reduced set of releases and acquires detected in Section 6.2.2, and the critical section unlocks and locks. Next, we determine the intervals that are outside critical sections, and whose accesses conflict. For each pair of such intervals $I_1$ and $I_2$, either $I_1 \uparrow I_2$.

For each b-interval in the program execution:

1. Make explicit all the lock-unlock and release-acquire edges in this b-interval of the partial-order graph.

2. Check for conflicting accesses between intervals outside any critical sections.

3. For each pair of conflicting intervals $I_1$ and $I_2$, where $I_1 \uparrow I_2$, find all the paths from $I_1$ to $I_2$.

4. Set up the dependence enforcement equation as discussed in Section 4.5.3.

5. Now, set to TRUE all variables on the right hand side corresponding to

   - Releases or acquires (due to pairwise synchronization)
   - Critical sections that are not ns-atomic

6. Solve the equation as described in Section 4.5.3. This gives us the maximal set of ns-atomic critical sections.

Figure 6.3 $R_X$ algorithm to detect ns-atomic critical sections.
or vice versa. We find all the paths in the partial-order graph from the earlier to the later interval. Each release and unlock in a path can be a sender for the accesses in the earlier interval. Each acquire and lock in a path can be a receiver for the accesses in the later interval. We then set up the equation enforcing the requirement that every set of accesses that needs a receiver or sender, has one. Each term on the right hand side corresponds to the set of receivers or senders for the accesses in an interval. Some terms can be omitted because one or more of the constituent synchronization variables are required. These synchronization variables correspond to the releases and acquires due to pairwise synchronization that are found to be required in Section 6.2.2, and to the critical sections that are found to be not ns–atomic by the run–time processing in Section 5.5. The resulting equation is then solved.

6.3.1 Aggregating and Vectorizing Critical Sections

Critical sections identified as ns–atomic may be aggregated and vectorized. In the context of a parallelizing compiler, these transformations correspond to detecting and optimizing recurrences. Figure 4.5 provides an example of these transformations.

A formal specification of these two transformations is beyond the scope of this thesis. The difficulty arises from having to recognize when these transformations can be applied. For the present, we just point out ns–atomic critical sections and state that these may be transformed if the operations on locations inside the critical sections are commutative and associative. Since we determine the critical sections to be ns–atomic, the programmer need not be concerned about accesses outside the critical sections.

6.4 Restructuring Computation

6.4.1 Postponing Computation

Figure 6.4 shows the algorithm used by $R_X$ to determine whether synchronization can be removed by postponing computation. We first use the equivalence class information recorded at run–time to determine computation on process $P_a$ for which the source operands were created on $P_i$ in the previous $b$–interval. Next, if this computation uses operands created in $(B0, B1)$ on another process $P_i$, we intersect the reads by $P_a$ in $(B1, B2)$ with the writes by $P_i$ in $(B0, B1)$ to determine a superset of the source operands for the computation. We then check for two conditions. If the results
Identify a set of four adjacent barriers in the execution: $B0$, $B1$, $B2$, and $B3$. Then, for each pair of processes $P_a$ and $P_b$:

1. Determine the computation on $P_a$ in $(B1, B2)$ for which the source operands were created on $P_b$ in $(B0, B1)$. Let $W$ be the set of shared locations written by the computation.

2. Determine the superset of source operands for this computation that was created in $(B0, B1)$ on each process. Let $O_i$ denote those created on $P_i$.

3. If both the following conditions are true, the computation involving $W$ can be postponed to $(B2, B3)$, and the synchronization from $P_b$ to $P_a$ removed.
   - No elements of $W$ are accessed in $(B2, B3)$ on any process.
   - No elements of $O_i$ are written in $(B1, B3)$ on any process.

**Figure 6.4** $R_X$ algorithm to determine whether computation can be postponed.

Identify a set of four adjacent barriers in the execution: $B0$, $B1$, $B2$, and $B3$. Then, for each pair of processes $P_a$ and $P_b$:

1. Determine the computation on $P_a$ in $(B1, B2)$ for which the source operands were created on $P_b$ in $(B0, B1)$. Let $W$ be the set of shared locations written by the computation.

2. Determine the superset of source operands for this computation that was created in $(B0, B1)$ on each process. Let $O_i$ denote those created on $P_i$.

3. From the set $W$, determine the locations read by each process in $(B2, B3)$. Let $R_i$ denote those read on $P_i$.

4. Find the process $P_b$ for which $|O_i| + |R_i|$ is maximum. If this is not $P_a$, the computation should be performed on $P_b$.

**Figure 6.5** $R_X$ algorithm to determine whether computation should be relocated.
of the computation are read or written in the next \( b \)-interval by any process, or if the source operands are modified in the current or next \( b \)-interval by any process, the computation cannot be postponed. Otherwise, the computation can be postponed, and the synchronization between \( P_b \) and \( P_a \) weakened or removed.

### 6.4.2 Relocating Computation

Figure 6.5 shows the algorithm used by \( R_X \) to determine whether computation should be relocated. We first use the equivalence class information recorded at run-time to determine computation on process \( P_a \) for which the source operands were created on \( P_b \) in the previous \( b \)-interval. Next, if this computation uses operands created in \((B0, B1)\) on process \( P_i \), we intersect the reads by \( P_a \) in \((B1, B2)\) with the writes by \( P_i \) in \((B0, B1)\) to determine a superset of the source operands. We also intersect the set of locations written by the computation on \( P_a \) (i.e., \( W \)) with the accesses by each process \( P_i \) in \((B2, B3)\) to determine how much of \( W \) is used on each process. We then add these two quantities together on a per-process basis. If \( P_j \) is the process for which the sum is maximum, the computation should ideally be performed on \( P_j \) in order to reduce data communication.

However, before suggesting that the computation be relocated, we check if the load balance will be disrupted. Currently, we use the amount of shared data written in the interval as an indicator of the time taken to perform computation. Let \( C_i \) be the shared data written by \( P_i \) in \((B1, B2)\). Let \( R_X \) detect that computation performed on \( P_a \) should be performed on \( P_b \). Let \(|C|\) denote \((|C_a| + |C_b|)/2\). Then, we check if \(|C_a| - |W|\) is closer to \(|C|\) than \(|C_a|\). If it is not, the load balance will likely be skewed by relocating the computation. In either case, we leave the ultimate decision of performing the relocation to the programmer.

### 6.4.3 Limitations

Our approach for restructuring computation is limited by the fact that we can determine only a superset of the memory locations that make up the source operands of a write. Consider an example where \( x \) and \( y \) are written by \( P_a \) in \((B1, B2)\). Let \( x \) depend on \( u \), and \( y \) on \( v \), where \( u \) and \( v \) are written by \( P_c \) in \((B0, B1)\). At run-time, the source operands for these writes are recorded as

\[
[P_c, (B0, B1)] \xrightarrow{\text{src ops}} x \quad \text{and} \quad [P_c, (B0, B1)] \xrightarrow{\text{src ops}} y
\]
During analysis, we can only determine that the superset of locations which are the source operands for \( x \) and \( y \) are \( u \) and \( v \).

This limitation affects us in step 2 of both algorithms. After determining the computation on \( P_a \) that uses source operands supplied by \( P_b \), we use the equivalence class information for the writes in \( W \) to determine where and when the other source operands for these writes were created. Let some of the source operands have been created by \( P_c \) in \( (B0, B1) \). Then, to determine the identity of these operands, we intersect the reads by \( P_a \) in \( (B1, B2) \) with the writes by \( P_c \) in \( (B0, B1) \). However, by doing so, we may include locations that provide operands for some computation that does not produce any element of \( W \).

6.5 Extending the Scope of Transformations

In Section 3.5, we briefly mentioned how we extend the scope of the debugger findings that are based on one execution to all possible executions of the program. We explain our approach in greater detail here.

What are the conditions under which a transformation suggested by \( R_X \) will be correct for all executions? In order to obtain a transformation, \( R_X \) analyzes shared accesses for conflicts and determines the source operands of computation. If the results of these analyses do not change when applied to a new execution, the transformation that is obtained is also applicable.

6.5.1 Eliminating and Transforming Excess Synchronization

Two conditions determine whether a transformation to eliminate excess synchronization applies to a new execution. Excess synchronization is detected by determining whether the accesses in two intervals conflict. Hence, the first condition is that the endpoints of the two intervals be the same for a new execution (i.e., the same synchronization operations delineate the intervals for a new execution). The second condition is just that the shared accesses in the two intervals do not conflict.

We use static analysis to identify potentially conflicting accesses. Given that shared data must be heap allocated in our system, we can just identify all pointer accesses as potentially touching shared data. All conflicting accesses must be made through these pointers. The static analysis to identify pointer accesses is trivial, unlike generic pointer analysis which sets out to find what data can be accessed through a pointer.
In the presence of a system-wide invariant that constrains shared memory allocation, we can further simplify the second condition. Most run-time environments constrain the allocation of shared data. For instance, stack data is generally considered private for Posix threads.* In our run-time environment, all shared memory is allocated from the heap. Hence, for C programs, in our environment, only accesses made through pointers can conflict. We use this metaknowledge to prune the set of pointers through which all conflicting accesses must occur. For instance, if only one process can execute a section of code (as in Figure 3.4), we mark the accesses that can be made from within that section appropriately. Existing static dependence analysis methods could also be used to prune the pointer set, but only when they can precisely determine the access set. We present the remaining pointer accesses to the user, who has to check them for conflicts. In the future, we plan to use alias analysis to further prune this set.

The prerequisite for aggregating and vectorizing critical sections is that they be ns-atomic. Conflict analysis decides whether the the first two conditions from Figure 4.7 are true for a critical section. Hence, we use the methods described above to verify that the ns-atomicity will hold in other executions. The third condition requires that no read outside a critical section depend on data written inside a critical section, which in turn depends on data written on another process. We ask the user to verify this condition for the critical sections that we determine to be ns-atomic.

6.5.2 Restructuring Computation

R_k has to carry out both conflict and source operand analysis to determine if computation can be postponed or relocated. To restate the main point of Section 4.6.1, the following questions must be answered before synchronization can be removed by postponing computation.

1. Does the postponed computation produce results read in the next b-interval?

2. Are the source operands for the postponed computation modified in the current or next b-interval?

3. After postponing the computation, will there still be conflicting accesses?

*Although Posix threads share the entire address space, any sane program can be expected to maintain a distinct stack for each thread.
The answers to the first and third questions depend on conflict analysis. The correctness conditions are hence the same as in the excess synchronization case. The correctness condition for the second question is just that the source operands of the computation identified for postponement not be modified in the current and next b-interval.

6.6 Summary

We described two aspects of $R_X$ in this chapter. First, we outlined the algorithms that are used to process information gathered at run-time. Then, we described our approach to verify that the transformations we suggest will hold for all program executions. We do this by using metaknowledge in the form of system and programmer supplied invariants. When these are not enough to let $R_X$ infer the correctness of the transformations, we make a contract with the programmer. If the programmer guarantees that certain conditions will hold for all executions, $R_X$ guarantees that the transformations it prescribes can be correctly applied.

In the next chapter, we demonstrate the viability of our approach by describing how we have used $R_X$ to performance tune a wide variety of applications.
Chapter 7

Results

7.1 Introduction

This chapter demonstrates the viability of our performance debugging approach. We describe the transformations provided by $R_X$ when we applied it to a suite of six applications. We then present the performance improvements obtained by incorporating the debugger feedback back into the applications. This provides a sense of the "usefulness" of $R_X$. We also show the viability of our approach through a detailed examination of the overheads imposed while performance debugging these applications.

Our application suite includes programs from a wide variety of sources. Barnes–Hut, Water, and Gauss are benchmark programs. The first two are from the SPLASH-2 suite [WOT+95], while the third is locally derived. Conjugate Gradient and Multigrid were written by first year graduate students in a parallel programming course, using specifications from the NAS benchmarking document [BBL93]. Shallow, also based on the NAS specifications, was provided to us by a source from industry, with a specific request to improve its performance. The applications range from fine-grained programs using locks, to coarse-grained programs that use only barriers. Furthermore, Barnes–Hut, Conjugate Gradient, and Multigrid make extensive use of pointers.

$R_X$ detected excess synchronization in five applications. For Shallow, $R_X$ suggested computation restructurings that significantly reduced data communication.

We used two shared memory platforms to study the performance impact of the transformations prescribed by $R_X$. Our hardware shared memory environment consists of an 8–processor Sun Enterprise server on which we use the Posix threads (pthreads) package. The software shared memory platform consists of an 8–processor IBM SP2 machine running TreadMarks [KDC94].

We first discuss the impact of $R_X$ feedback when the optimized applications were executed on the hardware shared memory system. For Gauss, the debugger findings had no impact on performance. $R_X$ feedback slightly improved the performance of
Shallow and Multigrid. At 8 processors, the optimized versions of both programs ran 1.02 times faster than the original versions. For Water, $R_X$ feedback resulted in the optimized program running 1.07 times faster than the original version. For Conjugate Gradient, at 8 processors, feedback from $R_X$ made the optimized program run 1.28 times faster. On this platform, the synchronization transformation that $R_X$ suggested for Barnes–Hut is effectively the same as the original synchronization.

$R_X$ feedback had much more impact on the software shared memory platform. For Gauss, the debugger feedback had very little impact on performance. For Multigrid, Barnes–Hut and Conjugate Gradient, at 8 processors, the optimized program ran 1.13, 1.18 and 1.43 times faster (respectively) compared to the original program. $R_X$ feedback improved the performance of Water and Shallow substantially. Optimized Water ran 10 times faster, while Shallow ran 66 times faster.

We evaluated the overheads of performance debugging using $R_X$ by comparing the execution of the instrumented and uninstrumented applications. We used three metrics in this evaluation: the execution time dilation, the memory overhead, and the size of the log files produced during execution. $R_X$ caused the instrumented applications to slow down by between factors of 7.5 and 108. Storing the state required for the performance tuning imposed a memory overhead of a factor of 40. The log files produced during the execution were all under 5.5 Mbytes in size.

The chapter outline is as follows. First, we describe the application suite in Section 7.2. Next, in Section 7.3, we explain the feedback provided by $R_X$ for each of the six applications. We describe the performance impact of the feedback on both hardware and software platforms in Section 7.4. We describe the costs of performance debugging, and how these can be brought down, in Section 7.5. We conclude with the chapter summary in Section 7.6.

### 7.2 Gross Application Characteristics

Table 7.1 summarizes the applications we used in this study: Water, Shallow, Conjugate Gradient, Multigrid, Barnes–Hut and Gauss. We chose this particular set as it demonstrates the different kinds of feedback that $R_X$ can provide. The applications vary widely in complexity. Shallow and Gauss are fairly simple, and do not use pointers except for dynamically allocating data structures. Thus, these two applications can arguably be performance tuned using static analysis methods. The remaining four applications do use pointers. Water uses a few, while Barnes–Hut,
<table>
<thead>
<tr>
<th>Application</th>
<th>Synchronization</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes–Hut</td>
<td>Barriers, Locks</td>
<td>SPLASH–2</td>
</tr>
<tr>
<td>Conjugate Gradient</td>
<td>Barriers, Locks</td>
<td>Course Project</td>
</tr>
<tr>
<td>Gauss</td>
<td>Barriers</td>
<td>Local benchmark</td>
</tr>
<tr>
<td>Multigrid</td>
<td>Barriers</td>
<td>Course Project</td>
</tr>
<tr>
<td>Shallow</td>
<td>Barriers</td>
<td>Industry</td>
</tr>
<tr>
<td>Water</td>
<td>Barriers, Locks</td>
<td>SPLASH–1</td>
</tr>
</tbody>
</table>

**Table 7.1** Application Summary

Conjugate Gradient and Multigrid use pointers extensively. All programs were written in C.

The sources of these applications also differ. Two of the applications, Barnes–Hut and Water, are benchmark programs. Multigrid and Conjugate Gradient were written by first and second-year graduate students as part of a course project. As such, they make excellent test cases for our performance debugger, because they represent programs written under time pressure by inexperienced parallel programmers. Shallow was given to us by a source from industry with a specific request to determine the reason for its poor performance. Gauss is a locally derived benchmark.

### 7.3 Excess Synchronization Detected

We applied our performance debugging approach to the applications listed in the previous section. Table 7.2 summarizes the $R_X$ findings. The first two columns list the application and the data set. The third column lists the kind of information supplied by $R_X$: whether it detected excess barriers and locks, and whether it was able to suggest computation restructurings to reduce synchronization and/or communication. The last column lists the reduction in synchronization when the application was run on 8 processors with the specified data set, after incorporating the $R_X$ feedback. Synchronization reported in this column is in terms of release–acquire edges. We model an 8-process barrier as requiring 14 release–acquire edges, and a lock acquire as one edge. Flags may require from 1 to 7 edges, depending on the number of processes waiting on the flag.

We describe the $R_X$ findings for each application in greater detail below.
<table>
<thead>
<tr>
<th>Application</th>
<th>Data Set</th>
<th>$R_X$ Findings</th>
<th>Dynamic reduction in sync (at 8 procs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>32768 bodies, 5 iters</td>
<td>✓</td>
<td>460 → 364</td>
</tr>
<tr>
<td>CG</td>
<td>LARGE</td>
<td>✓</td>
<td>42928 → 16618</td>
</tr>
<tr>
<td>Gauss</td>
<td>2000 × 2000</td>
<td>✓</td>
<td>28042 → 14042</td>
</tr>
<tr>
<td>Multigrid</td>
<td>128 × 128, 20 iters</td>
<td>✓</td>
<td>13524 → 5598</td>
</tr>
<tr>
<td>Shallow</td>
<td>800 × 800, 20 iters</td>
<td>✓</td>
<td>No change</td>
</tr>
<tr>
<td>Water</td>
<td>1728 molecules, 5 iters</td>
<td>✓</td>
<td>10132124 → 45684</td>
</tr>
</tbody>
</table>

Table 7.2 $R_X$ findings

7.3.1 Barnes–Hut

Barnes–Hut, from the SPLASH-2 suite [WOT+95], is a simulation of a system of bodies influenced by gravitational forces. A body is represented as a point mass that exerts forces on all other bodies. The algorithm uses a hierarchical oct-tree representation of space in three dimensions. Space is broken into cells. The internal nodes of the oct-tree represent the cells, and the leaves represent the bodies in the corresponding cells. Each time step consists of the following key phases: a process traverses the tree to obtain a set of bodies that results in good load balance between processes; it then computes the forces on these bodies; and finally computes the new positions of the bodies. We refer to these phases as the load balancing phase, the force computation phase, and the position computation phase. Barriers separate the different phases.

$R_X$ was able to detect three instances of excess synchronization in this program.

Barnes–Hut uses a few critical sections in the force computation phase to update global variables ($n2bcalc, nbccalc, keten, etc.$), that are subsequently read at the beginning of the position computation phase by process 0, to print out the progress made by the application. $R_X$ pointed out that these critical sections could be vectorized. $R_X$ also pointed out that the printing of the information could be postponed to after the position computation phase. As a result, the barrier separating the force and position computation phases is necessary only to enforce an anti-dependence, and can therefore be weakened to a control barrier (Section 4.4.1).
Barnes–Hut also uses critical sections at the end of the position computation phase to determine the smallest cube within which all the bodies lie. $R_X$ pointed out that these critical sections, which update the global variables $\text{min}[]$ and $\text{max}[]$, could also be vectorized.

$R_X$ also identified a barrier that is used only for anti-dependences and pointed out that it could be weakened to a control barrier.

### 7.3.2 Conjugate Gradient

This application implements a conjugate gradient method to compute an approximation to the largest eigenvalue of a symmetric positive definite sparse matrix. The computation is typical of many unstructured grid applications. The main data structures are a set of vectors. Computation partitioning is achieved by splitting each vector equally among the processes. There are two key computation steps that are repeated several times during each iteration. The dot–product step involves computing the dot product of a vector. A critical section is used to accumulate the values from each process. In the sparse-multiply step, a sparse matrix is multiplied by a vector. Each process computes a portion of the result vector, for which it needs the entire input vector. These two steps are used in different phases of the program. The phases are separated by barriers.

This application was developed using the NAS parallel benchmark specifications [BBLS93], as part of a Rice Comp520 course project [LNW94].

$R_X$ detected three instances of excess synchronization in this program.

First, in the dot–product step, it detected an unnecessary critical section executed only by process 0 to clear the location where the dot product would be accumulated. In addition, it also suggested that the critical section used for accumulating the dot product be vectorized.

Second, in the sparse-multiply step, $R_X$ detected computation that could be moved outside a critical section. $R_X$ also pointed out that this was a non-synchronizing atomic (ns-atomic) critical section (see Section 4.5). Since the critical section is ns-atomic, and envelops no computation, it can be removed. In the original program, the processes use this critical section to copy their portions of the input vector to a global vector, which all the processes then read. The critical section is not needed because each process copies over distinct portions of the input vector.
Finally, $R_X$ detected a barrier that was used only to enforce an anti-dependence. This anti-dependence is caused by the accesses to the variable into which the dot product is accumulated. The barrier can be removed by applying the odd–even solution (Section 4.4.1) using two variables to accumulate the dot product.

### 7.3.3 Gauss

Gauss is an implementation of Gaussian elimination with partial pivoting, to solve a set of linear equations. The array is partitioned cyclically among the processes in order to improve load balance. At each iteration, one process computes the elements of the pivot row. The other processes read these elements to compute the values of all remaining rows in the array. A barrier is used to ensure that the processes read the pivot elements only after they have been computed by the pivot process. This program is a locally developed benchmark.

Gauss uses a barrier to ensure that the processes will read the pivot row only after it has been computed. $R_X$ detected this producer–consumer relationship, finding the barrier to introduce synchronization stronger than necessary. It suggested that the barrier be replaced with a flag. In each iteration, the pivot process sets this flag, signaling to the other processes that the pivot row has been computed.

### 7.3.4 Multigrid

This application implements a multigrid algorithm in order to obtain an approximate solution, $u$, to the discrete Poisson problem $\nabla^2 u = v$, on a cubic grid with periodic boundary conditions. $v$ is initially set to 0, except at twenty randomly selected points. The iterative solution begins with $u = 0$, and in each of the iterations several operators are applied that lead to a more precise approximation. A key phase of this program, which we refer to as global-communicate, is invoked multiple times in each iteration. During this phase, the boundary elements of the grids are copied to facilitate sharing between the processes. Synchronization is achieved within this phase using barriers.

This application was developed using the NAS parallel benchmark specifications [EBLS93], as part of a Rice Comp520 course project [LNW94].

$R_X$ detected two instances of excess synchronization in this program.

First, it found a barrier that does not enforce any dependence, permitting it to be removed. In addition, it found two barriers invoked in the global-communicate phase to be stronger than necessary. The first of these only enforces true dependences
between the first and the last processes. This is caused by these processes reading the edges of an array that "wraps around". This barrier can therefore be replaced by two flags that synchronize only these processes.

The second barrier enforces dependences with a more complicated pattern. Depending on where the global-communicate phase is invoked from, this barrier enforces only dependences between adjacent processes. Again, this barrier can be replaced by a flag that synchronizes only the relevant processes.

7.3.5 Shallow

Shallow is an implementation of the shallow water benchmark from the National Center for Atmospheric Research [Sad75]. This code solves a set of difference equations on a two-dimensional grid for the purpose of weather prediction. The main data structures are a set of arrays, with each process being assigned the computation on a band of rows. Barriers are used to synchronize the processes between the different phases.

This application was given to us by a source from industry with a request to improve its performance on TreadMarks.

No excess synchronization was detected in this program. However, R_{X} detected that the program had two places where restructuring the computation could reduce communication.

In the original program, the program data structures are initialized by the last process. During this phase, R_{X} detected a significant amount of data being communicated from process 0 to the last process. This is primarily an artifact of the TreadMarks version used by the instrumented application. In this version, at program startup, process 0 "owns" the entire address space. Consequently, when the last process accesses the data structures, the entire address space is effectively shipped to it. By relocating the initialization to process 0, this communication can be avoided. This change does not affect the running time of the parallel portion of the program.

At periodic intervals during the parallel execution, all four edges of the arrays are exchanged. We refer to this as the periodic continuation phase. In the original program, process 0 was doing all of the data exchanges. R_{X} pointed out that the exchanges between the first and last columns on a particular set of rows should be relocated to the process to which that band of rows was assigned. This relocation also improves the load balance of the program.
7.3.6 Water

Water, from the original SPLASH suite [SWG91], is a molecular dynamics simulation. The molecules are divided equally among the processes. There are two key phases in each timestep. In the first phase, called the force computation phase, a process updates the forces due to the interaction of its molecules with those of half of the other processes. The force computation requires reading the displacement vectors of the interacting molecules, which are calculated in the previous timestep. As each force value is updated by multiple processes, a critical section is used to protect the updates to a molecule record. In the second phase, called the displacement computation phase, a process updates the displacements of its molecules based on the forces calculated in the previous phase. Barriers separate these and other phases.

$R_X$ detected that the barrier at the end of the main loop in $\text{MDMAIN}(\cdot)$ enforces only anti-dependences. The odd-even fix can be applied here to remove the anti-dependence and the barrier. In addition, $R_X$ also suggested postponing some computation in the main loop, after the kinetic energy is computed, enabling another barrier to be removed.

The most important change suggested by $R_X$ (in terms of the amount of synchronization it reduced) was in the force computation phase. Here, $R_X$ prescribed that the critical sections used to update the forces be aggregated by accumulating the updates locally, and applying them once per molecule. The original program uses a large number of critical sections ($O(n^2)$ where $n$ is the number of molecules). Aggregation significantly reduces this number." $R_X$ also pointed out that some other updates (to $*\text{VIR. SUM}[]$, $*\text{POTA}$, $*\text{POTR}$, and $*\text{POTF}$) could be vectorized.

7.4 Performance Impact

To measure the performance impact of the debugger feedback, we incorporated the $R_X$ transformations into the original program, producing an optimized version. We then executed the original and optimized versions on both a hardware and a software shared memory platform. Both versions used the data sets described in Table 7.2. The data sets fit into the main memory of both systems.

*This change appears in the SPLASH-2 versions of Water [WOT+95].
7.4.1 Hardware Shared Memory

The hardware shared memory platform is an 8-processor Sun Enterprise server, with 248 MHz UltraSparc processors and 1 Gbyte of memory. We used Posix threads (pthreads) to create and synchronize threads. Basic synchronization costs on this platform are given in Table 7.3.

We report on three measurements using locks. The no contention lock measures the cost of an empty critical section. This is implemented as a lock acquire, immediately followed by a release. The contention locks measure the cost of acquiring a lock when several threads contend for the same lock. Finally, we report the cost of incrementing one word of memory while holding the critical section.

As pthreads does not support barriers, we implemented them using the mutual exclusion primitives that are provided.

We implemented flags from scratch, using memory reads and writes, and the SPARC membar instruction. We could have used semaphores, but the operating system on this platform, Solaris 2.5, does not support Posix semaphores. While the Solaris threads library does support semaphores, that implementation appears to be very slow. For instance, signaling 7 processes with a semaphore-based flag (using sema_post() and sema_wait()) is more than 1.5 times slower than using an 8-process barrier.

Table 7.4 summarizes the performance of the applications on this platform. The second column lists the sequential running time, devoid of any calls to the pthreads library. Columns 4 through 10 give the running times on 2 through 8 processors.

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>No contention locks</td>
<td></td>
</tr>
<tr>
<td>Contention locks (no work)</td>
<td></td>
</tr>
<tr>
<td>Contention locks (with work)</td>
<td></td>
</tr>
<tr>
<td>Flags</td>
<td>0.6</td>
</tr>
<tr>
<td>Barrier</td>
<td>1.8</td>
</tr>
</tbody>
</table>

Table 7.3 Basic operation costs on the Sun 8-processor machine. All times are in microseconds. The cycle time is 4.03 ns.
<table>
<thead>
<tr>
<th>App</th>
<th>Seq</th>
<th>Ver.</th>
<th>Processes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>2</td>
</tr>
<tr>
<td>CG</td>
<td>122.80</td>
<td>Orig</td>
<td>59.61</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Opt</td>
<td>59.81</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Factor</td>
<td>1.00</td>
</tr>
<tr>
<td>Gauss</td>
<td>172.33</td>
<td>Orig</td>
<td>87.25</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Opt</td>
<td>87.25</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Factor</td>
<td>1.00</td>
</tr>
<tr>
<td>MG</td>
<td>71.70</td>
<td>Orig</td>
<td>36.83</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Opt</td>
<td>37.05</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Factor</td>
<td>0.99</td>
</tr>
<tr>
<td>Shallow</td>
<td>37.59</td>
<td>Orig</td>
<td>18.12</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Opt</td>
<td>18.09</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Factor</td>
<td>1.00</td>
</tr>
<tr>
<td>Water</td>
<td>223.61</td>
<td>Orig</td>
<td>111.63</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Opt</td>
<td>106.67</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Factor</td>
<td>1.05</td>
</tr>
</tbody>
</table>

Table 7.4 Application performance on the Sun. All times are in seconds.

For each application, we also list the factor by which the optimized program is faster compared to the original program.

No results are reported for Barnes–Hut, as converting a barrier to a control barrier (Section 4.4.1) has no effect on this platform.\(^1\) A control barrier requires only the reads preceding the barrier to be performed before any following operations can be performed. In contrast, a barrier requires both reads and writes preceding it to be performed before any operation following it can be performed. The advantage of a control barrier is hence negligible (if not lost) on this platform.

The improvement in Conjugate Gradient is mainly due to the removal of the ns-atomic critical section in the sparse–multiply step. Removing this critical section eliminates the serialization of the execution of this part of the program.

---

\(^1\)A control barrier can be implemented using the `membar` instructions provided by the SPARC v9 instruction set [SPA94].
Reducing synchronization in Gauss and Multigrid has little impact on the performance because of the small cost of synchronization on this platform. Compared to the running time, both programs have so few barriers that converting them to flags has negligible effect on the performance. For instance, at 8 processes, the 2000 barriers in Gauss take up just 0.25% of the running time (i.e., 0.05 seconds).

No excess synchronization was found in Shallow. The (slight) improvement in performance is due to the increase in data locality and the better load balance caused by distributing the the computation in the periodic continuation phase among the processes.

The performance improvement in Water comes about because of the large number of critical sections that are eliminated. In spite of locks being cheap on this platform, the reduction is large enough for the performance to improve noticeably.

### 7.4.2 Software Shared Memory System

<table>
<thead>
<tr>
<th>Primitive</th>
<th>Processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2</td>
</tr>
<tr>
<td>No contention locks</td>
<td>7</td>
</tr>
<tr>
<td>Contention locks (no work)</td>
<td>160</td>
</tr>
<tr>
<td>Contention locks (with work)</td>
<td>1120</td>
</tr>
<tr>
<td>Flags</td>
<td>same as barrier</td>
</tr>
<tr>
<td>Barrier</td>
<td>167</td>
</tr>
</tbody>
</table>

**Table 7.5** Basic operation costs on the SP2 8-processor machine. All times are in microseconds. The cycle time is 4.03 nS.

The software shared memory platform is an IBM SP2 machine with eight 66 Mhz RS/6000 processors (thin nodes), having 128 Mbytes of memory per processor. We used TreadMarks version 1.1.0 with the IBM Message Passing Library (MPL) as our software distributed shared memory (DSM) system. As base TreadMarks does not provide flags, we implemented them. Basic synchronization costs on this platform are given in Table 7.5. The cost of a barrier and a flag operation that involve the same number of processors is the same. This is because our implementation of flags sends the same number of messages as barriers. The improvements from using flags come about by reducing the number of processors that need to be involved.
<table>
<thead>
<tr>
<th>App</th>
<th>Seq</th>
<th>Ver.</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>143.07</td>
<td>Orig</td>
<td>76.51</td>
<td>53.18</td>
<td>42.58</td>
<td>36.09</td>
<td>32.26</td>
<td>29.12</td>
<td>27.37</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Opt</td>
<td>74.90</td>
<td>51.19</td>
<td>39.81</td>
<td>32.79</td>
<td>28.58</td>
<td>25.29</td>
<td>23.21</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Factor</td>
<td>1.02</td>
<td>1.04</td>
<td>1.07</td>
<td>1.10</td>
<td>1.13</td>
<td>1.15</td>
<td>1.18</td>
</tr>
<tr>
<td>CG</td>
<td>238.79</td>
<td>Orig</td>
<td>85.80</td>
<td>54.27</td>
<td>46.10</td>
<td>43.11</td>
<td>42.59</td>
<td>42.83</td>
<td>42.57</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Opt</td>
<td>81.57</td>
<td>49.42</td>
<td>40.07</td>
<td>35.61</td>
<td>32.54</td>
<td>30.90</td>
<td>29.84</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Factor</td>
<td>1.05</td>
<td>1.10</td>
<td>1.15</td>
<td>1.21</td>
<td>1.31</td>
<td>1.39</td>
<td>1.43</td>
</tr>
<tr>
<td>Gauss</td>
<td>217.65</td>
<td>Orig</td>
<td>121.39</td>
<td>84.63</td>
<td>67.03</td>
<td>56.71</td>
<td>50.47</td>
<td>46.23</td>
<td>43.53</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Opt</td>
<td>121.81</td>
<td>85.28</td>
<td>67.63</td>
<td>57.18</td>
<td>50.48</td>
<td>45.84</td>
<td>42.34</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Factor</td>
<td>1.00</td>
<td>0.99</td>
<td>0.99</td>
<td>0.99</td>
<td>1.00</td>
<td>1.01</td>
<td>1.03</td>
</tr>
<tr>
<td>MG</td>
<td>80.04</td>
<td>Orig</td>
<td>66.61</td>
<td>–</td>
<td>45.14</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>30.66</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Opt</td>
<td>65.99</td>
<td>–</td>
<td>40.55</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>27.04</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Factor</td>
<td>1.01</td>
<td>–</td>
<td>1.11</td>
<td>–</td>
<td>–</td>
<td>–</td>
<td>1.13</td>
</tr>
<tr>
<td>Shallow</td>
<td>35.60</td>
<td>Orig</td>
<td>441.25</td>
<td>415.45</td>
<td>400.59</td>
<td>403.66</td>
<td>397.48</td>
<td>394.54</td>
<td>401.79</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Opt</td>
<td>25.47</td>
<td>16.07</td>
<td>10.70</td>
<td>8.81</td>
<td>7.70</td>
<td>6.73</td>
<td>6.04</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Factor</td>
<td>17.32</td>
<td>25.85</td>
<td>37.44</td>
<td>45.82</td>
<td>51.62</td>
<td>58.62</td>
<td>66.52</td>
</tr>
<tr>
<td>Water</td>
<td>390.96</td>
<td>Orig</td>
<td>255.92</td>
<td>639.99</td>
<td>765.95</td>
<td>686.94</td>
<td>707.36</td>
<td>618.01</td>
<td>578.23</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Opt</td>
<td>205.17</td>
<td>140.40</td>
<td>107.38</td>
<td>87.65</td>
<td>73.99</td>
<td>65.09</td>
<td>57.20</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Factor</td>
<td>1.25</td>
<td>4.56</td>
<td>7.13</td>
<td>7.84</td>
<td>9.56</td>
<td>9.49</td>
<td>10.11</td>
</tr>
</tbody>
</table>

**Table 7.6** Application performance on the SP2. All times are in seconds.

Table 7.6 summarizes the performance of the applications on this platform. The second column lists the sequential running time, devoid of any calls to the TreadMarks library. Columns 4 through 10 give the running times on 2 through 8 processors. For each application, we also list the factor by which the optimized program is faster compared to the original program. In contrast to the hardware platform, reducing synchronization has a fairly significant impact on the performance. We discuss the reasons for this below.

The cost of making the shared address space consistent is a significant factor affecting performance on this platform. The performance improvements in Barnes-Hut and Shallow can be attributed to this.
In Barnes–Hut, the bodies "owned" by a process during an iteration are scattered throughout the address space. At the end of the force computation phase, the processes modify the bodies that they own. In the next phase (position computation), the processes continue working on the bodies they own. However, the barrier between these phases in the original program forces the run–time software DSM system to make the address space consistent. This causes a large amount of data communication during the position computation phase, when the processes continue working on the bodies they own. As detected by \( R_X \), making the address space consistent at this point is unnecessary. The semantics of the control barrier convey this to the run–time system. Thus, at eight processes, while the original program suffers from false sharing and moves 173 Mbytes of data in about 120,000 messages, the optimized program moves 141 Mbytes in half as many messages.

Shallow shows a remarkable performance improvement for the same reason. When process 0 does the boundary element exchanges during the periodic continuation phase, it is essentially shipped the entire address space. Later, when the other processes access the band of rows they "own", process 0's changes have to be shipped back to them. Thus, while the original program moves 830 Mbytes of data in about 344,000 messages, the optimized program moves 548 Mbytes in just 16,000 messages.

As in the case of the hardware platform, the improvement in Conjugate Gradient is primarily because of the removal of the ns–atomic critical section in the sparse–multiply step. Removing this critical section eliminates the serialization of the execution of this part of the program.

The improvement in Multigrid is due to a reduction in the number of messages. By converting the barriers into flags, the number of messages required for synchronization is reduced. For instance, an 8–process barrier requires 14 messages to and from the barrier manager, while a flag release followed by an acquire takes just 2 messages when the releasing process is the flag manager, and 3 when it is not. Thus while the amount of data transferred remains the same for both versions, the optimized version sends only about 77,000 messages compared to 92,000 for the original.

The lack of improvement in Gauss is because the pivot process has to signal all the other processes. Thus, the two versions send the same number of messages. There is an interesting anecdote about this application. When we first ran the flags version, it was much slower, sending more data and messages than the original version. This was because in the optimized version, TreadMarks was not performing any garbage collection on its internal data structures. Garbage collection in TreadMarks
is performed, if needed, at barriers. By eliminating all barriers from the bulk of the computation, TreadMarks was unable to perform garbage collection. We remedied this by inserting a barrier once every 128 iterations, giving TreadMarks an opportunity to perform the garbage collection. This brought the performance to be on par with the original version, as we had originally anticipated.

The performance improvement in Water comes about because of the large number of critical sections that are eliminated. While the original version moves 1.4 Gbytes of data in 12.2 million messages, the optimized version sends just 56 Mbytes of data in 140,000 messages. The improvement in performance can be directly attributed to this decrease.

7.5 Cost of Performance Debugging

7.5.1 Experimental Platform

Our experimental platform for running the instrumented executables is a network of four SparcStation-20s with 53Mhz SuperSparc processors running SunOS4.1.3, connected by a 155Mbps ATM network. The main reason for choosing this platform was the convenience it provided from a software development point of view. In particular, the two platforms used in the performance evaluation were not available under local control.

Our instrumented TreadMarks library is based on TreadMarks version 0.9.3. Lazy diffing was disabled in order to facilitate recording the writes in each interval. We used gcc-2.5.8 for compiling the applications. Subsequent versions of gcc for the Sparc (including version 2.7.2), do not implement the -ffixed-reg option correctly.\footnote{The bug is in config/sparc/sparc.h, and will apparently be fixed with version 2.8.0}

7.5.2 Performance Debugging Costs

In earlier sections, we have examined the efficacy of our performance debugging technique. In this section, we focus on the associated cost. We use three indicators to measure the overheads of performance debugging: the memory overhead, the execution time dilation, and the size of the log files produced during execution.

Several challenges had to be surmounted in order to tractably collect the run-time information needed for prescribing source-level feedback. Minimizing the information
collected at run-time and computing the source operands of writes to memory are two important ones. We explored several means of performing these operations before arriving at the techniques used in our prototype implementation.

The memory overhead imposed by our prototype is a factor of 40. This includes all memory, i.e., for the data+bss section, and for memory allocated on the heap and stack. The overhead is high because we use a sparse data structure for storing the state information described in Chapter 5. We use this data structure mainly to facilitate faster code development. In addition, the smallest granularity at which memory can be accessed on our platform is a byte. Therefore, we allocate state for each byte of memory. In Section 7.5.3, we describe how this overhead can be reduced.

Table 7.7 presents the execution times of the instrumented applications, and also the size of the log files that are generated. The first and second columns list the application and the data set that we used for performance debugging. The third column shows the time taken to run the program on one processor, devoid of any TreadMarks library calls (the sequential running time). The next two columns list the times taken to run the uninstrumented and instrumented versions of the program on four processors. The sixth column lists the execution time dilation as the ratio of the instrumented and uninstrumented parallel execution times. The last column reports the total size (in Mbytes) of the compressed log files generated during execution.

<table>
<thead>
<tr>
<th>App</th>
<th>Data set</th>
<th>Seq</th>
<th>4-P</th>
<th>4-P, instr</th>
<th>Dilation</th>
<th>Log size</th>
</tr>
</thead>
<tbody>
<tr>
<td>Barnes</td>
<td>8192 bodies*. 5 iters</td>
<td>30.87</td>
<td>15.65</td>
<td>1686.75</td>
<td>107.8</td>
<td>0.722</td>
</tr>
<tr>
<td>CG</td>
<td>SMALL</td>
<td>9.45</td>
<td>23.79</td>
<td>343.98</td>
<td>14.5</td>
<td>0.234</td>
</tr>
<tr>
<td>Gauss</td>
<td>1024 × 1024</td>
<td>72.29</td>
<td>33.49</td>
<td>4789.61</td>
<td>143.0</td>
<td>5.361</td>
</tr>
<tr>
<td>Multigrid</td>
<td>64 × 64 × 64. 20 iters</td>
<td>35.46</td>
<td>26.56</td>
<td>1083.82</td>
<td>40.8</td>
<td>2.438</td>
</tr>
<tr>
<td>Shallow</td>
<td>256 × 256. 10 iters</td>
<td>4.66</td>
<td>32.61</td>
<td>804.37</td>
<td>24.7</td>
<td>0.672</td>
</tr>
<tr>
<td>Water</td>
<td>64 molecules, 5 iters</td>
<td>0.83</td>
<td>6.87</td>
<td>51.70</td>
<td>7.5</td>
<td>0.673</td>
</tr>
</tbody>
</table>

Table 7.7 Instrumented application characteristics. All times are in seconds.

*We erroneously reported this as 16384 in our earlier work [RC97b].

Note that in spite of this dilation, the perturbation effects can be reduced so that they do not skew the results obtained from the performance debugging process. Section 5.7 discusses the details of the "record and replay" technique that can be used to achieve this.
The data sets that we used for performance debugging are in general, smaller than those used in the performance evaluation. The main reason for this is to permit the profiled executable to execute in a reasonable amount of time. In the case of Conjugate Gradient and Multigrid, we were also limited by the amount of swap space available.

The extremely small data set used for Water is due to our disabling lazy diffing inside TreadMarks. Owing to the large number of critical sections in this program, a large number of write notices are created during the force computation phase. At a synchronization point, all this consistency data has to be shipped to another processor. When the data set is increased beyond 64 molecules, the amount of consistency data exceeds the maximum packet size (the MTU) for the operating system on our platform. Three steps can be taken to fix this limitation. First, we can remove the restriction that lazy diffing be disabled. This limitation is currently present only to permit the writes in an interval to be determined from the diffs created during that interval. We can record the writes even in the absence of the diffs by comparing all modified pages with their twins. Second, we can port our implementation to a more recent TreadMarks version that compactly encodes the consistency information. Finally, we can enable TreadMarks to send consistency messages that are made up of multiple packets.

Two points regarding the data in Table 7.7 are worth mentioning. First, the execution time dilation shows a wide disparity. This is because the cost of merging the PROCINT information in the mirror basic blocks dominates the instrumentation overhead. Function calls to the mirror basic blocks, and to the profiling code for memory accesses also have an impact, though to a much lesser degree. The total number of merge operations during execution is proportional to the number of source operands of the stores in the program (see Figure 5.5). Gauss and Barnes–Hut have more operands per store than the other programs; hence they get penalized more.

The second point is that the running times and the log file sizes do not exhibit a correlation. This is because the log file sizes are determined by the synchronization rate and the spatial and temporal locality in the accesses of the program. When we write the data to the log files, we merge adjacent address ranges together. Thus, if the program has a high spatial and temporal locality, and its synchronization rate is small, we are able to merge a large number of the accesses and represent them compactly.
7.5.3 Reducing the Performance Debugging Costs

Both the memory and execution time overheads imposed by our prototype can be fairly high. Below, we discuss methods that can be used to reduce these overheads.

Our prototype currently performs very limited, localized analysis when generating the mirror functions. By applying simple optimizations during this process, the number of PROCINT merges at run-time can be significantly reduced. For instance, we performed two optimizations manually in Gauss. First, we unrolled the main computation loop that adjusts the columns in the array based on the current pivot column. Then, we performed a variant of common subexpression elimination when generating the mirror function. Together, these two optimizations brought the execution time dilation down from 143 to 77. We have not attempted this optimization on the other applications on account of their complexity. Gauss’ simple structure facilitated the manual optimization.

A limited amount of compile-time support can also reduce the instrumentation overhead. Currently, the source operands of program accesses are tracked at the granularity of individual stores. By identifying updates at a larger granularity, the cost of merging the PROCINT information can be amortized. The static analysis would only have to identify the granularity with which data is accessed in a region of the program. This can have a significant impact on applications like Barnes–Hut and Water, which operate on arrays of structures.

The execution time overhead can also be reduced by specializing the instrumentation procedures that are invoked from the mirror functions. These functions use several variables that are initialized once at program startup, and remain unchanged through the life of the program. Since these variables are different for each process, specializing the instrumentation code on a per-process basis will eliminate costly accesses to these variables. This can be done by generating and installing per-process versions of the instrumentation functions at program startup.

Compile-time support or programmer annotations can also reduce the memory overhead associated with the state information required for performance debugging. Specifically, if static analysis or programmer annotation can identify the granularity of all accesses to a particular region of memory, state information for that region can be stored at the larger granularity. We use this mechanism currently to store the

\[ \text{gcc-2.5.8 does unroll loops, but the resulting code is peppered with branches. This negated our purpose of unrolling, which was to increase the size of the basic block being instrumented.} \]
state for all shared memory locations at a word granularity. We can do this because in the version of TreadMarks we use, different processes cannot modify portions of a word concurrently, without one of the modifications being lost after a synchronization between the two.\textsuperscript{11} The memory overhead can also be reduced significantly by moving away from a sparse representation of the PROCINT information.

Finally, the log file sizes can be reduced using increased aggregation and by employing a special (binary) format. Currently, accesses are aggregated at a page granularity. Furthermore, the log information is generated as ASCII files, that are compressed on the fly using the Gnu gzip utility. Increasing the scope of aggregation, and devising a special binary format for storing the log data can reduce the log size.

7.6 Summary

This chapter demonstrates the viability of using a prescriptive tool to performance tune complex applications. We also examine three aspects of the performance debugging process to demonstrate the tractability of our approach.

First, we present the feedback provided by $R_x$ for a broad class of applications derived from benchmarks, course programming projects and an industry source. For all six applications, $R_x$ was able to prescribe transformations that significantly reduced synchronization and data communication.

Next, we used two shared memory platforms to evaluate the performance impact of the debugger feedback.

The hardware platform is representative of the SMP systems that are widely available now. Synchronization is fairly cheap in this environment: an 8-processor barrier costs only $25\mu$s. As is to be expected, reducing synchronization alone has little performance impact on this platform, unless the reduction in synchronization is large compared to the running time of the application. Water illustrates this fact.

For the application suite we looked at, performance improvements come about by improving locality and load balance, as in the case of Shallow, and by eliminating serialization of the execution, as in the case of Conjugate Gradient.

The second platform is representative of software DSM systems available today. The underlying communication paradigm in this environment is message passing. Consequently, synchronization is a fairly expensive operation, an 8-processor barrier costs $880\mu$s. This causes the performance improvements from implementing the

\textsuperscript{11}This is due to a deliberate optimization employed by TreadMarks to speed up the "diffing" of data.
debugger feedback to be much more significant than in the case of the hardware platform. For Water and Shallow, the debugger feedback improved performance dramatically, by factors of 10 and 66 respectively. The software layer we use is fairly aggressive (TreadMarks with MPL), so the improvements we obtained are likely to be more substantial with other, less optimized, software DSM systems.

Finally, we demonstrate the feasibility of our approach through a detailed examination of the overheads imposed when performance debugging these applications. Our implementation maintains state for memory locations, and partially processes it at run-time. This keeps the performance debugging process tractable. Depending on the application characteristics, our implementation caused the execution time to increase by between factors of 7.5 to 143. Storing the state associated with the performance debugging imposed a memory overhead of a factor of 40. The log files produced during the execution were small, all under 5.5 Mbytes in size. We also indicate how to (significantly) reduce the memory overhead and execution time dilation imposed by our prototype.
Chapter 8

Conclusions and Future Directions

This dissertation demonstrated a new approach to performance tuning: *prescriptive* performance debugging. Our approach can greatly reduce the burdens imposed on the programmer compared to existing performance analysis tools. To substantiate this claim, we developed $R_x$, a tool that helps improve the performance of explicitly parallel shared-memory programs. $R_x$ automatically analyzes run-time data from program executions to prescribe transformations that reduce synchronization and some forms of data communication. This feedback is at the *source-code level*, eliminating the need for machine-level reasoning about the program. A correctness framework ensures that transformations obtained from one or a small set of executions will be applicable to all executions.

8.1 Conclusions

We began this dissertation by outlining a new approach for designing performance tools. The key idea is that by satisfying three basic requirements, a performance tool can *prescribe* source-level changes to improve performance. The requirements are:

- Automatically analyze run-time data to derive feedback
- Correlate the feedback with the source program
- Provide a framework to establish correctness of the feedback

This approach can be used to design prescriptive tools for a wide variety of purposes, such as reducing inter-process interactions in concurrent programs, improving the cache behavior of sequential programs by changing the data layout, and indicating the best communication primitives to use in message-passing programs. The advantage of building a tool using our approach is that it can be used by novice programmers to correct performance problems, while requiring only source-level, as opposed to architecture-related, reasoning about the program.
$R_X$ is one such tool that we have developed to improve the performance of explicitly parallel shared-memory programs. $R_X$ targets inter-process synchronization and data communication, two significant sources of overhead in shared-memory applications. $R_X$ automatically analyzes run-time data from program executions to prescribe transformations. This feedback, which is at the source-code level, can be used to reduce synchronization and some forms of data communication. A correctness framework ensures that transformations obtained from one or a small set of executions will be applicable to all executions.

We presented the set of algorithms that are used in $R_X$ to detect excess synchronization and data communication. These include new algorithms we have developed for reducing synchronization and data communication by restructuring computation. Although we implemented these algorithms in a run-time tool, they are also suitable for inclusion in a compiler. Our choice of a run-time approach ensures that we are not constrained by the limitations of static analysis, and enables us to performance tune complex, pointer-ridden applications.

Several challenges need to be overcome to make our run-time approach feasible. Two of the most important issues are what information to collect at run-time, and dynamically finding out the source operands of writes to memory. We present novel techniques to handle these and other problems. By judiciously mixing run-time processing and data collection, we are able to perform all our offline analyses using only information on the shared accesses. Using equivalence classes for stores enables us to compactly record the source operands of memory writes. Finally, compared to interpreting statically produced basic block information, mirror functions enable us to carry out the run-time processing with far lower overheads.

Our results corroborate the usefulness of designing tools based on our approach. $R_X$ was able to successfully performance tune complex, pointer-ridden applications, providing source-level changes that reduced synchronization and data communication. The impact of these changes depend on the platform under consideration. Feedback from $R_X$ improved performance significantly on the software DSM platform. Compared to the sequential running time, $R_X$ prescriptions enabled two applications, Shallow and Water, to achieve speedups of 5.9 and 6.8 on eight processors. The original versions of both programs exhibited significant slowdowns on eight processors. The reduced cost of synchronization on the hardware DSM platform resulted in the $R_X$ feedback having a smaller impact. Still, at eight processors, $R_X$ feedback improved the performance of Conjugate Gradient by a factor of 1.28.
We also present the overheads associated with our approach. For the applications we looked at, the execution time dilation ranged between factors of 7.5 and 143. The log files produced during execution were all under 5.5 Mbytes in size. We also presented methods to reduce these overheads.

Although non-quantifiable, a very important measure is the ease with which we were able to performance tune the applications. In most cases, the entire operation of instrumenting the applications, executing them, collecting the data, and applying the feedback prescribed by $R_X$ took under two hours.

8.2 Future Directions

Our view is that the machine-level complexity of application development ought to be handled by programming systems and tools, freeing the programmer to concentrate on the higher-level, algorithmic aspects. Thus, we feel that in order for high-performance computing to truly succeed, there is an urgent need for tools. The approach for designing prescriptive tools that we have demonstrated in this dissertation is one step towards this goal. We foresee two main avenues for future work.

8.2.1 Prescriptive Tools for Other Domains

Improving the Performance of Concurrent Programs

Parallel and distributed applications can be sped up by reducing inter-process interactions. Although the implementation described in this dissertation handles programs written in C, our ideas are directly applicable to other programming models and languages. Multithreaded Java programs are one example.

Operating systems are another area in which our techniques can be applied. Algorithmic complexity and the temporally distributed development process often cause operating systems to have large amounts of excess synchronization. We plan to demonstrate the feasibility of using the approach presented in this dissertation to reduce synchronization in an operating system.

Improving the Cache Behavior of Sequential Programs

It is well known that the memory performance of a sequential program is affected by the placement of its data structures in memory. Changing this layout can reduce
conflict misses and also increase implicit prefetching by improving spatial locality. Consequently, the application performance can be improved.

Existing tools that target memory performance [GH93, LW94, MGA95, SB94] present only descriptive information, such as cache statistics, to the user. In contrast, a tool based on our approach can directly specify the best layout to use.

A prescriptive tool can treat this as an optimization problem where the goal is to optimally arrange the program data structures. The first step is to analyze an access trace from the application and determine an alternate arrangement of the program data structures. The problem of finding a suitable arrangement that best utilizes the cache can be cast as a variant of the graph coloring problem. Nodes in the graph correspond to the program data structures. Edges represent temporally interleaved accesses to data. The problem is to assign addresses (i.e., cache lines) to the nodes, such that two data structures connected by an edge fall on distinct cache lines. An optimal solution can be found by formulating this as an integer programming problem and solving it using one of the many solvers (such as CPLEX [Bix92]) that are currently available. The next step is to correlate this feedback with the source program, so that it can be provided in terms of the data structure names used by the programmer. In the case of rearrangements to statically allocated data structures, the tool can directly provide a mapfile to the loader, obviating the need for programmer involvement. Dynamic data structures need to be handled differently, possibly by specifying the creation of different memory “pools”, and informing the programmer of the particular pool from which to allocate each heap-based data structure. Each memory pool can be made to map to a particular portion of the cache by also specifying a mapping between virtual and physical memory to the operating system.

If the programming language permits variable aliasing, program correctness may be affected by changing the internal arrangement of data structures. The debugger must then provide the means to determine that program correctness will not be affected. An approach similar to the one we take for establishing the correctness of $R_X$ transformations (see Section 3.5) can be used for this purpose.

Optimizing Message-passing Programs

Our approach can also be used to design tools that facilitate the development and porting of message passing programs. With the increasing complexity of message-passing interfaces, the task of choosing the appropriate communication primitive is
not easy. For instance, MPI [Mes95], a widely implemented message–passing interface standard, specifies around 100 primitives for communication, including several complex ones for group communication.

The large selection of communication primitives makes it difficult for a novice programmer to choose the best primitive from the communication library. A prescriptive tool can automatically analyze run–time data to specify the best primitive to use from among those available.

The first step is to analyze messages sent by the application, and potentially, their contents. This could be used to determine opportunities for using less expensive collective communication primitives in place of the point–to–point primitives being used. This can be cast as a least–cost isomorphism problem on graphs.\(^*\) Again, integer programming solvers could be used to automatically arrive at the optimal choice of communication primitives. The second step would be to correlate this feedback with the source program. This requires pointing out the specific communication calls in the program that would need to be changed; information that can be easily obtained from the symbol table of the executable. Finally, a framework would have to be provided to ensure that program correctness will not be violated by the substitution. An approach similar to the one we take for establishing the correctness of $R_X$ transformations (see Section 3.5) can be used for this purpose.

### 8.2.2 Augmenting Run–time Analysis with Compile–time Analysis

Run–time analysis forms the basis of most performance tools. Our plan is to complement this with compile–time analysis to yield an integrated approach to performance tuning. Such a combination has several potential benefits. The run–time analysis need then be carried out only for those parts of the program that cannot be precisely analyzed without running the program. Static analysis can also provide information about the program which can then be used to reduce the amount of data that must be collected at run–time. For instance, as we mention in Section 7.5.3, compile–time information can be used to reduce the instrumentation overheads associated with the performance debugging process.

---

\(^*\)The communication structure of the application can be used to create a communication graph. Each communication primitive provided by the library can be viewed as a subgraph with an assigned cost. The problem is to find a least–cost set of subgraphs that covers the communication graph.
Appendix A

Terminology for Chapter 3

This appendix defines four terms used in Chapter 4, viz. candidate critical path, critical path, sender, and receiver. With the exception of the first, these terms were originally introduced and defined by Adve [Adv93]. Unless otherwise stated, all definitions are being presented from Adve’s work [Adv96].

A.1 Preliminary Definitions

Definition 1: Program order: For a given execution, the program text imposes an order on the operations of an individual process. The union of these per-process orders is the program order.

Definition 2: Conflicting operations: Two operations conflict if they access the same locations and at least one of them is a write [SS88].

Definition 3: Execution order: With atomic memory, there is a total order on all memory operations of an execution, such that a read returns the value of the last write to the same location in this total order. This order is called the execution order [Col92].

We say an operation \( O_1 \) executes before (or after) an operation \( O_2 \) if \( O_1 \) is before (or after) \( O_2 \) by execution order.

Definition 4: Conflict order (\( \xrightarrow{\text{conflict}} \)): Let \( X \) and \( Y \) be two memory operations in an execution. \( X \) is ordered before \( Y \) by conflict order \( (X \xrightarrow{\text{conflict}} Y) \) iff \( X \) and \( Y \) conflict and \( X \) executes before \( Y \) [AH92].

Definition 5: The program/conflict graph for an execution is a directed graph where the vertices are the memory operations of the execution and the edges represent the program order and conflict order on the operations.
Definition 6: An ordering path is a path in the program/conflict graph that is between two conflicting memory operations and has at least one program order edge.

Definition 7: A synchronization loop [GAG+92] is a sequence of instructions that satisfies the following. (i) The sequence executes one or more reads, called exit reads. Depending on whether the value returned by each read belongs to one of specified values, called exit values, the construct either terminates or repeats the above. (ii) The loop terminates in every sequentially consistent [Lam79] execution.

An exit read can be part of a read-modify-write if it is the only (static) exit read instruction in the loop.*

Definition 8: Unessential and essential operations: A set of operations in an execution are unessential if they are from an iteration of a synchronization loop that does not terminate the loop. Other operations are essential [GAG+92].

Definition 9: Two conflicting operations in an execution, X and Y, are called consecutive iff there is no ordering path between X and Y that has another write on it that conflicts with X and Y.

A.2 Final Definitions

Definition 10: A critical set of paths for an execution is a set of ordering paths that obey the following properties. Ignore all unessential operations. Let Y be any (essential) operation in the execution.

Case 1: Y is not an exit read of a synchronization loop: If X and Y are consecutive conflicting operations and if there is an ordering path from X to Y, then one such path is in the critical set.

Case 2: Y is an exit read of a synchronization loop: Let W and W' be the last two (by execution order) conflicting writes before Y that write an exit value for Y; let W' execute before W. If W' exists, then for the set of ordering paths to Y from writes between (by execution order) W' and Y, one ordering path that ends in a program order edge is in the critical set (if such a path exists).

*More general forms of synchronization loops are possible [Adv93, GAG+92]. A read-modify-write, as used in Definition 7, is defined as a read followed by a write to the same location; further, the system guarantees that there is no other conflicting write between the read and write of a read-modify-write (as ordered by the execution order) [Adv93].
For every execution, we consider one specific critical set, and call the paths in that set as critical paths.

**Definition 11:** Any ordering path that can potentially be a critical path is called a *candidate critical path*.

**Definition 12:** Consider the two operations that lie on either side of a conflict order edge of a candidate critical path. The first operation on the conflict order edge is called a *sender*, and the second, a *receiver*.

---

1 The term *candidate critical path* has not been defined by Adve.
2 While Adve defines these for critical paths [Adv96], we define them also for candidate critical paths.
Bibliography


IMAGE EVALUATION
TEST TARGET (QA-3)

APPLIED IMAGE, Inc
1653 East Main Street
Rochester, NY 14609 USA
Phone: 716/482-0300
Fax: 716/288-5989
© 1993, Applied Image, Inc., All Rights Reserved