RUN-TIME PARALLELIZATION:
A FRAMEWORK FOR PARALLEL COMPUTATION

BY
LAWRENCE RAUCHWERGER

Dipl., Institutul Politehnic București, 1980
M.S., Stanford University, 1987

THESIS
Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois
Abstract

The goal of parallelizing, or restructuring, compilers is to detect and exploit parallelism in sequential programs written in conventional languages. Current parallelizing compilers do a reasonable job of extracting parallelism from programs with regular, statically analyzable access patterns. However, if the memory access pattern of the program is input data dependent, then static data dependence analysis and consequently parallelization is impossible. Moreover, in this case the compiler cannot apply privatization and reduction parallelization, the transformations that have been proven to be the most effective in removing data dependences and increasing the amount of exploitable parallelism in the program. Typical examples of irregular, dynamic applications are complex simulations such as SPICE for circuit simulation, DYNA-3D for structural mechanics modeling, DMOL for quantum mechanical simulation of molecules, and CHARMM for molecular dynamics simulation of organic systems. Therefore, since irregular programs represent a large and important fraction of applications, an automatable framework for run-time parallelization is needed to complement existing and future static compiler techniques.

In this thesis we present several original techniques that together sketch how automatic compilation can go beyond statically analyzable codes. The methods described are fundamentally efficient, scalable and general, i.e., their characteristics are not based on heuristics with a wide performance distribution across their input domain but are algorithms that can be analytically proven to produce speedups given the necessary resources and available parallelism. We introduce the idea of testing only for full parallelism in the presence of run-time transformations rather than computing an execution schedule. We introduce the aggressive strategy of speculative parallel execution (scalable to any parallel system – from micros to multiprocessors). We describe a new technique for analyzing and scheduling loops which are only partially parallel. Additionally we present a framework for the parallelization of loops which contain recurrences and have an unknown iteration space.

We believe that within the domain of automatic parallelization the true importance of this work is in breaking the barrier at which automatic parallelization had stopped: regular, well-behaved programs.

We also attempt to convey a few general ideas and dispel some older ones. Namely, optimizing at run-time implies overhead but, can actually reduce overall execution time through better exploitation of resources. Speculating about optimizations, more specifically parallelism, may be an attractive and more generally applicable alternative to the 'inspect first and execute later' strategy.

Thus we view this thesis as the first step into the full integration of these techniques into commercial parallelizing compilers.
Acknowledgments

I have been a graduate student for a long time – too long a time. Now, as I am graduating, I find it to be a good time to look back and remember all the people that have contributed to or influenced my career and my outlook on life.

First I would like to remember my grandfather, who was the first to awaken my imagination by introducing me to the universe of Greek mythology, and to Homer’s Iliad and Odyssey. He was also the one who made it clear to me that I have to study hard and be the best I can be – sometimes in no uncertain terms. My parents have pushed me hard in school – they made work and the quest for knowledge second nature to me. My drive in trying to “make something out of myself”, my hard-headed resilience to the ups and downs of life and my fighting spirit I owe to my mother and father – they conveyed it sometimes by example but always through tough education. My mother has spent hour after hour trying to teach me how to think – She’s still trying...

My late literature teacher at the German School, Nicolae Săftu, who has been my mentor throughout my 8 years of high-school, has explained me what the word 'intellectual' means. Both he and my mother teach me how to write an essay that has “head and tail” and conveys an idea rather than empty words, they made me “use my brain” and eventually crystallize my thoughts into a 'Weltanschaung’. I wish he would be still around.

At the EE department in Bucharest, my thesis adviser A. Murgan and his colleague C. Radoi told me that ‘that whatever you build, has to work. If it does, nobody will care how, but if it doesn’t, everybody will have better ideas about how you should have designed it’. Later they, as well as my physics professor, I. M. Popescu, had the courage to write a letter of reference for me, while I was considered a 'defector’. I will never forget that.

My former manager at the “Felix” computer company, Mircea Brozici, gave me the opportunity to learn a few things about computer design and then covered up my intentions to defect, while risking his position.

The road to the United States was long and very risky but I made it thanks to all those Israeli, Greek and Italian friends who have helped me in the most selfless way. My cousin Jeanette Targow has made my new start in the United States much less painful than it could have been and continues to be very supportive. Within the same context I would also like to remember my late cousin Lille.

Tsu-Chang Lee (aka Mr. Lee) has been my very good friend and office-mate at Stanford. It was an enlightening experience to realize how close our values and views were – in spite of coming from opposite ends of the world. One negative point though: He promised to teach me one new Chinese
word per day – and stopped after the first one. I picked up a few others from his conversations with his girlfriend. I encouraged him to marry her and he returned the favor a year later.

All this time I had only one fellow Romanian friend: Andrei Vladimirescu. He has given me advice and help as I walked in his footsteps. I have always enjoyed our chats about the old country.

Professor Faiman cut a few bureaucratic corners for me and helped with advice, references, financial support, and good English humor. His 'second in command', Barb Cicone, has been always very helpful.

Thanks to Prof. Dave (C. L.) Liu – I could always get a word of encouragement, help, and a good joke from him.

Prof. David Kuck listened to what I had to say for 2 hours and a month later I joined CSRD as P. Michael Farmwald’s student. 'Mike' gave me lots of ideas and encouragement but unfortunately left the school after less than two years.

Prof. Dan Reed taught me the importance of 'confidence intervals' and convinced me that statistics is not only black magic. His students – Brian, Celso, Chris, David, and Tara – all helped me at some point or another.

Prof. Ravi Iyer has had long conversations with me about performance modeling and about my career plans, offering help whenever I needed. I will always be indebted to him.

Thanks to Sharad Mehrotra’s job 'tip' I was hired for a summer at IBM by Dr. Pradheep Dubey. Pradheep took my panic mode before the paper deadline in a relaxed way.

While at IBM I have to credit Ravi Nair for believing in my ability as a researcher. He spent 2 hours every day, for four months, discussing computer architecture and just about anything else with me. He argued with me, listened to me, gave me ideas. It was one of the most productive and pleasant periods of my graduate student years – mostly due to Ravi.

When I returned to Illinois I joined the Polaris compiler group – it was the end (more correctly 'top') of the line from more than one point of view. Paul Petersen believed from the very beginning in the importance of my research and my approach to it. He helped me crystallize some of my 'fuzzy' ideas into real algorithms and then defend them to the rest of the group. Bill Blume, a great friend indeed, has been my 'sounding board' for all these years. He has given me help and discussed my ideas in detail. I have taken a lot of his time – but he gave it generously. Those discussions in the middle of night, those programming bugs fixed 'on the fly', have contributed to this thesis. The rest of the Polaris bunch, R. Eigenmann, K. Faigin, J. Grout, J. Hoellinger, J. Ku, T. Lawrence, Y. Paek, B. Pottenger, M. Sharma, P. Tu and S. Weatherford have been always helpful. The 'fights' in the group meetings were the highlight of the week – but no matter how heated the discussions, in the end we were always good friends.
Thanks to professors Kyle Gallivan, Constantine Polychronopoulos and Josep Torrellas for many useful discussions and advice. I am indebted to Bret Mansol for his help with those nasty sparse computations. The technical discussions and fun times with Antonio Lain and Dan Lavery were very enjoyable. Sheila typed all those addresses for me. Donna was always fun and helpful.

Many thanks to my doctoral committee, Professors Mike Heath, Laxmikant Kale, Wen-Mei Hwu and Pen Yew for reading my thesis and offering advice. Professors Yew and Hwu have always been very helpful in discussing both technical and career issues with me.

General camaraderie, much needed moral support, many good dinners, and “Three Dog Beer” were also provided by: Sarah Aston, Laura Kotovsky, Ran Libeskind-Hadas, Jim and Amy Needham, and Dan Simon.

There are however two people that have contributed the most (each in his/her own way), to this thesis: My adviser, Prof. David Padua (aka Señor Professor) and my wife, Nancy Amato.

Prof. Padua first convinced me to change my research topic and “do something at run-time”, because I would have a much better chance to “make an impact” in this field. It remains to be seen if I have succeeded but I thank him for giving me the opportunity to try. We had unending technical and 'worldly' discussions that have been and will likely continue to be of great use to me in the future. Whenever I told him about some other 'great idea', he focused me on the right issue without stifling my enthusiasm. He has always insisted that that I should write a paper only when I had something to say – and never before. He read my first paper on Christmas Eve and didn't complain. He has listened to me being upset, being happy, being in any way one can imagine. But I believe that his most important single contribution to my education was building up my confidence. I think that success can not be achieved without confidence and strength – because we spend most of our time on a roller-coaster to success – and safe arrival is never guaranteed. 'Señor Professor' has treated me more like a younger friend than a 'lowly' student. I will always think of him as my adviser and not as a 'boss' – I cannot imagine a better adviser than him.

Nancy has been always optimistic - always giving hope. She has been the only steady bright spot in my life since I have met her. She has done more then what 'wives usually do' (I may get in trouble here) – she has listened to all my 'little do-loop' bedtime stories and made suggestions – excellent ones – in theory at least (again I may get in trouble). I could have never made it this far without her. I won't write more about Nancy – she doesn’t need to read it – I can repeat it to her for the rest of my life.

Last, but certainly not least, I would like to thank Clouseau, for his unquestioning love and perpetual good cheer. Too bad he can't read.
Table of Contents

Chapter | Page
---|---
1 Introduction | 1
1.1 High Performance Computing through Parallel Processing | 1
1.2 Automatic Parallelization | 2
1.3 The Need for Run-Time Parallelization | 3
1.4 Thesis Organization | 5
1.5 The Goal | 6
2 A Run–Time Technique for Doall Loop Identification: The Doall Test | 7
2.1 Introduction | 7
2.2 Preliminaries | 8
2.3 The Doall Test | 11
2.4 The Privatizing Doall (PD) Test | 13
2.5 Run–time Reduction Parallelization | 16
2.5.1 The RPD test: Extending the PD Test for Reduction Validation | 16
2.5.2 Multiple Potential Reduction Statements | 19
2.6 Implementing the Tests on a Shared Memory Machine | 19
2.6.1 The Inspector Loop | 20
2.6.2 Private Shadow Structures | 21
2.6.3 Hash Tables | 21
2.6.4 Allocating the Private Variables | 22
2.7 Variants of the RPD Test | 23
2.7.1 A Processor–wise Version of the RPD Test | 23
2.7.2 Distinguishing between Fully Independent and Privatizable Accesses | 24
2.7.3 Supporting Copy-in of External Values | 25
3 Speculative Run–Time Parallelization of Loops: The LRPD Test | 27
3.1 Introduction | 27
3.1.1 A New Approach: Speculative Doall Parallelization | 27
3.2 Speculative Parallel Execution of Do Loops | 28
3.2.1 The LRPD Test | 29
3.3 Run–Time Reduction Recognition | 34
3.3.1 Static Reduction Recognition and Run-Time Check .......... 35
3.3.2 Single Statement Reduction Recognition .................. 35
3.3.3 Multiple Statement Reduction Recognition: Expanded Reduction Statements 36
3.3.4 Applicability of the Run-Time Reduction Algorithm ............ 39

4 Complexity, Performance Prediction and Implementation Strategies .... 41
4.1 Analysis Before Run-Time .................................. 41
4.2 Complexity of the Doall Tests ................................ 41
  4.2.1 Complexity of Saving and Restoring State .................. 43
  4.2.2 Complexity of Run-Time Privatization .................... 43
  4.2.3 Complexity of the Last Value Assignment .................. 44
4.3 Performance Prediction ..................................... 44
4.4 Putting it All Together ..................................... 47
4.5 Optimizations .............................................. 49
  4.5.1 Marking Phase Overhead Reduction through Reference Aggregation ..... 49
  4.5.2 Schedule Reuse ....................................... 49
  4.5.3 Inspector Decoupling ................................... 50
  4.5.4 Reducing The Risk of Potential Slowdown .................. 50

5 Experimental Results for the LRPD Test .......................... 53
5.1 Experimental Setup ........................................ 53
5.2 Experimental Results ....................................... 54
5.3 Conclusion ................................................. 57

6 Run-Time Methods for Parallelizing Partially Parallel Loops .......... 61
6.1 Introduction .............................................. 61
6.2 The Inspector ............................................. 62
  6.2.1 Implementing the Inspector ................................ 64
  6.2.2 Privatization and Reduction Recognition at Run-Time .......... 67
  6.2.3 Complexity of the Inspector ................................ 68
6.3 The Scheduler ............................................. 69
  6.3.1 A Simple Scheduler .................................... 69
6.4 Strategies for Applying Run-Time Parallelization .................. 72
6.5 Experimental Results ....................................... 74
  6.5.1 Synthetic Access Patterns .............................. 75
6.5.2 Real Access Patterns .................................................. 77
6.6 Future Improvements ..................................................... 79

7 Previous Work in Run–Time Concurrency Detection ..................... 84
7.1 Hardware Implementations ............................................... 84
7.2 Software Methods for Run–Time Parallelization of Partially Parallel Loops .... 84
    7.2.1 Methods Utilizing Critical Sections ............................... 85
    7.2.2 Methods for Loops Without Output Dependences ................. 86
    7.2.3 Related Work ..................................................... 87

8 Parallelizing While Loops for Multiprocessor Systems ..................... 90
8.1 Introduction ............................................................. 90
8.2 Transforming While Loops for Parallel Execution ........................ 92
8.3 Parallelizing the Dispatcher ............................................ 94
    8.3.1 The Dispatcher is an Induction .................................. 95
    8.3.2 The Dispatcher is an Associative Recurrence ................. 96
    8.3.3 The Dispatcher is a General Recurrence ....................... 98
8.4 Undoing Iterations that Overshoot the Termination Condition .......... 100
8.5 While Loops with Unknown Cross-Iteration Dependences ............... 101
    8.5.1 Detecting Errors in the Parallel Execution .................. 103
8.6 Transforming Arbitrary while Loops for Parallelization ................. 104
8.7 Predicting Performance ................................................ 106
8.8 Strategies for Applying the Techniques ................................ 108
    8.8.1 Statistics Enhanced Strip-Mining ................................. 109
    8.8.2 Resource Controlled Self-Scheduling ............................. 109
    8.8.3 The 1 Processor/(p – 1) Processor Solution .................. 110
8.9 Experimental Results .................................................. 110
    8.9.1 While–Doany ...................................................... 112
8.10 Related Work ......................................................... 112
8.11 Conclusion ............................................................ 113

9 Extensions, Significance of this Work, Future Research .................. 117
9.1 Further Applicability of Run–Time Methods ................................ 117
    9.1.1 Verifying Parallelized Loops ................................... 117
    9.1.2 Run–Time Check as a Parallel Language Extension ............ 117
9.2 The Contribution of this Work .................................................. 118
9.3 More Work Ahead ................................................................. 119
9.4 A Final Word ................................................................. 121
References ............................................................................ 122
Vita ...................................................................................... 130
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1 Irregular applications</td>
<td>4</td>
</tr>
<tr>
<td>5.1 Summary of experimental results</td>
<td>55</td>
</tr>
<tr>
<td>7.1 A comparison of run-time parallelization techniques</td>
<td>88</td>
</tr>
<tr>
<td>8.1 A taxonomy of while loops</td>
<td>94</td>
</tr>
<tr>
<td>8.2 Summary of experimental results</td>
<td>111</td>
</tr>
</tbody>
</table>
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1 Examples of loops with different data dependences</td>
<td>8</td>
</tr>
<tr>
<td>2.2 Example and results for doall test</td>
<td>12</td>
</tr>
<tr>
<td>2.3 Example and results for PD test</td>
<td>15</td>
</tr>
<tr>
<td>2.4 Example for RPD test</td>
<td>18</td>
</tr>
<tr>
<td>2.5 Allocating the private variables</td>
<td>22</td>
</tr>
<tr>
<td>2.6 PD test example from TRFD</td>
<td>26</td>
</tr>
<tr>
<td>3.1 Example for LRPD test</td>
<td>31</td>
</tr>
<tr>
<td>3.2 Example for LPD test</td>
<td>34</td>
</tr>
<tr>
<td>3.3 Example for single statement reduction recognition</td>
<td>36</td>
</tr>
<tr>
<td>3.4 Example for multiple statement reduction recognition</td>
<td>37</td>
</tr>
<tr>
<td>3.5 Example for expanded reduction statement (ERS)</td>
<td>40</td>
</tr>
<tr>
<td>4.1 Generated code for the LRPD test</td>
<td>48</td>
</tr>
<tr>
<td>5.1 Speedup and potential slowdown in MDG</td>
<td>58</td>
</tr>
<tr>
<td>5.2 Speedup and potential slowdown in BDNA</td>
<td>58</td>
</tr>
<tr>
<td>5.3 Speedup and potential slowdown in TRFD</td>
<td>58</td>
</tr>
<tr>
<td>5.4 Speedup and potential slowdown in TRACK</td>
<td>59</td>
</tr>
<tr>
<td>5.5 Speedup and potential slowdown in ADM</td>
<td>59</td>
</tr>
<tr>
<td>5.6 Speedup and potential slowdown in OCEAN</td>
<td>59</td>
</tr>
<tr>
<td>5.7 Speedup and potential slowdown in SPICE</td>
<td>60</td>
</tr>
<tr>
<td>6.1 Data structures for array element dependence graphs</td>
<td>63</td>
</tr>
<tr>
<td>6.2 Result of the marking phase</td>
<td>65</td>
</tr>
<tr>
<td>6.3 Example of inspector loop</td>
<td>67</td>
</tr>
<tr>
<td>6.4 A simple scheduler</td>
<td>70</td>
</tr>
<tr>
<td>6.5 Parallelism profile</td>
<td>81</td>
</tr>
<tr>
<td>6.6 Parallelism profile</td>
<td>81</td>
</tr>
<tr>
<td>6.7 Parallelism histogram</td>
<td>81</td>
</tr>
<tr>
<td>6.8 Parallelism histogram</td>
<td>81</td>
</tr>
<tr>
<td>6.9 Overhead speedup</td>
<td>81</td>
</tr>
<tr>
<td></td>
<td>Section</td>
</tr>
<tr>
<td>---</td>
<td>------------------------------------------------------------------------</td>
</tr>
<tr>
<td>6.10</td>
<td>Overhead speedup</td>
</tr>
<tr>
<td>6.11</td>
<td>Speedup</td>
</tr>
<tr>
<td>6.12</td>
<td>Speedup</td>
</tr>
<tr>
<td>6.13</td>
<td>Speedup</td>
</tr>
<tr>
<td>6.14</td>
<td>Overhead</td>
</tr>
<tr>
<td>6.15</td>
<td>Overhead</td>
</tr>
<tr>
<td>6.16</td>
<td>Scheduling overhead</td>
</tr>
<tr>
<td>6.17</td>
<td>Marking, analysis phase speedup</td>
</tr>
<tr>
<td>6.18</td>
<td>Scheduling phase speedup</td>
</tr>
<tr>
<td>6.19</td>
<td>Marking, analysis phase speedup</td>
</tr>
<tr>
<td>6.20</td>
<td>Scheduling phase speedup</td>
</tr>
<tr>
<td>8.1</td>
<td>While loop examples</td>
</tr>
<tr>
<td>8.2</td>
<td>While Loop with inductive dispatcher</td>
</tr>
<tr>
<td>8.3</td>
<td>While loop with associative dispatcher</td>
</tr>
<tr>
<td>8.4</td>
<td>While loop with linked list dispatcher</td>
</tr>
<tr>
<td>8.5</td>
<td>While loops with data dependences</td>
</tr>
<tr>
<td>8.6</td>
<td>General algorithm for loops with multiple recurrences</td>
</tr>
<tr>
<td>8.7</td>
<td>Speedup</td>
</tr>
<tr>
<td>8.8</td>
<td>Speedup</td>
</tr>
<tr>
<td>8.9</td>
<td>Speedup</td>
</tr>
<tr>
<td>8.10</td>
<td>Speedup</td>
</tr>
<tr>
<td>8.11</td>
<td>Speedup</td>
</tr>
<tr>
<td>8.12</td>
<td>Speedup</td>
</tr>
<tr>
<td>8.13</td>
<td>Speedup</td>
</tr>
<tr>
<td>8.14</td>
<td>Speedup</td>
</tr>
<tr>
<td>8.15</td>
<td>Speedup</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 High Performance Computing through Parallel Processing

Almost from the beginning of computing people have tried to build supercomputers. These high performance systems were mostly dedicated to the study of the most complex problems in science and technology. For many years, large companies and government laboratories have spent large sums of money on these systems because there was no other alternative and because the political motivation (and money) behind these massive computations was always there – national security. But then came the microprocessors: first as an inexpensive educational tool or an inexpensive embedded system for industrial applications, next as a high performance workstation for engineers, and finally as a high performance PC for everybody. In fact, in the last few years the performance, and its rate of increase, of these 'killer micros' as well as their tumbling price has created the impression that supercomputing is finished. In a way this is true – the 'big iron' of the past, i.e., the mainframes, are indeed dead but the quest for ever higher computing capacity has not stopped. Indeed, it seems that we are witnessing the reincarnation of the supercomputer at the micro-level. Semiconductor technology advances have brought a dramatic reduction of the geometric features of the 'conventional' uniprocessor chips allowing higher clock rates and a higher device density. At the same time available silicon area for a chip has increased (although not at the same rate). These technological advances have resulted in the design of more uniprocessor equivalents on a single chip thus creating the lowest level of parallel processing. Additionally, the low cost of these devices and the desire for higher throughput systems has lead to the economical use of many 'off-the-shelf' processors working in a more or less tightly coupled manner creating today's supercomputer – a parallel machine.

This development is only natural: for any given level of technology we try to create the most powerful machine – and the only way to achieve this, is to distribute a problem onto many cooperating processors. We believe that the trend of cooperating processors, whether it is the chip or system level, will increase for two reasons: first, the advance in technology will make parallel processing less expensive and more technically feasible (smaller geometries increase density and shorten interconnection lengths and therefore reduce interconnection times) and, second, it will be technology itself that will reach its limit and so any increase performance will have to be made through parallelism.
1.2 Automatic Parallelization

We have previously said that we can, we are, and we are likely to continue to build parallel machines. But exploiting their potential to solve large problems fast, i.e., obtaining scalable speedups, has remained an elusive goal. We believe that a necessary condition for parallel processing to succeed is to deliver sustainable performance (speedups across a wide variety of applications) while requiring only the same or similar efforts on the part of the user as is the case for sequential computing.

We can recognize three complementary avenues towards obtaining a scalable speedups on parallel machines:

- Good parallel algorithms – to create intrinsic parallelism in a program.
- Development of a standard parallel language for expressing parallel algorithms.
- Restructuring compilers to optimize parallel code and parallelize sequential programs.

We believe parallel algorithms are absolutely essential for writing a program that is to be executed on a parallel system. It is not possible to obtain a scalable concurrent execution from an application that employs sequential methods. Once this is agreed upon, there are two different ways to express this parallelism: explicitly or implicitly. Explicitly written parallel programs can potentially produce good performance if the programmer is very good and the problem size (and difficulty) is manageable. Achieving this requires a parallel language that is both expressive as well as standardized. If we cannot clearly express the algorithmic parallelism then the effort is wasted, and if the language is not a recognized standard, then portability difficulties will make it non-economical. Additionally, we know from experience that even for very well trained people coding only with explicit parallelism may be significantly more difficult and time-consuming than programming in a sequential language. In particular solving concurrency and data distribution issues is a very difficult and error-prone task, which is contrary to our principle that parallel and serial program development should require similar effort. Expressing explicitly all levels (from instruction to task level) of parallelism within the same language, performing explicit optimizations that are architecture specific are additional difficulties in coding parallel programs.

Just as important is the fact that parallel systems don’t run only newly written applications. There is an enormous body of existing software that must be ported and perform well on these new systems. One solution is to rewrite the so named ‘legacy’ programs [BEH+94], but this could prove to be prohibitively expensive. The alternative is to automatically transform them for concurrent execution by means of a restructuring or parallelizing compiler. This compiler should be able to
safely (without errors) detect the available parallelism and transform the code into an explicitly parallel (machine) language. Most likely it would not able to modify the algorithms used by the original programmer and therefore the performance will be limited (upper bounded by the intrinsic, and limited, parallelism of the original application).

We therefore think that the ideal parallelizing compiler should be capable of transforming both 'legacy' code and modern code into a parallel form. The new codes that use parallel algorithms can be written in an established, standard language like Fortran 77, in an explicitly parallel language or, even simpler, in a sequential language with parallel assertions. Coding in a hybrid language (serial language with non-mandatory directives) should require the least amount of effort: the programmer expresses as much knowledge about the code as he/she has, while leaving the rest for automatic processing. Such a compiler has to address two major issues: (a) parallelism detection and (b) parallelism exploitation.

Parallelism detection means finding the portions of code that can be safely executed concurrently and the relative order (synchronizations) that must be maintained between their components. For that purpose the compiler needs to analyze the code and transform it into explicitly parallel constructs.

Parallelism exploitation or optimization means using the data and control dependence information previously detected for scheduling the work on a target architecture. In this process issues like data distribution for communication minimization and scheduling for load balancing have to be addressed.

We should specify at this point that both presented compiler tasks as well as explicit parallel coding are greatly simplified if we consider a shared address space rather than a distributed one. Since we believe in ease of programming we have chosen to develop our techniques for the shared memory paradigm.

1.3 The Need for Run-Time Parallelization

The first task of a restructuring compiler is to perform a data dependence analysis and detect the segments of code that can be executed concurrently. Essentially this means performing an analysis of the memory access pattern, mainly through inspection of the array subscript expressions. From the point of view of access pattern analysis we can distinguish two major classes of applications:

1. *Regular programs* whose memory accesses can be described by a 'well-behaved' analytic function which is statically defined. Current state-of-the-art compilers do a reasonable job in analyzing them and extracting the parallelism.
<table>
<thead>
<tr>
<th>Program</th>
<th>Application Domain</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPICE</td>
<td>circuit simulation</td>
</tr>
<tr>
<td>DYNA-3D</td>
<td>structural mechanics</td>
</tr>
<tr>
<td>PRONTO-3D</td>
<td></td>
</tr>
<tr>
<td>GAUSSIAN</td>
<td>quantum mechanical simulation of molecules</td>
</tr>
<tr>
<td>DMOL</td>
<td></td>
</tr>
<tr>
<td>CHARMM</td>
<td>molecular dynamics simulations of organic systems</td>
</tr>
<tr>
<td>DISCOVER</td>
<td></td>
</tr>
<tr>
<td>FIDAP</td>
<td>modeling complex fluid flows</td>
</tr>
</tbody>
</table>

**Table 1.1 Irregular applications**

2. *Irregular programs* whose memory accesses are described through an ad-hoc map, usually represented by a subscript array in Fortran or pointers in C. These indirection maps cannot be analyzed at compile time if they are filled in only during program execution. Depending on how the subscript arrays are defined at run-time we distinguish two classes of irregular applications:

(a) Programs with *Static memory patterns* which are defined at the beginning of the execution by reading in an input file (data set). After the initial setup, the access pattern will not change but it is still not available statically.

(b) Programs with *Dynamic memory patterns* which are computation dependent and are modified from one execution-phase to another because of the changing interactions of the underlying physical phenomena they are simulating.

Techniques addressing the issue of data dependence analysis have been studied extensively over the last two decades [PW86, Wol89] but parallelizing compilers cannot perform a meaningful data dependence analysis and extract a significant fraction of the available parallelism in a loop if it has a complex and/or statically insufficiently defined access pattern. Unfortunately irregular programs, as previously defined, represent a large part of all scientific applications. We note here several examples of codes that fall into this category and which are summarized in Table 1.1. The first two, SPICE and DYNA3D are input dependent while the others are dynamic programs. Since our modeling techniques are becoming more sophisticated we believe that future large scale simulations and computation will be dynamic in nature and will only increase the fraction of statically non-analyzable codes (currently estimated to be over 50%).

Thus, in order to realize the full potential of parallel computing it has become clear that static (compile-time) analysis must be complemented by some new methods [BE92, CPHL94, EHLP91].
We need techniques that let us access the information necessary to decide if a loop is parallel and perform parallelizing transformations. The only time this data is available is during program execution, at run-time. Run-time techniques can succeed where static compilation fails because they have access to the input data. For example, input dependent or dynamic data distribution, memory accesses guarded by run-time dependent conditions, and subscript expressions can all be analyzed unambiguously during execution.

Compilers can fail also due to limitations in their current analysis algorithms. For example, most dependence analysis algorithms can only deal with subscript expressions that are linear in the loop indices. In the presence of complex non-linear expressions, a dependence is usually assumed. Global parallelization, i.e., deeply nested loops containing subroutine calls, is quite often not possible because inter-procedural analysis generates extremely complex expressions that are in the end intractable although statically defined.

This thesis will present several new run-time methods – some are improvements of previous ones [BS90, LZ93, MP87, SM91, SMC89, SMC91, WSHB91, ZY87, CYT94] and others represent totally new approaches. They collectively constitute an effective framework for run-time parallelization.

In the next sections we will be mostly concerned with the parallelism detection side of the compiler. We will narrow the scope of our discussion to loop level parallelism in a shared memory environment. We believe that these restrictions do not result in any loss of generality.

1.4 Thesis Organization

In the Chapter 2 we briefly introduce the important issues in data dependence analysis and present what the problems in automatic parallelism detection are. We then present a new algorithm for run-time detection of fully independent loops: the doall test. The same method is used to determine when it is correct to apply some of the most efficient code transformations: privatization and reduction parallelization. These transformations are considered to be so important because they are capable of removing certain types of dependences that occur quite frequently in the benchmarks we have studied and thus release more parallelism. Chapter 3 introduces a novel strategy for automatic parallelization based on the previously described doall test: speculative parallel execution. In Chapter 4 we analyze the performance of the presented techniques analytically and predict their performance in a real code setting. Further we describe ways for their optimization and strategies for their automatic application within compiler. Chapter 5 presents
the experimental setup and the results obtained in the evaluation of the techniques described in Chapters 2 and 3.

Chapter 6 presents a method for extracting parallelism from partially parallel loops, i.e., loops that are neither sequential nor fully parallel – it represents an improvement of earlier techniques. Algorithm analysis, performance prediction and experimental results are included.

Chapter 7 enumerates and characterizes other previous run-time parallelism detection methods described in the literature. Other related work is also mentioned.

Chapter 8 deals with compile/run-time techniques for the automatic parallelization of loops with unknown iteration space and those containing recurrences. A general framework is given and experimental results are shown. Previous work for this separate topic is also listed.

Finally, in Chapter 9, we introduce other possible applications of the presented material and suggests future improvements. The thesis concludes with a summary of this works’ most important contributions to parallel processing and its more wide-ranging importance to related fields.

1.5 The Goal

The immediate goal of this work is to lay the groundwork for a run-time parallelization framework. The collection of techniques described in this thesis is intended to be used by a parallelizing compiler as complementary solutions when all static analysis fails. It is a hybrid approach that we hope will make possible the automatic parallelization of irregular applications.

We are also trying to convey a few general ideas and dispel some older ones: optimizing at run-time implies overhead but, if done efficiently, can actually reduce overall execution time through better exploitation of resources. Speculating about optimizations, more specifically parallelism, may be an attractive and more generally applicable alternative to the 'inspect first and execute later' strategy.

Thus, we view this thesis as the first step in a process, whose goal is to fully integrate these techniques into commercial parallelizing compilers.
Chapter 2

A Run–Time Technique for Doall Loop Identification: The Doall Test

2.1 Introduction

In this chapter we approach the problem of determining the parallelism of loops at run–time from a different viewpoint – instead of finding a valid parallel execution schedule for the loop, we focus on the problem of simply deciding if the loop is fully parallel, that is, determining whether or not the loop has cross-iteration dependences. Our interest in identifying fully parallel loops is motivated by the fact that they arise frequently in real programs: From an analysis of the manual parallelization of the PERFECT suite [EHLP91] it has been found that most of the DO loops (75%) are fully parallelizable and that they often represent almost all sequential execution time of the programs. In addition, an efficient test for full parallelism of a loop could be used as a building block for more complex run–time dependence analysis techniques in order to produce valid execution schedules for partially parallel loops.

As we show, the analysis needed to test whether or not a loop is fully parallel can be done very efficiently at run–time without the use of any synchronization (except for barriers), and can produce scalable speedups.

In order to parallelize DO loops, quite often it is necessary to apply transformations that remove certain types of dependences. It has been experimentally shown that the most effective transformations are privatization and reduction parallelization. The privatization transformation is capable of eliminating certain types of memory related dependences (anti and output dependences) through the replication of the ‘offending’ shared variables and their allocation in private, non-shared storage. The reduction parallelization transformation eliminates the specific data dependences (including flow dependences) caused by the reduction operation; It is in fact a simple form of algorithm substitution. An effective parallel reduction method is to compute the partial results of the operation in private storage and then perform a cross-processor reduction and update the global variable. These transformations can be performed only when the access and operation pattern to the shared variable under question fits a certain pattern. In the latter part of this chapter we will show how to check for this patterns at run–time.

Our methods are fully parallel, require no synchronization, and can be applied to any loop.
At this point it is necessary to point out that the presented material has been developed within the context of a Fortran parallelizing compiler. We believe that the principles on which the work is based will remain valid in other environments, e.g., C and C++.

2.2 Preliminaries

A loop can be executed in fully parallel form, without synchronization, if and only if the desired outcome of the loop does not depend in any way upon the execution ordering of the data accesses from different iterations. In order to determine whether or not the execution order of the data accesses affects the semantics of the loop, the data dependence relations between the statements in the loop body must be analyzed [Ban88, KKP+81, PW86, Wol89, Zim91]. There are three possible types of dependences between two statements that access the same memory location: flow (read after write), anti (write after read), and output (write after write). Flow dependences are data producer and consumer dependences, i.e., they express a fundamental relationship about the data flow in the program. Anti and output dependences, also known as memory–related dependences, are caused by the reuse of memory, e.g., program variables.

\[
\text{do } i=1, n \\
\text{A}(K(i)) = A(K(i)) + A(K(i-1)) \\
\text{if } (A(K(i)) \text{ eq. true.}) \text{ then} \\
\text{.................} \\
\text{endif} \\
\text{enddo}
\]

\[
\text{do } i = 1, n/2 \\
\text{S1: } \text{tmp} = A(2^i) \\
\text{A}(2^i) = A(2^{i-1}) \\
\text{enddo}
\]

\[
\text{do } j = 1, m \\
\text{S1: } A(j) = A(j) + \exp() \\
\text{enddo}
\]

\[
\text{do } i=1, n \\
\text{endif} \\
\text{enddo}
\]

\[
\text{S2: } A(2^i-1) = \text{tmp} \\
\text{enddo}
\]

\[
\text{Figure 2.1 Examples of loops with different data dependences}
\]

If there are flow dependences between accesses in different iterations of a loop, then the semantics of the loop cannot be guaranteed if the loop is executed in fully parallel form. The iterations of such a loop are not independent because values that are computed (produced) in some iteration of the loop are used (consumed) during some later iteration of the loop. For example, the iterations of the loop in Fig. 2.1(a), which computes the prefix sums for the array \( A \), must be executed in order of iteration number because iteration \( i + 1 \) needs the value that is produced in iteration \( i \), for \( 2 \leq i \leq n \).

Quite often, if there are no flow dependences between the iterations of a loop, then the loop may be executed in fully parallel form. The simplest situation occurs when there are no anti, output, or flow dependences. In this case, all the iterations of the loop are independent and the loop, as is, can be executed in parallel. For example, there are no cross-iteration dependences in the loop shown
in Fig. 2.1(b), since iteration $i$ only accesses the data in $A[i]$, for $1 \leq i \leq n$. If there are no flow dependences, but there are anti or output dependences, then the loop must be modified to remove all these dependences before it can be executed in parallel. Unfortunately, not all situations can be handled efficiently. In order to remove certain types of dependences and execute the loop as a doall, two important and effective transformations can be applied to the loop: privatization and reduction parallelization. In order to remove certain types of anti and output dependences a transformation called privatization can be applied to the loop. Privatization creates, whenever correct, for each processor cooperating on the execution of the loop, private copies of program variables that give rise to anti or output dependences (see, e.g., [PW86, BCFH89, Li92, MAL92, TP92, TP93]). The loop shown in Fig. 2.1(c), which, for even values of $i$, swaps $A[i]$ with $A[i - 1]$, is an example of a loop that can be executed in parallel by using privatization; the anti dependences between statement $S4$ of iteration $i$ and statement $S2$ of iteration $i + 1$, for $1 \leq i < n/2$, can be removed by privatizing the temporary variable $tmp$.

Reduction parallelization is another important technique for transforming certain types of data dependent loops for concurrent execution.

**Definition:** A reduction variable is a variable whose value is used in one associative operation of the form $x = x \odot exp$, where $\odot$ is the associative operator and $x$ does not occur in $exp$ or anywhere else in the loop.

Reduction variables are therefore accessed in a certain specific pattern (which leads to a characteristic data dependence graph). A simple but typical example of a reduction is statement $S1$ in Fig. 2.1(c). The operator $\odot$, in this case the $+$ operator, the access pattern of array $A(\cdot)$ is read, modify, write, and the function performed by the loop is to add a value computed in each iteration to the value stored in $A(\cdot)$. This type of reduction is sometimes called an update and occurs quite frequently in programs.

There are two tasks required for reduction parallelization: recognizing the reduction variable, and parallelizing the reduction operation. (In contrast, privatization needs only to recognize privatizable variables by performing data dependence analysis, i.e., it is contingent only on the access pattern and not on the operations.)

Parallel reductions algorithms have been developed for quite some time. If the reduction operation is commutative – a quite frequent case – then the implementation of such methods is less restrictive. One typical method for the case of commutative reductions is to transform the do loop into a doall and enclose the access to the reduction variable in an unordered critical section [EHLP91, Zim91] – a section of code guarded by a lock – unlock operation which allows mutually
exclusive operations on the shared variable. Drawbacks of this method are that it is not always scalable and requires synchronizations which can be very expensive in large multiprocessor systems.

A scalable method can be obtained by noting that a reduction operation is an associative recurrence and can thus be parallelized using a recursive doubling algorithm [Kru85, Kru86, Lei92]. In this case the reduction variable is privatized in the transformed doall, and the final result of the reduction operation is computed in an interprocessor reduction phase following the doall, i.e., a scalar is produced using the partial results computed in each processor as operands for a reduction operation (with the same operator) across the processors. We should note here that if the reduction operation is commutative then it can be parallelized using dynamically scheduled doalls (not only statically scheduled in monotonic order) and the cross-processor merging phase can be done in any order. Most of the reductions encountered in our experiments were commutative.

Thus, the difficulty encountered by compilers in parallelizing loops with reductions arises not from finding a parallel algorithm but from recognizing the reduction statements. So far this problem has been handled at compile-time by syntactically pattern matching the loop statements with a template of a generic reduction, and then performing a data dependence analysis of the variable under scrutiny to guarantee that it is not used anywhere else in the loop except in the reduction statement. The case in which a reduction variable appears in more than one statement within the body of loop will be dealt with in Section 2.5.2.

In the next sections, we describe run-time techniques that can be used to determine if a loop can be executed in parallel. We first describe the doall test which determines if the loop, in its original form, can be executed in parallel, i.e., it decides if there are any cross-iteration dependences in the loop. We then explain how the doall test can be augmented to determine at run-time whether all existing memory-related dependences can be removed by privatization. Finally we show how we can extend even further the run-time test to validate reduction operands and remove all their associated data-dependences. If it is found that all dependences can be eliminated, then the sequential loop can be safely transformed into a doall by privatizing the variables which give rise to the various dependences and a cross-processor update of the original shared variables.

In order to identify the dependence relations between the iterations of the loop, the test inspects the accesses to the variables that cannot be analyzed at compile time. It is assumed that all other variables (that are statically analyzable) do not cause any cross-iteration dependence (otherwise the loop cannot possibly be parallel).

Briefly, the inspection is done by using shadow versions of the variables under scrutiny to follow (keep a record of) the data access pattern in an inspector loop which was extracted from the original loop. An inspector loop can be obtained by distributing the original loop into two loops: The first
one will compute the addresses of the shared array that is under test and the second will use these addresses to access the actual storage and perform the computation. It is important to remark here that a proper distribution is not possible ([KM90]) if the statements computing the addresses and those that use it to access the shared structure under test form a strongly connected component in the dependence graph. This fact makes proper inspector extraction impossible in the general case.

After processing all the accesses contained in the inspector loop, some additional computation determines whether all its iterations can be performed in parallel while guaranteeing the semantics of the loop. For the Privatizing doall test, an additional phase may be required to actually allocate the private variables.

It must be emphasized that the doall tests are designed to be used only on loops for which the compiler could not evaluate with certainty the data dependence relations. We recall that there are several possible situations in which it is either difficult or impossible to determine the data dependence relationships at compile time: very complex subscript expressions which could only be computed statically through deeply nested forward substitutions and constant propagations across procedure boundaries, nonlinear subscript expressions (a fairly rare case) and, most often, subscripted subscripts. Another important point is that these run-time tests must be fully parallel. We consider it imperative for the inspector loop to be parallel, and, if possible without side-effects, i.e., without modifying shared variables (see Chapter 3). If the tests cannot be executed in parallel, then they would not scale with the number of processors or the data size, and the overhead associated with the tests could become a sequential bottleneck of the loop.

2.3 The Doall Test

The doall test described below is a pass/fail test for full parallelism of a loop, i.e., it detects if there are any cross-iteration dependences in the loop through analysis of access patterns in an inspector loop. If there are any dependences, then this test does not identify them, it only flags their existence. We first describe the doall test, as applied to a shared array \( A \), and then give a few examples illustrating its use.

**Doall Test**

1. **Marking Phase.** For each shared array \( A[1 : s] \) whose dependences cannot be determined at compile time, we declare read and write shadow arrays, \( A_r[1 : s] \) and \( A_w[1 : s] \), respectively. The shadow arrays are initialized to zero, and are marked as follows.

   In parallel, for each iteration \( i \) of the loop, process all accesses to the shared array \( A \):
S1: Do i = 1, 8

S2: A[w[i]] = ...

S3: ... = A[R[i]]

S4: Enddo

\[ \begin{array}{|c|c|c|c|c|c|c|c|c|c|c|c|c|}
\hline
\text{Position in shadow arrays} & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 \\
\hline
A_w[1:12] & 1 & 1 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
A_r[1:12] & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\
A_w[1:12] \land A_r[1:12] & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
A'_w[1:12] & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 \\
A'_r[1:12] & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\
A'_w[1:12] \land A'_r[1:12] & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
\hline
\end{array} \]

<table>
<thead>
<tr>
<th>Written tm(A)</th>
<th>Counted tw(A)</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>8</td>
<td>8</td>
</tr>
</tbody>
</table>

**Figure 2.2** Example and results for doall test.

(a) Writes: If this is the first write to this array element in this iteration, then set the corresponding element in \( A_w \).

(b) Reads: If this array element is never written in this iteration, then set the corresponding element in \( A_r \).

(c) Count the total number of write accesses to \( A \) that are marked in this iteration, and store the result in \( tw_i(A) \), where \( i \) is the iteration number.

2. **Analysis Phase.** For each shared array \( A \) under scrutiny:

(a) Compute (i) \( tw(A) = \sum tw_i(A) \), i.e., the total number of writes that were marked by all iterations in the loop, and (ii) \( tm(A) = sum(A_w[1:s]) \), i.e., the total number of marks in \( A_w[1:s] \).

(b) If \( any(A_w[1:s] \land A_r[1:s]) \), i.e., if the marked areas are common anywhere, then the loop is not a doall and the phase ends. (Since we read and write from the same location in different iterations, there is at least one flow or anti dependence.)

(c) Else if \( tw(A) = tm(A) \), then the loop is a doall and the phase ends. (Since we never overwrite any memory location, there are no output dependences.)

(d) Otherwise, it is not a doall. (There are output dependences since we overwrite at least one memory location.)

To illustrate the doall test we consider the loop shown in Fig. 2.2. Assume that the shared array \( A[1:12] \) is accessed in a manner such that the dependences cannot be determined at compile

---

1\text{any} \text{returns the "OR" of its vector operand's elements, i.e.,} \text{any}(v[1:n]) = (v[1] \lor v[2] \lor \ldots \lor v[n]).
time. In the first example, the reference pattern of \( A \) within the loop is given by the subscript arrays \( W[1:8] \) and \( R[1:8] \). The `doall` test allocates, and initializes to zero, the write and read shadow arrays, \( A_w[1:12] \) and \( A_r[1:12] \), respectively. After marking and counting, we obtain the results depicted in the table. Because \( A_w[2] = A_r[2] = 1 \), we know there exists at least one flow or anti dependence. Since the number marked does not equal the number written, we know that there are output dependences (element 3 is written in iteration 2 and 4). Therefore, the loop cannot be executed in parallel. In the second example, we use the subscript arrays \( W'[1:8] \) and \( R'[1:8] \), and the shadow arrays \( A'_w[1:12] \) and \( A'_r[1:12] \). Because the number of write accesses marked equals the number written, and since \( A_w[1:12] \land A_r[1:12] \) is zero everywhere, we conclude that there are no cross-iteration dependences, i.e., the loop can be executed in parallel.

An implementation of the `doall` test will usually not adhere exactly to the above description. In particular, is should be optimized to take advantage of the target machine's architecture (e.g., private storage for the processors), and any special operations available on it. We discuss some of these implementation details in Section 2.6, and then analyze the complexity of the `doall` test in Section 4.2.

### 2.4 The Privatizing `doall` (PD) Test

The `doall` test described above is only able to detect the presence of dependences between the iterations of the loop. In this section we explain how to augment the test so that it can determine if all dependences caused by a particular array can be removed by privatizing the array. If it finds that all the dependences can be eliminated, then the Privatizing `doall` test (PD test) transforms the loop by allocating the private variables.

We now define a private variable, and state the criterion that must be satisfied in order for a variable to be determined as privatizable by the PD test.

**Definition.** A variable is private if its scope is the loop body. Therefore a private variable can only be accessed by the loop iteration to which it belongs.

**Privatization Criterion.** Let \( A \) be a shared array that is referenced in a loop \( L \). \( A \) can be privatized by the PD test if every read access to an element of \( A \) is preceded by a write access to that same element of \( A \) within the same iteration of \( L \).

In general, dependences that are generated by accesses to variables that are only used as workspace (e.g., temporary variables) within an iteration can be eliminated by privatizing the workspace. However, there are some types of dependences that the PD test does not handle. Specifically,
if a shared variable is initialized by reading a value that is computed outside the loop, then we will not privatize that variable. Such variables could be privatized if a copy-in mechanism for the external value is provided. The techniques dealing with this situation will be discussed in Section 2.7.3. The last value assignment problem is the conceptual analog of the copy-in problem. If a privatized variable is live after the termination of the loop, then the privatization technique must ensure that the correct value is copied out to the original (non-privatized) version of that variable. One way this problem can be handled is to associate with each private variable a time stamp (iteration number) that is updated at every write. After the loop has been executed, the value of the private variable with the latest time stamp is copied to the original version of the variable. Obtaining for each array element the location with the last time-stamp can be done in parallel using a pairwise-merge algorithm for the cross-processor comparison. It is in fact a cross-processor reduction operation which could also be implemented in hardware. If all iterations of the loop are writing the same elements of the array then the last value assignment problem can be simplified by writing to the original version of the variable during the last iteration. It should be noted that private loop variables are seldom live after the loop.

In order to simplify the description of the PD test given below, the last value assignment problem is not addressed. (The additions to the doall test are italicized.)

Privatizing Doall Test (PD Test)

1. Marking Phase. For each shared array \( A[1:s] \) whose dependences cannot be determined at compile time, in addition to the shadow arrays, \( A_r[1:s] \) and \( A_w[1:s] \), we declare a shadow array \( A_{np}[1:s] \) that will be used to flag array elements that cannot be privatized. As before, the shadow arrays are initialized to zero. Initially, the test assumes that all array elements are privatizable, and if it is found in any iteration that an element is read before it is written, then it is marked as non-privatizable.

In parallel, for each iteration \( i \) of the loop, process all accesses to the shared array \( A \):

(a) Writes: If this is the first write to this array element in this iteration, then set the corresponding element in \( A_w \).

(b) Reads: If this array element is never written in this iteration, then set the corresponding element in \( A_r \). If this array element has not been written in this iteration before this read access, then set the corresponding element in \( A_{np} \), i.e., mark it as non-privatizable.

(c) Count the total number of write accesses to \( A \) that are marked in this iteration, and store the result in \( tw_i(A) \), where \( i \) is the iteration number.
S1: Do i = 1, 8
S2: ... = A[R1[i]] R1[1:8] = [2 2 2 10 8 8 8 10]
S3: A[W[i]] = ... W[1:8] = [1 3 5 4 7 3 6 12]
S4: ... = A[R2[i]] R2[1:8] = [1 3 2 10 7 3 8 12]
S5: Enddo

<table>
<thead>
<tr>
<th>Position in shadow arrays</th>
<th>Written (tw(A))</th>
<th>Counted (tm(A))</th>
</tr>
</thead>
<tbody>
<tr>
<td>(A_w[1:12])</td>
<td>1 0 1 1 1 1 1 0 0 0 0 1</td>
<td></td>
</tr>
<tr>
<td>(A_r[1:12])</td>
<td>0 1 0 0 0 0 0 1 0 1 0 0</td>
<td></td>
</tr>
<tr>
<td>(A_{np}[1:12])</td>
<td>0 1 0 0 0 0 0 1 0 1 0 0</td>
<td></td>
</tr>
<tr>
<td>(A_w[1:12] \land A_r[1:12])</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
<tr>
<td>(A_w[1:12] \land A_{np}[1:12])</td>
<td>0 0 0 0 0 0 0 0 0 0 0 0</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2.3** Example and results for PD test

2. **Analysis Phase.** For each shared array \(A\) under scrutiny:

(a) Compute (i) \(tw(A) = \sum tw_i(A)\), i.e., the total number of writes that were marked by all iterations in the loop, and (ii) \(tm(A) = \text{sum}(A_w[1:s])\), i.e., the total number of marks in \(A_w[1:s]\).

(b) If \(\text{any}(A_w[1:s] \land A_r[1:s])\), i.e., if the marked areas are common anywhere, then the loop is not a doall and the phase ends. (Since we read and write from the same location in different iterations, there is at least one flow or anti dependence.)

(c) Else if \(tw(A) = tm(A)\), then the loop is a doall and the phase ends. (Since we never overwrite any memory location, there are no output dependences.)

(d) Else if \(\text{any}(A_w[1:s] \land A_{np}[1:s])\), then the loop is not a doall and the phase ends. (There is at least one dependence that cannot be removed by privatization).

(e) Otherwise, the loop can be made into a doall by privatizing all elements of the shared array that are written in the loop. (We remove all memory-related dependences by privatizing these array elements.)

In order to illustrate the differences between the doall test and the PD test, we consider the loop shown in Fig. 2.3, which contains memory-related dependences that can be removed by privatization. Assume the loop has 8 iterations, accesses a vector of dimension 12, and that the access pattern is given by the subscript arrays \(R1, R2\) and \(W\). After marking and counting we obtain the results depicted in the table. Since \(A_w[1:12] \land A_r[1:12]\) and \(A_w[1:12] \land A_{np}[1:12]\) are zero everywhere, the loop can be made into a doall, but only after privatization because \(tw(A) \neq tm(A)\).
2.5 Run-time Reduction Parallelization

As mentioned in Section 2.2, there are two tasks required for reduction parallelization: recognizing the reduction variable, and parallelizing the reduction operation. Of these, we focus our attention on the former since, as previously noted, techniques are known for performing reduction operations in parallel. So far the problem of reduction variable recognition has been handled at compile-time by syntactically pattern matching the loop statements with a template of a generic reduction, and then performing a data dependence analysis of the variable under scrutiny to validate it as a reduction variable [Zim91]. There are two major shortcomings of such pattern matching identification methods.

1. The data dependence analysis necessary to qualify a statement as a reduction cannot be performed statically in the presence of input-dependent access patterns.

2. Syntactic pattern matching cannot identify all potential reduction variables (e.g., in the presence of sub scripted subscripts).

For example, in order to parallelize the loop in Figure 2.4 (a) we need to remove the dependences associated with statement S3 by first recognizing that it is a reduction and then proving that the accesses to array A in statements S1 and S2 do not overlap with those in statement S4. Of course we have to also show that statements S1 and S2 do not introduce any other dependence. Because the values stored in the subscript arrays K, L and R are statically unknown, we will have to resort to a run-time method. An example for a case when syntactic pattern techniques are insufficient for flagging all potential reductions in a loop, is shown Fig. 3.3 and will be discussed in Section 3.3.

Below we show how each of these two difficulties can be overcome with a combination of static and run-time methods.

2.5.1 The RPD test: Extending the PD Test for Reduction Validation

In this section we consider the problem of verifying that a statement is a reduction using run-time data dependence analysis. The potential reduction statement is assumed to syntactically pattern match the generic reduction template \( x = x \otimes \text{exp} \); reduction statements that do not meet this criterion are treated in the next section. To verify that such a statement is a reduction we need to check that the reduction variable \( x \) satisfies the definition given in Section 2.2, i.e., that \( x \) is only accessed in the reduction statement, and that it does not appear in \( \text{exp} \).

Our basic strategy is to extend the PD test to check all statically un-verifiable reduction conditions. We first consider how the test would be augmented to check only that the reduction variable
is not accessed outside the single reduction statement. This situation could arise if the reduction variable is an array element accessed through subscripted subscripts and the subscript expressions are not statically analyzable. For example, although statement $S_3$ in the loop in Fig. 2.4(a) matches a reduction statement, it is still necessary to prove that the elements of array $A$ referenced in $S_1$ and $S_2$ do not overlap with those accessed in statement $S_3$, i.e., that: $K(i) \neq R(j)$ and $L(i) \neq R(j)$, for all $1 \leq i, j \leq n$. Thus, the RPD test must check at run-time that there is no intersection between the references in $S_3$ and those in $S_1$ and/or $S_2$; in addition it will be used to prove, as before, that any cross-iteration dependences in $S_1$ and $S_2$ are removed by privatization. To test this new condition we use another shadow array $A_{nx}$ to flag the array elements that are not valid reduction variables. Initially, all array elements are assumed to be valid reduction variables, i.e., $A_{nx}[i] = false$. In the marking phase of the test any array element defined or used outside the reduction statement is invalidated as a reduction variable, i.e., its corresponding element in $A_{nx}$ is set to true.

As before, after the parallel marking phase, the analysis phase of the test is performed. An element of $A$ is a valid reduction variable if and only if it was not invalidated during the marking phase, i.e., it was not marked in $A_{nx}$ as not a reduction variable for any iteration. The other shadow arrays $A_{np}$, $A_w$, and $A_r$ are initialized, marked, and interpreted just as before.

The RPD test can also solve the case when the $exp$ part of the RHS of the reduction statement contains references to the array $A$ that are different from the pattern matched LHS and cannot be statically analyzed. To validate such a statement as a reduction we must show that no reference in $exp$ overlaps with those of the LHS. This is done during the marking phase by setting an element of $A_{nx}$ to true if the corresponding element of $A$ is referenced in $exp$.

In summary, the RPD test is obtained by modifying the PD test. The following step is added to the Marking Phase.

1(d) Write and reads: if a reference to $A$ is not one of the two known references to the reduction variable (i.e., it is outside the reduction statement or it is contained in $exp$), then set the corresponding element of $A_{nx}$ to true (to indicate that the element is not a reduction variable).

(See Fig. 2.4(a) and (b).)

In the Analysis Phase, Steps 2(d) and 2(e) are replaced by the following.

2(d') Else if $any(A_w[i] \land A_{np}[i] \land A_{nx}[i])$, then some element of $A$ written in the loop is neither a reduction variable nor privatizable. Thus, the loop, is not a doall and the phase ends. (There exist iterations (perhaps different) in which an element of $A$ is not a reduction variable, and in which it is used (read) and subsequently modified.)
\begin{figure}[h]
\centering
\begin{subfigure}{0.5\textwidth}
\begin{verbatim}
do i = 1, n
  S1:   A(K(i)) = ........
  S2:   ............ = A(L(i))
  S3:   A(R(i)) = A(R(i)) + exp()
  enddo
\end{verbatim}
\end{subfigure}
\begin{subfigure}{0.5\textwidth}
\begin{verbatim}
do i = 1, n
  S1:   A(S(i)) = A(S(i)) + exp(X(i))
  S2:   A(R(i)) = A(R(i)) + exp()
  enddo
\end{verbatim}
\end{subfigure}

(a)

\begin{subfigure}{0.5\textwidth}
\begin{verbatim}
doall i = 1, n
  markwrite(R(i))
  markredux(K(i))
  markread(L(i))
  markredux(L(i))
  markwrite(R(i))
  enddoall
\end{verbatim}
\end{subfigure}
\begin{subfigure}{0.5\textwidth}
\begin{verbatim}
doall i = 1, n
  markwrite(R(i))
  if (A_nx(R(i)) .ne. 0) then
    markredux(R(i))
  else
    A_nx(R(i)) = '*';
  endif
  markread(X(i))
  markredux(X(i))
  markwrite(S(i))
  if (A_nx(S(i)) .ne. 0) then
    markredux(S(i))
  else
    A_nx(S(i)) = '+';
  endif
  enddoall
\end{verbatim}
\end{subfigure}

(b)

\begin{subfigure}{0.5\textwidth}
\begin{verbatim}
A_nx(:) = 0
doall i = 1, n
  markwrite(R(i))
  if (A_nx(R(i)) .ne. 0) then
    markredux(R(i))
  else
    A_nx(R(i)) = '*';
  endif
  markread(X(i))
  markredux(X(i))
  markwrite(S(i))
  if (A_nx(S(i)) .ne. 0) then
    markredux(S(i))
  else
    A_nx(S(i)) = '+';
  endif
enddoall
\end{verbatim}
\end{subfigure}

(c)

\begin{subfigure}{0.5\textwidth}
\begin{verbatim}
doall i = 1, n
  markwrite(K(i))
  markredux(K(i))
  markread(L(i))
  markredux(L(i))
  markwrite(R(i))
enddoall
\end{verbatim}
\end{subfigure}

(d)

\caption{Example for RPD test. The transformation of the \texttt{do} loops in (a) and (c) is shown in (b) and (d), respectively. The \texttt{markwrite} (\texttt{markread}) operation marks the indicated element in the shadow array $A_w$ ($A_r$ and $A_{np}$) according to the criteria given in Step 1(a) (1(b)) of the PD test. The \texttt{markredux} operation sets the shadow array element of $A_{nx}$ to true. In (d), the type of the reduction is tested by storing the operator in $A_{nx}$.}
\end{figure}

2(e') Otherwise, the loop can be made into a \texttt{doall} by parallelizing reduction operations and privatizing the shared array $A$. (All data dependences are removed by these transformations.)

If the analysis phase validates (passes) the loop as a \texttt{doall} then after its parallel execution, the last-value assignments are performed for any live shared variables, and the scalar result of each reduction is computed using the processors' partial results in a reduction across the processors. (See Fig. 4.1.) (If reductions are implemented by placing the reduction statements in unordered critical sections, then this last step is not necessary.)
2.5.2 Multiple Potential Reduction Statements

A more complicated situation is when the loop contains several reduction statements that refer to the same array $A$. In this case the type of the reduction operation performed on each element must be the same throughout the loop execution, e.g., a variable cannot participate in both a multiplicative and an additive reduction since the resulting operation is not associative and is therefore not parallelizable. The solution to this problem is to mark the shadow array $A_{nx}$ with the reduction type. Whenever a reference in a reduction statement is marked, the current reduction type (e.g., summation, multiplication) is checked with with previous one. If they are not the same, the corresponding shadow element of $A_{nx}$ is set to true.

In Fig 2.4(c) and (d), we show how a loop containing two potential reduction statements with different operators and an $exp$ operand that contains references to the array under test can be transformed to perform a run-time dependence and reduction test. The subsequent analysis of the shadow arrays will detect which elements were used in a reduction and which are privatizable or read-only. If any element is found not to belong to one of these categories, then the speculative parallelization was incorrect and a sequential re-execution must be initiated.

As a final remark, we note that a more aggressive implementation could promote the type of a reduction at run-time: if a memory element is first involved in a \'+\' reduction and then switches over to a \'*\' reduction and stays that way for all the remaining references, then the speculative parallel execution can still yield valid partial results on each processor. It is important to remember that a reduction type can be promoted in only one direction (it cannot be demoted back to its initial type) and only once per loop invocation. Of course, the reduction across processors must reflect the reduction operator promotion.

In the next chapter we will present more aggressive techniques for reduction recognition and run-time validation.

2.6 Implementing the Tests on a Shared Memory Machine

As mentioned above, implementations of the doall tests should be optimized to take advantage of any features available on the target architecture. For example, on shared memory multiprocessor machines, each processor may have some private memory. In this case, contention in the global shared memory can often be reduced if a significant portion of the computation can be performed in the processors' private memories, and then combined to give the final result.
2.6.1 The Inspector Loop

The use of an 'inspector' loop is, as its name says, to inspect the access pattern of a loop, most importantly that of the shared variables. Such a loop, if obtained, will traverse the references to the shared address space without actually performing any computation on the data stored at these locations. In other words, no shared data will be modified so the inspector is side-effect free.

As mentioned in Section 2.2, the RPD test can be applied to the inspector and a decision about the loop's parallelism can be made. Before we elaborate on the principles of inspector extraction we should point out that we consider the requirement that such an 'inspector' loop be parallel (a doall) to be extremely important for providing scalable performance.

Briefly inspector loop extraction requires the compiler to distribute the original loop into two loops. The first one will compute the addresses of the shared array that is under test and the second will use these addresses to access the actual storage and perform the computation. It is important to remark here that if the statements computing the addresses of the shared variables being tested and those that use it are strongly connected in the dependence graph, a proper distribution is not possible [KM90]. This fact limits the applicability (generality) of all 'inspector/executor' run-time techniques. Once the address computing loop has been obtained we can generate the final inspector loop by augmenting the first loop with the shadow structures presented first in Chapter 2 and the whole construct is then transformed for parallel execution.

In the best case, the access pattern of the shared variable is known before entering the loop, e.g., a pre-determined subscript array. In this simple case, the access pattern of the array can be processed in complete isolation from the rest of the loop. There are other cases in which the resulting inspector loop is fully parallel, (the distribution conditions have been satisfied) but contains a significant amount of computation which may have side-effects, e.g., when the access pattern is computed inside the loop itself. For example, loop Actfor_240 from the Perfect code BDNA, first collects in a subscript array the indices of the elements of a much larger data structure that satisfy a certain condition. Subsequently the obtained list of elements will be processed in the loop and accessed through the previously built subscript array. Sometimes it is possible to overcome this disadvantage by passing down the computed values to the second loop. Unfortunately this may have the effect of increasing the working set dramatically.

As a final remark we should mention that efficient inspector loop generation is complex and requires inter-procedural analysis if the parallelization is to be performed at levels with large granularity. The greatest benefits of this technique can be obtained for irregular programs that have an
input dependent but otherwise static distribution. Codes with dynamically changing access pattern can see little benefit, except for small, inner loops.

### 2.6.2 Private Shadow Structures

The doall tests can take advantage of the processors’ private memories by using private shadow structures for the marking phase of the tests. Then, at the conclusion of the private marking phase, the contents of the private shadow structures are merged into the global shadow structure referred to in the description of the tests (Sections 2.3 and 2.4). Note that since the order of the writes (marks) to an element of the shadow structure is not important, all processors can transfer the contents of their private shadow structures to the global shadow structure without synchronization. Also, the transfer from the private to the global shadow structures could be done in vector-concurrent mode if available.

In fact, using shadow structures that are private to each processor enables some additional optimization of the doall tests as follows. Since the shadow structures are private to each processor, the iteration number can be used as the “mark” and assigned to them. In this way, no re-initialization of the shadow structures will be required between successive iterations, and checks such as “is this the first write access to this memory location in this iteration?” simply require checking if the corresponding element in the write shadow structure \( A_w \) is marked with the iteration number. Another benefit of the iteration number “marks” is that they can double as time-stamps, which are needed for performing the last-value assignment to any shared variables that are live after loop termination – unless the processor-wise version of the RPD test (described in Section 2.7.1) is used.

An example which uses private shadow arrays that are marked with the iteration number is shown in Figure 2.6. The original loop (shown on the left) is extracted from loop 140 in subroutine INTGRL of TRFD from the PERFECT Benchmarks. The doall test is applied to the shared array \( X \). In the inspector loop (fully parallel) (shown on the right), writes are marked in the private shadow array \( X_w \) by iteration number. The number of writes marked is recorded in the private variable \( tw_i \). At loop termination, the total number of marks that were attempted is accumulated (with synchronization) into the global variable \( tw(X) \), and the contents of the private shadow arrays are transferred to the global shadow array. Then, the loop is executed appropriately.

### 2.6.3 Hash Tables

Thus far, we have assumed that the shadow structures are structurally equivalent to the original shared variable, and that each processor has its own private copy of the entire shadow structure.
A potential problem with this strategy is that the number of operations in the DO ALL tests would not scale, i.e., although, as the number of processors increases, each processor is responsible for fewer accesses, each processor must still inspect every element of its private shadow structure when transferring it to the global shadow structure. Another related issue is that the resource consumption (memory) would not scale.

To combat these problems we would like a processor to have private shadow copies only of the elements accessed in the iterations assigned to that processor for the RPD test. Unfortunately, the shadow elements needed by a processor will generally not form a contiguous subset of the shadow structure because most of the loops that require run-time analysis have irregular access patterns. However, these issues can be addressed by using hash tables for the shadow structures. Of course, the cost per access would increase since array accesses would be replaced by accesses to a hash table. Thus, there exists a trade-off that should be resolved using a cost model.

### 2.6.4 Allocating the Private Variables

If the PD test determines that privatization can be used to transform the loop into a doall, then there are two choices: privatize the entire shared variable, or only privatize the shared memory locations (e.g., elements of the array) that are written in the loop. If only the elements that are written are privatized, it is possible that this technique might yield performance gains over privatizing the whole array for two reasons: First it may save work in the last value assignment step and second, if the data access pattern is sparse enough, the reduction in the size of the working set could improve the cache performance perhaps to the point of super-linear speedups.

<table>
<thead>
<tr>
<th>A</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>A_w</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Prefix Sums</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>d</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R2</td>
<td>1</td>
<td>3</td>
<td>2</td>
<td>10</td>
<td>7</td>
<td>3</td>
<td>8</td>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>PR2</td>
<td>d+1</td>
<td>d+2</td>
<td>2</td>
<td>10</td>
<td>d+6</td>
<td>d+2</td>
<td>8</td>
<td>d+7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Figure 2.5** Allocating the private variables; privatized elements are shaded.

If it is decided to privatize only the elements that are written, then the private variables can be allocated as follows. Consider the array A from the previous example. The elements of A that are privatized are exactly those elements k for which \( A_w[k] = 1 \). Since \( tm(A) = 7 \), we allocate enough space for seven elements of \( A \); denote this space by \( PA[1 : 7] \). We can determine the positions of
the privatized elements of A in PA from the prefix sums of \( A_w[1:12] \), e.g., the private version of \( A[5] \) is contained in \( PA[4] \) since the prefix sum value of \( A_w[5] = 4 \) (see Figure 2.5).

In general, on each access to a shared array element \( A[k] \), it must be determined whether or not \( A[k] \) has been privatized, e.g., by checking \( A_w[k] \). However, in the case of subscripted subscripts, we can remove the need for this check as follows. Each iteration is provided with a private subscript array (of the same dimension as the original subscript array), and all references to the subscript array use the private version. If \( A[k] \) is not privatized, then references to it in the subscript array remain the same. If \( A[k] \) is privatized, then occurrences of \( k \) in the subscript array are replaced by the prefix sum value in \( A_w[k] \) plus the offset between the starting addresses of \( A \) and \( PA \). The private version \( PR2 \) of the subscript array \( R2 \) is shown in Figure 2.5, where \( d \) is the offset between the starting addresses of \( PA \) and \( A \), i.e., \( d = \&PA[0] - \&A[0] \). Actually, we can handle other cases besides subscripted subscripts in a similar manner by constructing a private subscript array \( PS[1:s] \), and and transforming references such as \( A[i] \) into \( A[PS[i]] \). The values of \( PS \) are set as follows: if \( A[k] \) is not private, then \( PS[k] = k \), and if \( A[k] \) is private, then \( PS[k] = s_k + d \), where \( s_k \) is the prefix sum value in \( A_w[k] \), and \( d \) is again the offset between the starting addresses of \( A \) and \( PA \). Since this technique requires indexing out of bounds, it may cause problems in certain languages and would be best implemented in machine language.

2.7 Variants of the RPD Test

2.7.1 A Processor-wise Version of the RPD Test

The RPD Test, as described in Sections 2.3 and 2.4, determines whether a loop has any cross-iteration data dependences that cannot be removed by privatization. It turns out that essentially the same method can be used to test whether the loop, as executed, has any cross-processor data dependences that cannot be removed by privatization.\(^2\) Assuming each processor has its own private shadow structures as discussed in Section 2.6.2, the only difference is that all checks in the test refer to processors rather than to iterations, i.e., replace “iteration” by “processor” in the description of the RPD test so that all iterations assigned to a processor are considered as one “super–iteration” by the test.

Note that a loop that is not fully parallel could potentially pass the processor-wise version of the RPD test because data dependences among iterations assigned to the same processor will not be detected. However, this is acceptable (even desirable) as long as these iterations are guaranteed

\(^2\)This fact was noted by Santosh Abraham[Abr94].
to be executed on the same processor in the same order. Therefore, if each processor keeps a record of the iterations that it executes during the RPD test, this information could be used to statically schedule the future parallel execution of the loop.

In Section 2.6.2 we noted that re-initialization of the private shadow structures could be avoided if iteration numbers were used for marking in the RPD test. A potential advantage of the processor-wise version of the RPD test is that no re-initialization is needed since we are only interested in cross-processor dependences. Therefore, memory requirements could be reduced by using boolean shadow structures. Of course, if any variable requires a last-value assignment, then it might still be desirable to use iteration numbers for marking.

### 2.7.2 Distinguishing between Fully Independent and Privatizable Accesses

Since privatization generally implies replication of program variables (i.e., an increase in memory requirements), this transformation should be avoided when there is no benefit to be gained. In Section 2.6.4 we noted that the RPD test offers potential gains in this regard over privatizing the entire shared array, i.e., the RPD test can identify and privatize only the elements that are actually written. However, if an element is written only once in the loop, then there is no need for it to be privatized and replicated on multiple processors.

Although the standard version of the RPD test described in Section 2.4 does not determine if an element is written more than once in the loop, it can easily be augmented to provide this information. The simplest approach is to use another shadow structure $A^p_{mw}$ to flag the array elements which have been written multiple times. On a write to an element during the marking phase, the corresponding entry of $A^p_{mw}$ is marked if it is known that that element has been written before (in any iteration), which can be determined by checking the corresponding entry in $A_w$. The global shadow structure $A_{mw}$ is now constructed from the private (per processor) shadow structures $A^p_{mw}$ and $A_w$. First, the marked elements in the private shadow structures $A^p_{mw}$ can be transferred directly, without synchronization, into $A_{mw}$. If the private shadow structures $A_w$ are merged pairwise into the global shadow structure $A_w$, then during this process it can be determined if an element is marked in more than one $A_w$, i.e., if it was written more than once. We note that the need for the pairwise merges can be eliminated if the accesses to the global shadow structure are placed in critical sections. If it is found that the loop can be transformed into a `doall` by privatization, then every element that is written more than once, i.e, with $A^p_{mw} = 1$, is privatized, and those elements that are only written once are not. It is simple to see that the need for the additional structure $A^p_{mw}$ can be eliminated by using three states for the structure $A_{np}$, e.g., negative
values for multiple writes and privatizable, 0 for at most one write and privatizable (initial value), and positive values for not privatizable (once set, never reset).

If the processor-wise version of the PD test is used then the elements that are written more than once can be identified in essentially the same manner. Note that for the processor-wise version it is possible that the number of private variables could be reduced even more since only the processors that actually write the elements need copies, and the private structures identify these elements.

2.7.3 Supporting Copy-in of External Values

Suppose that a loop is determined as fully parallel by the RPD test except for the accesses to one element $a$. If the first time(s) $a$ is accessed it is read, and for every later iteration that accesses $a$ it is always written before it is read, then the loop could actually be executed as a \texttt{doall} by having the initial accesses to a \texttt{copy-in} the global value of $a$, and the iterations that wrote $a$ used private copies of $a$. The PD test can be augmented to detect this situation by keeping track of the maximum iteration $\hat{i}_r^+$ that read $a$ (before it was ever written), and the minimum iteration $\hat{i}_w^-$ that wrote $a$. Then, if $\hat{i}_r^+ \leq \hat{i}_w^-$, the loop can be executed in parallel. In order to collect this information we need two additional private shadow structures, which are merged, pairwise, into the global shadow structure (as in Section 2.7.2).

We can remove the need for these additional shadow structures in a restricted version of the processor-wise RPD test where the iteration space is assigned to the processors in contiguous chunks, i.e., processor $i$ gets iterations $\lceil n/p \rceil \times i$ through $\lceil n/p \rceil \times (i + 1) - 1$, $0 \leq i < p$. Then, we need only check that the first write to $a$ appears in a chunk that is not less than the last chunk in which $a$ is marked as non-privatizable. The potential disadvantage of this chunking method is that the assignment of iterations to processors may produce an imbalanced workload.
**Original Version**

```fortran
**Extracted Loop for Marking**
integer pX_w(0:numproc-1,:,:), ptw(0:numproc-1)
integer X_w(,:,:), tw, tm
DOALL I = 1,NP
C **-- private variables**
integer IJ, MAXL, J, K, L, KL
C **-- do once per processor**
ptw(my_proc) = 0
LOOP ** do once per iteration (I)**
DO J=1,I
IJ=IA(I)+J
DO K=1,I
MAXL=K
IF(K,EQ,I) MAXL=J
DO L=1,MAXL
KL=IA(K)+L
C **mark shadow array X_w (if not already marked)**
IF (pX_w(my_proc,IJ,KL),NE, I) THEN
pX_w(my_proc,IJ,KL) = I
ptw(my_proc) = ptw(my_proc) + 1
ENDIF
ENDDO
ENDDO
ENDDOALL
C **transfer private arrays to global shadow array**
DOALL i=1,nproc
   if (pX_w(i,:,:).ne. 0) X_w(:, :) = pX_w(i, :, :)
ENDDO
C **accumulate global number of marks attempted**
tw = sum(pw(0:numproc-1))
C **accumulate number of marks in global shadow array**
tm = sum(X_w(:, :))
C **execute loop appropriately**
IF (tw.eq. tm) THEN
   execute loop as a DOALL
ELSE
   execute loop sequentially
```

**Figure 2.6** PD test example from TRFD. The original data accesses are replaced by accesses to the shadow variables for the marking phase of the doall test. This loop is extracted from loop 140 in subroutine INTGRAL of TRFD from the PERFECT Benchmarks. The doall test is applied to the shared array X. In the inspector loop (right), writes are marked in the shadow array X_w by iteration number, so the shadow array can be reused. The number of writes marked is recorded in the private variable tw_w.
Chapter 3

Speculative Run–Time Parallelization of Loops: The LRPD Test

3.1 Introduction

In Chapter 2 we have presented a general method for detecting fully parallel loops when the two most important transformations are applied. This method is centered around the possibility of extracting an inspector loop that analyzes the data access pattern “off-line,” i.e., without side effects [BS90, LZ93, MP87, RP94a, SM91, SMC89, SMC91, WSHB91, ZY87, CYT94]. Unfortunately the extraction of an inspector loop that can traverse the access pattern without actually having to perform the data computation is often not possible: if the address computation of the array under test depends on the actual data computation, as exemplified by Fig. 2.1(a), then the inspector becomes both computationally expensive and has side-effects. This means that shared arrays would be modified during the execution of the inspector loop and saving the state of these variables would be required – making the inspector equivalent to the loop itself. In our opinion, the desirable goal of exploiting coarse-grain parallelization, i.e., at the level of large complex loops, makes it even less likely that an appropriate “inspector” loop can be extracted. Thus, the inspector/executor approach is not a generally applicable method, i.e., it is limited to special cases.

3.1.1 A New Approach: Speculative doall Parallelization

In this chapter we propose a novel framework for parallelizing do loops at run-time. The proposed framework differs conceptually from previously presented method. Instead of distributing the loop into inspector and executor loops, we speculatively execute the loop as a doall, i.e., execute all its iterations concurrently, and apply a run-time test to check if there were any cross-iteration dependences. If the run-time test fails, then we will pay a penalty in that we need to backtrack and re-execute the loop serially. This new method yields several advantages: first, in this approach the operations associated with the actual computation and those associated with test are interleaved within each of the iterations of the loop, i.e., the memory access pattern does not need to be extracted and analyzed separately as in inspector/executor methods. In this way we can apply the test to any loop, even if a parallel inspector loop cannot be found.

Second, the new algorithm considers only data dependences caused by actual cross-iteration data-flow (a flow of values). Thus, they may potentially qualify more loops as parallel than the method in Chapter 2 which conservatively considered the dependences due to every memory refer-
ence—even if no cross-iteration data-flow occurred at run-time. Such a situation is shown in Fig. 3.2. Element $A(K(i))$ is always read but only conditionally used in the computation of $A(L(i))$. If a read variable is not used, then it does not contribute to the global data flow and should therefore not be allowed to inhibit parallelization. We will show in Section 3.2.1 how the new method finds the array $A$ privatizable while the RPD test described in Chapter 2 cannot.

Third, we extend the technique for reduction recognition introduced in Chapter 2: in contrast to the static pattern matching and run-time validation techniques employed so far, the new method detects if the values stored in an array participate in a reduction operation, even if they are transferred through private variables and/or are affected by statically unpredictable control flow.

### 3.2 Speculative Parallel Execution of Do Loops

Consider a do loop for which the compiler cannot statically determine the access pattern of a shared array $A$ that is referenced in the loop. Instead of generating pessimistic, sequential code when it cannot unequivocally decide whether the loop is parallel, the compiler could decide to *speculatively* execute the loop as a doall, and produce code to determine at run-time whether the loop was in fact fully parallel. In addition, if it is suspected that some data dependences could be removed by privatization and/or reduction parallelization the compiler may further speculatively apply these transformations in order to increase the chances that the loop can be executed as a doall. If the subsequent run-time test finds that the loop was not fully parallel, then it will be re-executed sequentially. In order to speculatively parallelize a do loop as outlined above we need the following:

- A mechanism of saving/restoring state: to save the original values of the program variables for the possible sequential re-execution of the loop.
- An error (hazard) detection method: to test the validity of the speculative parallel execution.
- An automatable strategy: to decide when to use speculative parallel execution.

**Saving/Restoring State.** There are several ways to maintain backups of the program variables that may be altered by the speculative parallel execution. If the resources (time and space) needed to create a backup copy are not too big, then a practical solution is checkpointing prior to the speculative execution. It might be possible to reduce this cost by identifying and checkpointing a point of minimum state in the program prior to the speculative parallel execution. A more attractive solution is to privatize all shared variables, copy-in (on demand) any needed external values, and copy-out any live values if the test passes, thereby committing the results computed
by the doall loop. This method could also yield better data locality and reduce the number of messages between processors (e.g., it would generate less coherency traffic in a cache coherent shared memory machine). Note that privatized arrays need not be backed up because the original version of the array will not be altered during the parallel execution.

**Hazard Detection.** There are essentially two types of errors (hazards) that could occur during the speculative parallel execution: (i) exceptions and (ii) the presence of cross-iteration dependences in the loop. A simple way to deal with exceptions is to treat them as an invalid parallel execution, i.e., if an exception occurs, abandon the parallel execution, clear the exception flag, restore the values of any altered program variables, and execute the loop sequentially. Below, we present techniques that can be used to detect the presence of cross-iteration dependences in the loop and to test the validity of any privatization and/or reduction parallelization transformations that were applied.

**An Automatable Strategy.** The main factors that the compiler should consider when deciding whether to speculatively parallelize a loop are: the probability that the loop is a doall, the speedup obtained if the loop is a doall, and the slowdown incurred if the loop is not a doall. For example, the compiler might base its decision on a ratio of the estimated run-time cost of an erroneous parallel execution to the estimated run-time cost of a sequential execution. If this ratio is small, then significant performance gains would result from a successful (valid) parallelization of the loop, at the risk of increasing the sequential execution time by only a small amount. In order to perform a cost/benefit analysis and to predict the parallelism of the loop, the compiler should use static analysis and run-time statistics (collected on previous executions of the loop or from different codes); in addition, directives about the parallelism of the loop might prove useful. In Section 4.2 a complexity analysis of our run-time tests is presented that can be used to statically predict the minimum obtainable speedup and the maximum potential slowdown for a loop parallelized using our techniques.

### 3.2.1 The LRPD Test

The test described in this section represents an improvement over the RPD test presented in Section 2. As mentioned before, we will apply it speculatively, i.e., we will mark the shadow arrays in the same iteration with the computation of the original loop. Additionally, we will change the relative position of the markread code to the statements containing a read to the shared array under test. We will postpone the markread operation until the value read from the shared array is actually used and contributes to the global dataflow of the loop. This new placement of the marking
code is accomplished through a technique named *dynamic dead reference elimination* which will be described below.

### 3.2.1.1 Dynamic dead reference elimination

An important source of ambiguity that cannot be analyzed statically and potentially generates overly conservative data dependence models is the run-time equivalent of *dead code*. A simple example (Fig. 3.2) is when a loop first reads a shared array element, \(A(K(i))\), into a local variable but then uses it only conditionally in the computation of other shared variables. If the consumption of the read value does not materialize at run-time, then the read access did not in fact contribute to the data flow of the loop and therefore could not have caused a dependence. Since predicates seldom can be evaluated statically, the compiler must be conservative and conclude that the read access causes a dependence in every iteration of the loop.

More formally, the references we want to identify are defined as follows. If a value stored in a shared memory location is read into a private variable and only conditionally used then it may not introduce a data dependence if the use does not materialize at run-time. In order to recognize statically which read references may not affect the data flow at run-time we first define a dynamic dead read reference:

**Definition** A *dynamic dead read reference* is a read access of a shared variable that both

(a) does not contribute to the computation of any other shared variable, and

(b) does not control (predicate) the references to other shared variables.

The value obtained through a dynamic dead read does not contribute to the data flow of the loop. Ideally, such accesses should not introduce false dependences in either the static or the run-time dependence analysis. If it is possible to determine the dead references at compile time then we can just ignore them in our analysis. Since this is generally not possible (control flow could be input dependent) the compiler should identify the references that have the potential to be unused and insert code to solve this problem at run-time. In Fig. 3.2 we give an example where the compiler can identify such a situation by following the def–use chain built by using array names only. To avoid introducing false dependences, the marking of the read shadow array is postponed until the value that is read into the loop space is indeed used in the computation of other shared variables. In essence we are concerned with the flow of the values stored rather than with their storage (addresses). We note that if the search for the actual use of a read value becomes too complex then it can be stopped gracefully at a certain depth and a conservative marking of the shadow array can be inserted (on all the paths leading to a possible use).
3.2.1.2 The Lazy (value-based) Reduction Privatizing Doall Test (LRPD Test)

The most general version of the test, as applied to a privatized shared array \( A \), is given below, i.e., it tests for all types of dependences, and also whether the array is indeed privatizable or participates in a reduction. If some of these conditions do not need to be verified, then the test can be simplified in a straightforward manner, e.g., if the array was not privatized for the speculative parallel execution, then all steps pertaining to the privatization check are omitted.

\[
\begin{align*}
\text{do } i &= 1, n \\
\text{S1: } A(K(i)) &= \ldots \ldots \\
\text{S2: } \ldots \ldots &= A(L(i)) \\
\text{S3: } A(R(i)) &= A(R(i)) + \exp() \\
\text{enddo}
\end{align*}
\]

(a)

\[
\begin{align*}
\text{do } i &= 1, n \\
\text{S1: } A(S(i)) &= A(S(i)) + \exp(X(i)) \\
\text{S2: } A(R(i)) &= A(R(i)) + \exp() \\
\text{enddo}
\end{align*}
\]

(b)

\[
\begin{align*}
\text{doall } i &= 1, n \\
\text{markwrite}(R(i)) \\
\text{if } (\text{A\_nx}(R(i)) .ne. 0) \text{ then} \\
\text{markredux}(R(i)) \\
\text{else} \\
\text{A\_nx}(R(i)) &= 'x' \\
\text{endif}
\end{align*}
\]

(c)

\[
\begin{align*}
\text{doall } i &= 1, n \\
\text{markwrite}(K(i)) \\
\text{markredux}(K(i)) \\
\text{markwrite}(L(i)) \\
\text{markredux}(L(i)) \\
\text{S1: } A(K(i)) &= \ldots \ldots \\
\text{markread}(L(i)) \\
\text{mark-redux}(R(i)) \\
\text{mark-redux}(K(i)) \\
\text{S2: } A(R(i)) &= A(R(i)) + \exp() \\
\text{enddoall}
\end{align*}
\]

(d)

**Figure 3.1** Example for LRPD test. The transformation of the do loops in (a) and (c) is shown in (b) and (d), respectively. The **markwrite** (markread) operation marks the indicated element in the shadow array \( A_w \) (\( A_r \) and \( A_{np} \)) according to the criteria given in Step 1(a) (1(b)) of the LRPD test. The **markredux** operation sets the shadow array element of \( A_{\_nx} \) to true. In (d), the type of the reduction is tested by storing the operator in \( A_{\_nx} \).
The LRPD Test

1. **Marking Phase.** (Performed during the speculative parallel execution of the loop.) For each shared array \( A[1 : s] \) whose dependences cannot be determined at compile time, we declare read and write shadow arrays, \( A_r[1 : s] \) and \( A_w[1 : s] \), respectively. In addition, we declare a shadow array \( A_{np}[1 : s] \) that will be used to flag array elements that cannot be validly privatized. Initially, the test assumes that all array elements are privatizable, and if it is found in any iteration that the value of an element is used (read) before it is redefined (written), then it will be marked as not privatizable. The shadow arrays \( A_r, A_w, \) and \( A_{np} \) are initialized to zero.

During each iteration of the loop, all definitions or uses of the values stored in the shared array \( A \) are processed:

(a) Definitions (done when the value is written): if this is the first modification (write) of the value stored in this array element in this iteration, then set the corresponding element in \( A_w \).

(b) Uses (done when the value that was read is used): if this array element is never modified (written) in this iteration, then set the corresponding element in \( A_r \). If the value stored in this array element has not been written in this iteration before this use (read access), then set the corresponding element in \( A_{np} \), i.e., mark it as not privatizable.

(c) Count the total number of write accesses to \( A \) that are marked in this iteration, and store the result in \( tw_i(A) \), where \( i \) is the iteration number.

(d) Definitions and uses: if a reference to \( A \) is not one of the two known references to the reduction variable (i.e., it is outside the reduction statement or it is contained in \( exp \)), then set the corresponding element of \( A_{nx} \) to true (to indicate that the element is not a reduction variable). (See Fig. 3.1(a) and (b).)

2. **Analysis Phase.** (Performed after the speculative parallel execution.) For each shared array \( A \) under scrutiny:

(a) Compute (i) \( tw(A) = \sum tw_i(A), \) i.e., the total number of definitions (writes) that were marked by all iterations in the loop, and (ii) \( tm(A) = \text{sum}(A_w[1 : s]), \) i.e., the total number of marks in \( A_w[1 : s] \).
(b) If \( \text{any}(A_w[i] \land A_r[i]) \), i.e., if the marked areas are common anywhere, then the loop is not a doall and the phase ends. (Since we define (write) and use (read, but do not define) values stored at the same location in different iterations, there is at least one flow or anti dependence.)

(c) Else if \( tw(A) = tm(A) \), then the loop is a doall (without privatizing the array \( A \)). (Since we never overwrite any memory location, there are no output dependences.)

(d) Else if \( \text{any}(A_w[i] \land A_{np}[i] \land A_{nx}[i]) \), then some element of \( A \) written in the loop is neither a reduction variable nor privatizable. Thus, the loop, as executed, is not a doall and the phase ends. (There exist iterations (perhaps different) in which an element of \( A \) is not a reduction variable, and in which it is used (read) and subsequently modified.)

(e) Otherwise, the loop was made into a doall by parallelizing reduction operations and privatizing the shared array \( A \). (All data dependences are removed by these transformations.)

As can be observed from the example in Fig. 3.2, this method allows the LRPD test to qualify more loops for parallel execution then would be otherwise possible by just inspecting the memory references as in the original RPD test described in Chapter 2. In particular, after marking and counting we obtain the results depicted in the tables. The loop fails the RPD test since \( A_w(;i) \land A_r(;i) \) is not zero everywhere (Step 2(b)). However, the loop passes the LRPD test as \( A_w(;i) \land A_r(;i) \) is zero everywhere, but only after privatization, since \( tw(A) \neq tm(A) \) and \( A_w(;i) \land A_{np}(i) \) is zero everywhere.

In conclusion, the two conceptual differences between the LRPD test and the RPD test presented in Chapter 2 are:

(a) The **Speculative strategy** which allows the application of the test to any loop regardless of the existence of a parallel inspector.

(b) The **value based approach** which is more concerned with data dependences caused by the actual, cross-iteration flow of values stored in the shared arrays rather than the sharing of addresses.

The net result is a more realistic (less conservative) evaluations of parallelism.

\[\text{any} \text{ returns the "OR" of its vector operand's elements, i.e., } \text{any}(v[1 : n]) = (v[1] \lor v[2] \lor \ldots \lor v[n]).\]
\[
\begin{align*}
\text{do } & i=1, 5 \\
& z = A(K(i)) \\
& \text{if } (B1(i) \text{ eq. true.}) \text{ then} \\
& A(L(i)) = z + C(i) \\
& \text{endif} \\
\text{enddo}
\end{align*}
\]

\[
\begin{align*}
\text{do } & i=1, 5 \\
& \text{markwrite(K(i))} \\
& z = A(K(i)) \\
& \text{if } (B1(i) \text{ eq. true.}) \text{ then} \\
& \text{markwrite(L(i))} \\
& A(L(i)) = z + C(i) \\
& \text{endif} \\
\text{enddo}
\end{align*}
\]

\[
\begin{align*}
\text{do } & i=1, 5 \\
& z = A(K(i)) \\
& \text{if } (B1(i) \text{ eq. true.}) \text{ then} \\
& \text{markwrite(K(i))} \\
& \text{markwrite(L(i))} \\
& A(L(i)) = z + C(i) \\
& \text{endif} \\
\text{enddo}
\end{align*}
\]

\[
\begin{align*}
B1(1:5) = (1 0 1 0 1) \\
K(1:5) = (1 2 3 4 1) \\
L(1:5) = (2 2 4 4 2)
\end{align*}
\]

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
original \hspace{1cm} & & & & \\
PD test & 1 & 2 & 3 & 4 & \hspace{1cm} tw & tm \\
\hline
\text{A}_w & 0 & 1 & 0 & 1 & 3 & 2 \\
\text{A}_r & 1 & 1 & 1 & 1 & \\
\text{A}_{np} & 1 & 1 & 1 & 1 & \\
\text{A}_w() \land \text{A}_r() & 0 & 1 & 0 & 1 & \\
\text{A}_w() \land \text{A}_{np}() & 0 & 1 & 0 & 1 & \\
\hline
\end{tabular}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|c|}
\hline
\text{new} \hspace{1cm} & & & & \\
LPD test & 1 & 2 & 3 & 4 & \hspace{1cm} tw & tm \\
\hline
\text{A}_w & 0 & 1 & 0 & 1 & 3 & 2 \\
\text{A}_r & 1 & 0 & 1 & 0 & \\
\text{A}_{np} & 1 & 0 & 1 & 0 & \\
\text{A}_w() \land \text{A}_r() & 0 & 0 & 0 & 0 & \\
\text{A}_w() \land \text{A}_{np}() & 0 & 0 & 0 & 0 & \\
\hline
\end{tabular}
\end{table}

Figure 3.2 Example for LPD test. The transformation of a do loop (a), using the original version of the PD test (b), and the lazy version (c). The markwrite (markread) operation marks the indicated element in the shadow array \(A_w\) (\(A_r\) and \(A_{np}\)) according to the criteria given in Step 1(a) (1(b)) of the LPD test. Since dynamic dead read references are not marked in the LPD test, the array \(A\) fails the PD test and passes the LPD test, as shown in (d) and (e), respectively.

3.3 Run–Time Reduction Recognition

In Section 2.5 we have shown how to verify at run-time syntactically matched reductions for arrays indexed by expressions that are statically not analyzable. But pattern matching alone cannot identify all reduction operations in a loop and therefore cannot exploit its parallelism. In this chapter we will take a more general approach and try to identify if the values stored in certain shared variables actually participate in a reduction operation, regardless of whether there are statements that 'look' like a reduction or not. Briefly, the idea is to follow the flow of values that are stored (written) in the shared arrays within the scope of the loop body (but not across iterations) and flag all potential (that can not be proven otherwise by the compiler) reduction statements. Then the run-time test will verify which array elements (variables) have indeed participated in a reduction, just as described in Chapter 2.

In the next sections we will give a step by step description of the compiler techniques needed to identify potential reductions and the marking code necessary for their run-time validation.
3.3.1 Static Reduction Recognition and Run–Time Check

A first example when syntactic pattern matching will fail to identify a reduction whenever all
the references on the RHS of the assignment “look different” from the reference on the LHS. Thus,
if a statement is in fact a reduction, but the references on the LHS and/or the RHS are indirect,
then syntactic pattern matching will fail. This situation could arise naturally, e.g., through the use
of temporary variables or subscripted subscripts. In the latter case, it can only be determined at
run-time if any of the array elements are reduction variables.

In the following we show that a combination of static and run-time techniques can be used to
successfully identify several types of potential reductions that could not be recognized with pattern
matching techniques. The general strategy is to speculate that every assignment to the array of
interest is a potential reduction, unless proven otherwise statically. At run-time this assumption is
then validated or invalidated on an element by element basis.

3.3.2 Single Statement Reduction Recognition

We first consider a single statement in which the references on the RHS are either dependent
on the array A (also referenced on the LHS) or are to values known to be independent of A, e.g.,
constants, loop invariants, or distinct global variables.

The simplest case is when the RHS contains exactly one reference to A. Consider the potential
reduction statement \( A(R(i)) = A(X(i)) + exp \). If \( R(i) = X(i) \), for some values of \( i \), and \( exp \)
is not a function of \( R(i) \), then the probability that the surrounding loop is parallel is increased. In
this case, the solution is simply to check this equality condition at run-time, and mark the shadow
array \( A_{nx} \) accordingly.

The situation is a bit more complex when the RHS contains multiple references to the array
A. Consider the statement \( A(R(i)) = A(X_1(i)) + A(X_2(i)) + \ldots + A(X_k(i)) \). This statement is a
reduction if and only if \( R(i) = X_j(i) \) for exactly one value of \( j \) (see Section 2.2). As the operation
is commutative and associative, we cannot discount the possibility of a reduction. In this example,
we must check for equality between \( R(i) \) and every \( X_j(i) \), \( 1 \leq j \leq k \). If this equality condition is
not met exactly once, then \( A_{nx}(R(i)) \) is set to true (to indicate it was not a reduction). We note
that a more aggressive strategy could be taken when there are multiple references to \( A(R(i)) \) on
the RHS: promote the ‘+’ reduction to a ‘*’ reduction. However, as mentioned in Section 2.5.1, the
reduction type can only be promoted once in the entire loop. Fig. 3.3 shows the code generated
for run-time validation when the RHS contains multiple references to A. In the interest of clarity,
reduction type promotion is not shown.
Figure 3.3 Example for single statement reduction recognition. The code generated for the do loop in (a) is shown in (c). In (c), the procedure in (b) is called. The mark-x operations are as described in Fig. 2.4.

3.3.3 Multiple Statement Reduction Recognition: Expanded Reduction Statements

We now relax all restrictions on the RHS and allow in it variables that are neither explicit functions of the array appearing on the LHS nor explicit loop invariants. Our goal is to uncover any possible link between the LHS and the RHS, if indeed one exists. The general strategy of our methods is a fairly straightforward demand driven forward substitution of all the variables on the RHS, a process by which all control flow dependences are substituted by data dependences as described in [AKPW83, TP93]. Once this expression of the RHS is obtained it can be analyzed and validated by the methods described in the previous section. In the following we explain by way of example how our new method can identify reductions by performing in essence a value-based rather than a dependence-based analysis.

In Fig. 3.4(a) statement S3 is first labeled at compile time as a potential reduction. Then, by following the def-use chains of the variables on the RHS (i.e., z and y) within the scope of the loop we find that in statement S1 z may potentially carry the value of $A(R(i))$, while y is a
Figure 3.4 Example for multiple statement reduction recognition. The code generated for the do loop in (a) is shown in (b). The mark-x operations are as described in Fig. 2.4.

constant with respect to A. The algorithm then examines statement S3 after forward substitution, but does not actually replace S3 in the generated code. The substitution is done only for compiler analysis purposes. This new version of S3, referred to as S33, is of the form: S33 : A(R(i)) = A(K(i)) + constant. Similarly, S5 becomes S55 : A(L(i)) = A(K(i)) + constant. Next, we label the statement pairs (S1, S3) and (S1, S5) in the original loop as expanded reduction statements (ERSs). If we treat each ERS as a single reduction statement, then this problem is reduced to one treated above.

The code generated for the run time marking of the ERS is inserted for both sides of the statement (RHS and LHS), but only in the same basic block as the LHS. As we will see in a later example, this rule insures that both sides are marked when and if there is an assignment, i.e., it insures that a value is actually passed from the RHS to LHS. Any uses of values participating in the reduction that occur outside the ERS invalidate the ERS, i.e., set the corresponding element of the shadow array An to true. In the case of ERSs obtained through forward substitution, the value of the reduction reference may pass through several memory locations (intermediate variables) before reaching the statement of the LHS. As any use of an intermediate variable represents a use of a value
that participates in the reduction, it invalidates the reduction for the corresponding element of \( A \).

The uses can be obtained by following the def-use chain within the scope of the loop. However, based on the dead reference elimination principle described in Section 3.2.1, only those uses that contribute to the actual data-flow of the loop (when the value is passed on to a shared variable or controls the access to a shared variable) are processed. If not all local variables carrying the reduction value end up being used in the global data-flow within the loop, then we have either to verify that they (the local variables) are indeed not live after loop exit, or, if that is not possible, make a conservative assumption (i.e., that all uses contribute to the data flow). In Fig. 3.4(a), statement \( S_4 \) passes the value of \( A(K(i)) \) to the local variable \( t \), which in turn passes it to \( A(L(i)) \) in \( S_5 \). The same value is also passed to the shared variable \( B(f(i)) \) in \( S_6 \). Both uses (in \( S_5 \) and \( S_6 \)) should, in principle, invalidate \( A_{\text{nr}}(K(i)) \). On the other hand, statement \( S_5 \) is another potential reduction of the same type as in \( S_3 \) and, thus only the use in \( S_6 \) needs to invalidate \( A_{\text{nr}}(K(i)) \). The transformed code is shown in Fig. 3.4(b).

We note that if one of the intermediate variables is itself an array element addressed indirectly, then an additional run-time test must be performed. For example, if \( S_1 \) and \( S_3 \) in Fig. 3.4(a) were of the form: \( S_1 : X(N(i)) = A(K(i)) \) and \( S_3 : A(R(i)) = X(P(i)) + y \), then a value would be passed from \( S_1 \) to \( S_3 \) only if \( N(i) = P(i) \). However, if the array \( X \) is privatizable, and occurs only in these two statements, then the run-time test is not necessary, i.e., if \( N(i) = P(i) \), then \( A(K(i)) \) would be processed with the read of \( X(P(i)) \) in \( S_3 \), and otherwise no data flow would occur.

**Taking control flow into account.** The final situation we consider is when the forward substitution procedure must take into account conditional branches and carry information into the expression of the ERS (see Fig. 3.5). The additional difficulty presented by this case is the fact that the exact form of the RHS is not known statically. What is known, however, is the set of all possible RHS forms, which can be computed by following all potential paths in the control flow graph. A direct approach uses a *gated static single assignment* (GSSA) [BMO90, TP94] representation of the program. In such a representation, scalar variables are assigned only once. At the points of confluence of conditional branches a \( \phi \) function of the form \( \phi(B, X_1, X_2) \) is used (in the GSSA representation) to select one of the two possible definitions of a variable (\( X_1 \) or \( X_2 \)), depending on the boolean expression \( B \). By proceeding backwards through the def-use chains (which include the \( \phi \) functions) it is easy to expand a scalar variable in terms of boolean expressions, other scalar variables, and array elements. In the example of Fig. 3.5, the variable \( w \) in statement \( S_9 \) would be expanded as follows:
\[ w \Rightarrow \phi(B3, t, A(M(i))) \]  
\[ \Rightarrow \phi(B3, \phi(B2, z, A(J(i)), A(M(i)))) \]  
\[ \Rightarrow \phi(B3, \phi(B2, A(K(i)), A(L(i)), A(J(i)), A(M(i)))) \]  

which means that the value of \( w \) is:

\[
\begin{align*}
  w &= \begin{cases} 
    A(K(i)) & \text{if } (B3 \land B2 \land B1) \text{ is true} \\
    A(L(i)) & \text{if } (B3 \land B2 \land \neg B1) \text{ is true} \\
    A(J(i)) & \text{if } (B3 \land \neg B2) \text{ is true} \\
    A(M(i)) & \text{if } (\neg B3) \text{ is true}
  \end{cases}
\end{align*}
\]  

This compound equation can then be used to generate a \texttt{markread} and a \texttt{markredux} operation at statement \$9$ where \( w \) is read. To save unnecessary work, we only expand those scalars that are on the RHS of assignments to shared variables or in potential reduction statements (e.g., in the case of \( z \) in statement \$8$). All other scalar references can be safely ignored. Fig. 3.5(b) shows the program in Fig. 3.5(a) after the insertion of the \texttt{markread} and \texttt{markredux} operations, which are based on the expansion of the scalar variables. The possible drawback of this approach is that the number of potential reductions and the number of terms in the logic expressions generated may be quite large. If this happens, we can gracefully degrade to a more conservative approach: test only some of the expressions of the ERS and invalidate all the rest.

It is important to note that the loop in Fig. 3.5 exemplifies the type of loop found in the SPICE2G6 program (subroutine \texttt{LOAD}) which can account for 70\% of the sequential execution time (Its vectorization has been studied before in [Vla82]).

Finally we mention that reductions such as \texttt{min}, \texttt{max}, \texttt{etc.}, would first have to be syntactically pattern matched, and then substituted by the \texttt{min} and \texttt{max} functions. From this perspective, they are more difficult to recognize than simpler arithmetic reductions. However, after this transformation, our techniques can be applied as described above.

### 3.3.4 Applicability of the Run–Time Reduction Algorithm

In this chapter we have presented a fairly powerful and complete technique that can detect if a variable participates in a reduction operation or not. We think that at this point a word of caution against the its indiscriminate use, is warranted. Extracting the Expanded Reduction Statement from a complex loop can be quite compute intensive for the compiler and can potentially yield
doall  i = 1, n
S1:    w = A(M(i))
S2:    t = A(J(i))
S3:    if  (B1) then
S4:        z = A(K(i))
S5:        else
S6:            z = A(L(i))
S7:        endif
S8:    if  (B2) t = z
S9:    if  (B3) w = t
enddoall
(a)

Figure 3.5 Example for expanded reduction statement (ERS). The code generated for the do loop in (a) is shown in (b). The mark-x operations are as described in Fig. 2.4. The expressions in the markread and markredux operations are abbreviations of if then else statements representing the different assignments to z (S8) and w (S9) as in Equation 3.5. The operators “*”, “+”, and “not” represent logical “and”, “or”, and “complement” operators, respectively.

very long expressions that have to be evaluated at run time. If the loops under test turn out to be indeed parallel then, in most cases, speedups can be obtained. However this effort has to be weighted against the probability that an expression that does not ‘look’ like a reduction is indeed a reduction. This trade-off can be handled at compile time by ‘gracefully’ narrowing the scope (reach) of the static analysis – while obtaining potentially conservative results.

Another factor that can impact on the effectiveness of the presented methods is the order of restructuring transformations applied to the code. For example, in the case of a wrap-around variable potentially involved in a reduction, our technique would fail to parallelize the loop. However, if it would be preceded by the appropriate transformation, then successful parallelization can be obtained.
Chapter 4

Complexity, Performance Prediction and Implementation Strategies

4.1 Analysis Before Run-Time

In the previous chapters we have described in some detail the general algorithms through which we can detect fully parallelizable loops at run-time. As we have seen there are many variants of the doall test and they can be applied using different strategies. In order to maximize the speedup obtained from the parallelization of an application it is necessary for the compiler to analyze the code and make the most appropriate decisions with regard to the use of run-time techniques. The compiler needs to evaluate the intrinsic potential speedup of a loop needing run-time analysis (as a function of granularity, number of iterations, work balance, etc.) and then predict the actual performance for the case when these techniques are indeed used.

First we will present the complexity of our algorithms and based on it find a lower bound of the expected speedup. The performance can be predicted with higher accuracy and better results can be obtained if the specifics of the problem are statically analyzed. For example, from the static analysis of an input dependent access pattern it is possible to show that a variable can not be privatizable (If it is always read before it is written) and therefore the doall test should be simplified accordingly.

Another decision that has to be made through analysis, is which of the flavors of the previously described test, inspector/executor or speculative to apply. This choice, as we will describe in the next sections, will depend on many factors (one of them being the possibility of extracting an inspector loop) and has the goal of minimizing overall execution time.

Finally, this chapter will conclude by briefly addressing the issue of the integration of run-time methods into the overall compilation process. We will adopt a hybrid approach in which the static analysis will extract as much information as possible and then transfer it to the run-time system in order to minimize its overhead.

4.2 Complexity of the Doall Tests

The analysis that follows will assume, for simplicity, that the loop to be tested for parallelism does not have any other work except the data movement of the shadowed array elements. We
can assume that the loop can be balanced on the $p$ processors through an appropriate scheduling policy and that there are more iterations than processors, i.e., We will show in Section 4.3 that this situation represents a lower bound on the expected speedup.

We now show that the time required by the doall tests is $T(n, s, a, p) = O(na/p + \log p)$ where $p$ is the number of processors, $n$ is the total iteration count of the loop, $s$ is the number of elements in the shared array, and $a$ is the (maximum) number of accesses to the shared array in a single iteration of the loop. We assume that the implementations of the tests have been optimized as discussed in Section 2.6, i.e., the tests use private shadow structures and, if necessary, hash tables. The analysis below is also valid for the variants of the RPD test discussed in Section 2.7.

The marking phase (Step 1) takes $O(na/p + s + \log p)$ time, i.e., proportional to $\max(na/p, s, \log p)$ time. We record the read and write accesses, and the privatization flags in private shadow arrays. In order to check whether for a read of an element there is a write in the same iteration, we simply check that element in the shadow array — a constant time operation. All accesses can be processed in $O(na/p)$ time, since each processor will be responsible for $O(na/p)$ accesses. If the test is performed speculatively and the loop is heavily imbalanced then a processor may become responsible for almost all accesses and the complexity of the marking phase could become $O(na)$. This case can occur only when there is a lot of 'non-shadowed work' in the loop and represents, as we will show in Section 4.3, a more advantageous situation for obtaining speedups.

After all accesses have been marked in private storage, the private shadow arrays can be merged into the global shadow arrays in $O(s + \log p)$ time; the $\log p$ contribution arises from the possible write conflicts in global storage that could be resolved using software or hardware combining. When using a variant of the test that requires pairwise merging of the private shadow arrays into the global shadow array, we can also perform the merge in $O(s + \log p)$ time, i.e., in each subsequent merge we use twice as many processors so the time is $O(s(\frac{1}{2^p} + \frac{1}{2^p} + \ldots + \frac{1}{2^np})) = O(s)$. In this case the $\log p$ contribution is due to the $\log p$ stages of the merge, and there will be no write conflicts as was possible in the one stage merge.

If $s \gg na/p$, then the time required to merge the private shadow arrays into the global shadow arrays will dominate the time required for the actual marking. As mentioned in Section 2.6.3, this can be avoided by using private hash tables of size $O(na/p)$ instead of the private shadow arrays. The hash tables can be transferred to the global shadow arrays in $O(na/p + \log p)$ time, and the check needed to avoid marking both a read and a write in the same iteration remains a constant time operation (although slightly more expensive). When using hash tables, the pairwise merges may become slightly more expensive because each processor will be responsible for $O(na/p)$ accesses in each of $\log p$ merging phases, i.e., the total cost of the pairwise merges using hash
tables is $O(na \log p/p)$. If the pairwise merge for hash tables is eliminated by placing the accesses
to the global shadow structure in critical sections (Section 2.7.2), then the complexity becomes
$O(na/p + \log p)$, where the $\log p$ contribution comes from the possible contention in the critical
sections.

The counting in Step 2(a) can be done in parallel by giving each processor $s/p$ values to add
within its private memory, and then summing the $p$ resulting values in global storage, which takes
$O(s/p + \log p)$ time [Lei92]. The comparisons in Step 2(b) and 2(d) of the $A_w$ and $A_r$ ($A_{np}$)
shadow arrays take $O(s/p + \log p)$ time. Again, if $s \gg na$, then the complexity can be reduced to
$O(na/p + \log p)$ by using hash tables.

From the above analysis, we conclude that the doall tests require little communication and
should scale well with all of the parameters, i.e., the number of processors, the size of the shared
variable, and the number of references to the shared variable (encompassing both the number of
iterations, and the number of accesses within an iteration).

4.2.1 Complexity of Saving and Restoring State

The minimum state required to be check-pointed in the case of speculative execution is com-
pletely program dependent. The simplest mechanism for saving and restoring state is to copy the
values of the modified, shared variables affected by the loop in a backup storage. This process can
be done in fully parallel mode and does not produce inter-processor communication. If we assume
that the required storage is $W$ then the checkpointing can be done in $O(W/p)$ time. It is impor-
tant to point out that this model is for a simplistic checkpointing mechanism. In practice many
schemes can be adopted by which the amount of copying can be significantly reduced. Methods for
optimization will be given in a later section.

4.2.2 Complexity of Run–Time Privatization

If the LRPD test determines that privatization is needed, then we either privatize the entire
shared array $A$, or we privatize only the elements of the shared array that are written during the
loop. If the entire array is privatized, then the allocation can be done in constant time. When
privatizing only the elements that are written, the information needed is $tm(A)$, the number of
elements written, and the prefix sums of $A_w$. Since $A_w$ and $tm(A)$ are computed during the test
itself, the only additional information needed is the prefix sums, which can be computed in time
$O(s/p + \log p)$ by recursive doubling [Lei92]. In fact, the prefix sums can be computed at the
same time that $tm(A)$ is accumulated without much extra work. A similar computation can be
performed on the array $A_{nw}$ if only the elements that are written more than once are privatized

43
(as discussed in Section 2.7.2). If the entire array is not privatized, then in the case of subscripted subscripts, private copies of the subscript arrays are also created. Given the original subscript array and \( A_w \), each private subscript array can be created in time \( O(m) \), where \( m \) is the size of the original subscript array.

The complexity of the privatization (of the written elements only) in the hash table implementation will remain the same. The global hash table (after merging) can be constructed as a contiguous segment of memory, i.e., an array, in which the different accesses are stored in a ‘random’ manner (rather than monotonically increasing, as it is the case with shadow arrays). We can compute in parallel the prefix sums of the written elements, rank them and then allocate the necessary private storage. The private subscript arrays will have to be modified so that they point to the newly allocated private, contiguous memory, just like in the case of privatized arrays – with the difference that contiguous elements of the original shared may not be contiguous in their privatized form. This is unlikely to affect performance (locality) because run-time methods are intended to be used for irregular applications.

### 4.2.3 Complexity of the Last Value Assignment

If a private variable is live after the loop terminates, then we also need to perform a last value assignment. In this case, we keep time-stamps (iteration numbers) with the private variables. As mentioned in Section 2.6, if the shadow structures are “marked” with the iteration number, then the marks can double as the time-stamps. Then, after loop termination, the private variable with the latest time stamp is copied to the original (non-privatized) version of the variable. The private variable with the latest time-stamp can be selected from pairwise merges of the private \( A_w \) shadow arrays in time \( O(s + \log p) \). If the test is implemented with hash tables then the merging phase will be more time consuming because the processors will be responsible for \( O(na/p) \) (on the average) comparisons at each of the \( \log p \) stages of the merge, resulting in a time complexity of \( O(na \log p/p) \). If we would implement the test with one global shadow structure that is accessed within critical sections then the complexity will be \( O(na/p + \log p) \) – with the strong possibility of contention.

### 4.3 Performance Prediction

Although it is not strictly necessary for the compiler to perform any cost/performance analysis, the overall usefulness of the tests will be enhanced if their run-time overhead is avoided when the test is likely to fail. Just as important is the evaluation of the speedup in case the loop is found to be fully parallel. If the parallelization of the original loop would give only marginal results, then
by adding the run-time overhead of the LRPD test, it may be possible to incur a slowdown. In this section we will give an analytical approximative bound on the potential speedups and slowdowns which are based on the complexity analysis presented above.

Given a fully parallel loop \( L \), the ideal speedup, \( S_{p_{id}} \), is the ratio between its sequential and its parallel execution times, \( T_{\text{seq}} \) and \( T_{\text{doall}} \), respectively. However, when \( L \) is parallelized using the doall test, the attainable speedup, \( S_p \), must account for the overhead required by the following phases of the test:

- \( T_{\text{mark}} \) - marking phase overhead for the speculative doall test
- \( T_{\text{imark}} \) - marking phase (inspector loop) for the inspector/executor doall test
- \( T_{\text{analysis}} \) - Analysis
- \( T_{\text{save}} \) - Checkpointing (speculative execution)
- \( T_{\text{restore}} \) - State Restoration (speculative execution)

The expected speedup (slowdown) of the doall test will depend on the chosen strategy: inspector/executor (\( S_{p_{/e}} \)) or speculative (\( S_{p_{spec}} \)) execution. In the following we will evaluate both situations and compare the benefits and drawbacks of each method.

\[
S_{p_{id}} = \frac{T_{\text{seq}}}{T_{\text{doall}}}
\]

\[
S_{p_{/e}} = \frac{T_{\text{seq}}}{T_{\text{imark}} + T_{\text{analysis}} + T_{\text{doall}}}
\]

\[
S_{p_{spec}} = \frac{T_{\text{seq}}}{T_{\text{save}} + T_{\text{mark}} + T_{\text{analysis}} + T_{\text{doall}}}
\]

Using static analysis, the compiler can compute an estimate for \( S_p \) by estimating \( T_{\text{seq}} \), \( T_{\text{doall}} \), \( T_{\text{save}} \), \( T_{\text{mark}} \), and \( T_{\text{analysis}} \). The values \( T_{\text{seq}} \) and \( T_{\text{doall}} \) can be estimated using some architectural model, e.g., instruction counting. Our analysis in Section 4.2 predicts that \( T_{\text{mark}} \) and \( T_{\text{analysis}} \) are \( O(na/p + \log p) \), where \( p \) is the number of processors, \( n \) is the number of iterations of the loop, and \( A \) is the maximum number of accesses to the shared array in a single iteration of the loop. In practice, \( T_{\text{analysis}} \) should be fairly well modeled by this expression, i.e., \( T_{\text{analysis}} \approx c(na/p + \log p) \), where \( c \) is some small constant. However, the estimate of \( T_{\text{mark}} \) and \( T_{\text{imark}} \) may not always be as good. The compiler can always evaluate the complexity of the inspector loop (\( T_{\text{imark}} \)). It is in the worst case computationally equivalent to the loop under test, i.e., \( T_{\text{imark}} \) and \( T_{\text{doall}} \) are of the same complexity. We consider the worst case a loop that does almost no computation on its data – it does only a storage redistribution which has to be all shadowed. In the case of speculative execution,
$T_{mark}$ is in the worst case as expensive as the $T_{dual}$ but, in contrast with the inspector approach, does not have the overhead associated with a parallel loop (e.g., setup, index computation, extra synchronization). On the other hand the fact that we access simultaneously the working set of the original loop and that of the shadow structures can make $T_{mark}$ potentially more time consuming than $T_{mark}$. While in the case of the inspector strategy we may replace useful cache lines, in the speculative approach we may thrash the cache because of an increased working set. In order to always make the best decision it would be desirable to incorporate this type of reasoning into a compiler.

Note that in the worst possible case $T_{mark}, T_{analysis}, T_{dual}$ are of the same complexity and the attainable speedup predicted for the inspector case is $\approx \frac{1}{4} S_{pid}$. In the speculative method $T_{mark}, T_{analysis}, T_{save}$ and $T_{dual}$ are all of the same complexity. Summarizing we have:

$$S_{spec} \approx \frac{1}{4} S_{pid} \quad \text{and} \quad S_{i/e} \approx \frac{1}{3} S_{pid}$$

Although 25% of ideal speedup may not appear impressive on an eight processor machine, these tests were designed for massively parallel processors (MPPs), and on such a machine this in an excellent performance when compared to the alternative of sequential execution.

It is also instructive to examine the slowdown, $S_{l_{spec}}$ and $S_{l_{i/e}}$, incurred by a failed test, i.e., when the loop must be executed sequentially. In this case, $T_{seq}$ is increased either by $T_{mark} + T_{analysis}$ or by $T_{save} + T_{mark} + T_{analysis} + T_{restore}$, where $T_{restore}$ is work-equivalent to $T_{save}$. Thus, the cost of performing a failed test is proportional to $\frac{1}{p} T_{seq}$:

$$S_{l_{i/e}} = T_{mark} + T_{analysis} \approx \frac{2}{p} T_{seq}$$

$$S_{l_{spec}} = T_{save} + T_{mark} + T_{analysis} + T_{restore} \approx \frac{4}{p} T_{seq}$$

We should again note that $T_{mark}$ is usually significantly larger than $T_{mark}$ unless, as mentioned before, the increase in the size of the working set of the speculative loop dramatically affects its performance. In the speculative case, the marking instructions are independent of the of the actual computation and could be scheduled on idle functional units and therefore not add at all to the length of the critical path of the loop.

Based on the outcome of the cost/performance analysis, the compiler determines whether the test should be performed, and if it decides to use the test, it must also decide how the test should be applied: using the inspector/executor paradigm (i.e., first test, and then execute) or in a speculative
manner. This decision, which is based on a comparison between \((T_{\text{save}} + T_{\text{mark}})\) and \(T_{\text{mark}}\) is first conditioned by the possibility of extracting an inspector loop – which is generally not the case.

Based on our analysis, the \texttt{doall} test provides potentially high scalable speedups and risks only small slowdowns of the loops are not parallelizable. These slowdowns are also scalable: the more processors we use the smaller the risk. Therefore, we think that unless it is known a priori with a high degree of confidence that the loop is not parallel, the test should always be applied to loops that would otherwise benefit from parallelization.

4.4 Putting it All Together

The run-time techniques described so far are automatable and a good compiler can easily insert them in the original code. In this section, we give a brief outline of how a compiler might proceed when presented with a \texttt{do} loop whose access pattern cannot be statically determined.

1. \textit{At Compile Time}.

   (a) A cost/benefit analysis is performed using both static analysis (based on the analysis presented above) and run-time collected statistics to determine whether the loop should be:

      (i) speculatively executed in parallel using the LRPD test,

      (ii) first tested for full parallelism, and then executed appropriately (using an inspector/executor version of the LRPD Test), or

      (iii) executed sequentially.

   (b) Generate the code needed for the speculative parallel execution. A parallel version of the original loop is augmented with the \texttt{markread}, \texttt{markwrite} and \texttt{markredux} operations for the LRPD test; if necessary to identify reduction variables, the loop is also augmented as described in Section 3.3.1. In addition, code is generated for: the analysis phase of the LRPD Test, the potential sequential re-execution of the loop, and any necessary checkpointing/restoration of program variables.

2. \textit{At Run-Time}.

   (a) Checkpoint if necessary, i.e., save the state of program variables.

   (b) Execute the parallel version of the loop, which includes the marking phase of the test.

   (c) Execute the analysis phase of the test, which gives the pass/fail result of the test.
(d) If the test passed, then compute the final results of all reduction operations (from the processors’ partial results) and copy-out the values of any live private variables. If the test failed, then restore the values of any altered program variables and execute the sequential version of the loop.

(e) Collect statistics for use in future runs, and/or for schedule reuse in this run.

\[
\begin{align*}
C & \quad \text{original loop} \\
\text{dimension} & \quad \text{A(1:m)} \\
\text{do} & \quad i=1, n \\
S1: & \quad A(R(i)) = A(R(i)) + \exp() \\
S2: & \quad \ldots \ldots = A(L(i)) \\
\text{enddo}
\end{align*}
\]

(a)

\[
\begin{align*}
C & \quad \text{Marking Phase} \\
\text{dimension} & \quad \text{A(m), pA(m,procs)} \\
\text{dimension} & \quad \text{A_w(m), pA_w(m,procs)} \\
\text{dimension} & \quad \text{A_r(m), pA_r(m,procs)} \\
\text{dimension} & \quad \text{A_nx(m), pA_nx(m,procs)} \\
& \quad \text{Initialize(pA, pA_w, pA_r, pA_nx)} \\
\text{doall} & \quad i=1,n \\
& \quad \text{private} \quad p \\
& \quad \quad p = \text{get_proc_id()} \\
& \quad \quad pA_w(R(i), p) = i \\
S1: & \quad pA(R(i), p) = pA(R(i), p) + \exp() \\
& \quad \quad \text{if} \quad (pA_w(L(i), p) \neq i) \\
& \quad \quad \quad pA_r(L(i), p) = i \\
& \quad \quad \quad pA_nx(L(i), p) = \text{true} \\
S2: & \quad \ldots \ldots = pA(L(i), p) \\
\text{enddoall}
\end{align*}
\]

(b)

\[
\begin{align*}
C & \quad \text{Analysis Phase} \\
\text{doall} & \quad i=1,n \\
& \quad A_w(1:m) = pA_w(1:m,i) \\
& \quad A_r(1:m) = pA_r(1:m,i) \\
& \quad A_nx(1:m) = pA_nx(1:m,i) \\
\text{enddoall} \\
& \quad \text{result} = \text{test(A_w, A_r, A_nx)} \\
& \quad \text{if} \quad (\text{result} \ .eq. \text{pass}) \quad \text{then} \\
& \quad \quad \text{compute reduction} \\
& \quad \quad \text{doall} \quad i=1, m \\
& \quad \quad \quad \text{if} \quad (A_nx(i) .eq. \text{false.}) \\
& \quad \quad \quad \quad A(i) = \text{sum(pA(i, 1:procs))} \\
& \quad \quad \text{enddoall} \\
& \quad \quad \text{else} \\
& \quad \quad \quad \text{execute the loop sequentially} \\
& \quad \text{endif}
\end{align*}
\]

(c)

**Figure 4.1** Generated code for the LRPD test. The simplified code generated for the do loop in (a) is shown in (b) and (c). Privatization is not tested because of a read before a write reference. The mark-x operations are as described in Fig. 2.4.

An example using iteration numbers as “marks” in private shadow arrays is shown in Fig. 4.1. If the speculative execution of the loop passes the analysis phase, then the scalar reduction results are computed by performing a reduction across the processors using the processors’ partial results. Otherwise, if the test fails, the loop is re-executed sequentially.
4.5 Optimizations

So far we have described a very general algorithm that can be applied to any loop. There are in fact techniques through which static and run-time analysis can be used to optimize the LRPD test and reduce, sometimes drastically, its associate overhead. In the following sections we will give several methods and hint to some ideas that could increase the usability of the described run-time methods.

4.5.1 Marking Phase Overhead Reduction through Reference Aggregation

The simple, and rather naive way to insert the marking code into a loop is to simply add a \texttt{markwrite}, \texttt{markread}, \texttt{markredux} macro for every occurrence of a write and read access to the shadowed array. But a relatively simple analysis the compiler can remove duplication. In certain nested loops slicing analysis [Wei84] could generate whole sections of arrays that need to be marked. This could either only improve locality or result in a different implementation of the \texttt{doall} test altogether. Instead of using shadow arrays or shadow hash-tables we could use shadow intervals and interval arithmetic for our analysis. If the intervals are long enough then this implementation could become much faster although each operation is more expensive. Sometimes a hybrid approach in which both intervals and arrays are used could be the best solution for a particular code. We believe that those loops from which an inspector can be extracted are the best candidates for reference aggregation.

4.5.2 Schedule Reuse

Thus far, we have assumed that a \texttt{doall} test is run each time a loop is executed in order to determine if the loop is parallel. However, if the loop is executed again, with the same data access pattern, the first test can be reused amortizing the cost of the test over all invocations. This is a simple illustration of the \textit{schedule reuse} technique, in which a correct execution schedule is determined once, and subsequently reused if all of the defining conditions remain invariant (see, e.g., Saltz \textit{et al.} [SMC91]). If it can be determined at compile time that the data access pattern is invariant across different executions of the same loop, then no additional computation is required. Otherwise, some additional computation must be included to check this condition, e.g., for sub scripted subscripts, the old and the new subscript arrays can be compared. We remark that most programs are of a repetitive nature, and thus there exists the potential for schedule reuse. A simple example in which schedule reuse could be considered is for multiply nested loops. If possible,
it is generally best to parallelize the outer loop in the nesting. However, if this is not possible, then it may be the case that schedule reuse could be attempted when parallelizing the inner loops.

Unfortunately this optimization technique can be used only when an inspector can be extracted, i.e., when the access pattern is determined before loop execution and therefore not dependent on the values computed in the loop itself. This implies that for the cases when speculative execution is the only possibility, i.e., for dynamic reference patterns, schedule reuse is not an option.

4.5.3 Inspector Decoupling

There may cases in which the defining parameters of an inspector loop become available (are defined) well before the actual loop under test. In this case the doall test code can be moved up and executed early, during a portion of the program that does not enough (or not at all) parallelism. This strategy assumes that it is not advantageous to release processors when they are idle for short periods of time. In this case the time consumed by the run-time test does not increase the overall length of the critical path of the program. It follows that if indeed our primary goal is to minimize the execution time of an application on a computer system rather than overall efficiency in a multiuser environment, then, in some cases, the overhead of using run-time tests can be completely hidden.

4.5.4 Reducing The Risk of Potential Slowdown

As it has been shown above run-time parallelization always produce the risk of unwanted slowdowns if the parallelism that we are looking for does not materialize. Though these overheads are scalable with the number of processors and are not very large, we will present strategies to minimize their impact.

4.5.4.1 Early Dependence Checking

The doall test, as described so far assumes that all references have to be accounted for before a decision can be made. That is true in the case we want to prove the loop parallel. However, in order to reduce the cost of failed test we can detect, almost without any additional cost, if an element causes a cross-iteration dependence among those iterations executing on the same processor. If this happens we can decide to fail the test immediately without actually finishing the rest of the un-executed code and without any cross-processor analysis. Of course, if we decide to apply the processor-wise version of the test then cross-iteration dependences are not a reason to disqualify a
loop. It should be noted here that if a loop has dependences it is highly likely that these dependences will appear well before the last iteration is executed.

4.5.4.2 The 1 Processor and \((p - 1)\) Processor Strategy

Another way to insure that we will never increase the critical path of the program and therefore not perform worse than the conservative sequential execution is to always fork two processes before applying run-time techniques. One processor will execute the sequential version of the loop under test and the rest of \((p - 1)\) processors will proceed on the more aggressive parallel path. Then the cluster of processor(s) finishing first will commit its results and interrupt the other processors. Of course this scheme is not completely without cost. Forking two processes with their own working space at the beginning of every run-time parallelized loop can have also a significant overhead – mostly due to data copying from and into committed storage. If the working set of the loop under test is not very large (and not cause a high cache miss-ratio) then this overhead is quite acceptable.

4.5.4.3 Use of Statistics as Feedback

Probably the most effective way to reduce the cost of failed parallelization attempts is to `guess` correctly when a loop might be parallel. Although, in principle, it is not possible to find an algorithm that produces 100\% accurate predictions, we can propose many heuristics. A method that may prove to be very powerful, is using statistics collected during past instantiations of the loop. Simply put, if the loop has been found to be parallel in the past, then there is a good chance that it will be parallel in the future. Such techniques have been extensively used in predicting paging behavior, cache misses and branch outcome. Unfortunately there is no experimental evidence that statistics about loop data dependence structure is significant. We believe that for programs with a dynamic data distribution structure we will be able to predict time-varying trends. The redistribution of data in an N-body problem has a much larger time-constant than the frequency of instantiations of the loop under test – otherwise the time-step of the simulation would be mismatched to the physical problem. It is reasonable to assume that during the execution of a program there is a strong correlation between the parallelism behavior of time-adjacent instantiations of a loop.

There are several levels at which a properly modeled and collected statistical information can be used. If the program has a long execution time and there are many invocations of the tested loop then the feedback (prediction) can be applied within the same run. Additionally, this information can be used in future invocations of the program – whether it is with the same or different input data. The larger the difference between input files (or initial conditions) the less effective our prediction is likely to be. Finally, the collected information could be used in future compilations
of a program if it proves to be stable with respect to input data. We believe that static compiler analysis can be useful in identifying which values affect the data dependence structure of the loops and based on this information check early on if the feedback data is likely to help predict correctly or may do more harm than good. It should be noted that 'guessing' wrong does not only generate more run-time overhead associated with the LRPD test, but also affects an overly conservative strategy: if we think that the loop is not parallel, we will not waste any time checking it, but we will miss the chance of speeding it up. So, generally speaking, without having any scientific proof, it would reasonable to advise biasing the decision-making process towards the frequent application of the run-time tests. As we have shown before, losses are small and gains can be very large, therefore more than compensating for any lost time.

It should be remarked here that the 'schedule reuse' technique is a trivial case of feedback use – it is not a statistical method since it is deterministic in character.

Another, more 'esoteric' heuristic in speculating successfully is to use a 'fuzzy' rule based system at compile time. Experience has shown that programmers give names to variables based on their semantics. If an array is named 'help' or 'temp' it is quite likely that it is privatizable. If a loop uses an index named 'time', then it is likely that this loop is sequential. It would be a useful study to try to correlate (statistically) names of variables, programming styles to their actual attributes. After all, a parallelizing compilation is, to a certain degree, a reverse engineering job – we want to rediscover the original intent, algorithm, of the programmer.
Chapter 5

Experimental Results for the LRPD Test

5.1 Experimental Setup

We will present experimental results obtained on two modestly parallel machines with 8 (Alliant FX/80 [All86]) and 14 processors (Alliant FX/2800 [All91]) using a Fortran implementation of our run-time library. This library consists of subroutines that can be called to perform the different phases of the test (backup, analysis, restoration). There are specialized routines for the different cases that have to be tested (privatization, reduction, output dependences, etc.).

The codes have been mantually instrumented with calls to the run-time library. However, we remark that our results scale with the number of processors and the data size and thus they could be extrapolated for massively parallel processors (MPPs), the actual target of our run-time methods.

We considered seven do loops from the PERFECT Benchmarks [BCK+89] that could not be parallelized by any compiler available to us. Our results are summarized in Table 5.1. For each loop, we note the type of test applied: *doall* indicates cross-iteration dependences were checked (Lazy Doall (LD) test), *privat* indicates privatization was checked (LPD test), *reduct* indicates reduction parallelization was checked (LRD test). For each method applied to a loop, we give the speedup that was obtained, and the potential slowdown that *would have been incurred* if, after applying the method, the loop had to be re-executed sequentially. If the inspector/executor version of the LRPD test was applied, the computation performed by the inspector is shown in the table: the notation *privatization* indicates the inspector verified that the shared array was privatizable and then dynamically privatized the array for the parallel execution, *branch predicate* and *subscript array* mean that the inspector computed these values, and *replicates loop* means that the inspector was work-equivalent to the original loop.

For all of the loops studied we will show a good fit of our experimental data to the speedups predicted by the modeling described in Section 4.3. Our estimates were made using a simple instruction counting model: for loops (vectors), we used the product of the number of instructions and the number of iterations (elements). The prediction could be significantly improved by using a more accurate model but for the purpose of this work we felt it was not necessary. In addition to the summary of results given in Table 5.1, we show in Figures 5.1 through 5.7 the speedup and the potential slowdown measured for each loop as a function of the number of processors used. For reference, these graphs show the ideal speedup, which was calculated using an optimally
parallelized (by hand) version of the loop. The potential slowdown reported is the percentage of the execution time that would be paid as a penalty if the test had failed, and the loop was then executed sequentially. In cases where extraction of a reduced inspector loop was impractical because of complex control flow and/or inter-procedural problems, we only applied the speculative methods.

Whenever necessary in the speculative executions, we performed a simple preventive backup of the variables potentially written in the loop. In some cases, the cost of saving/restoring might be significantly reduced by using another strategy. The shadow arrays have been implemented by allocating private arrays of sufficient size on all processors. In order for our methods to scale better with the number of processors, the shadow arrays must be distributed over the processor space, rather than replicated on each processor (Section 4.2). For this purpose, we tried using hash tables. Since we had at most 14 processors, the extra cost of the hash accesses dominated the benefit of reducing the size of the shadow arrays. This was particularly true for the loops from the OCEAN and TRFD Benchmarks. However, on a larger machine we would expect the use of hash tables to pay off. Due to this problem, the results reported do not reflect the use of hash tables. The cross-processor merge operations have not been implemented with a 'pair-wise merge' algorithm because our experiments have been performed on a small machine. This fact has resulted in an apparent lack of scalability. We are quite confident that this problem will be solved by the proper implementation.

5.2 Experimental Results

The graphs show that in most cases the speedups scale with the number of processors and are a very significant percentage of the ideal speedup. When they do not scale, as mentioned above, we believe that the use of hash tables (for MPPs) will preserve the scalability of our methods. We note that with the exception of the TRFD loop (Fig. 5.3), the speculative strategy gives superior speedups versus the inspector/executor method. For both methods the potential slowdown is small, and decreases as the number of processors increases. As expected, the potential slowdown is smaller for the inspector/executor method.

We now make a few remarks about individual loops for which Table 5.1 does not give complete information.

TRACK–NLFILT–Loop 300. The loop from TRACK is parallel for only 90% of its invocations: Out of its 59 instantiations, five times it is found not to be a doall. In the cases when the test failed, we restored state, and re-executed the loop sequentially. The speedup reported includes both
<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Experimental Results</th>
<th>Tested</th>
<th>Description of Loop</th>
<th>Inspector (computation)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MDG</td>
<td>14 processors</td>
<td>dail</td>
<td>accesses to a privatizable vector guarded by loop computed predicates (92% Tseq)</td>
<td>privatization</td>
</tr>
<tr>
<td>INTERF</td>
<td>spec. 11.55 1.09</td>
<td>dail</td>
<td>accesses privatizable array indexed by a subscript array computed inside loop (32% Tseq)</td>
<td>data accesses</td>
</tr>
<tr>
<td>loop 1000</td>
<td>1/e 8.77 1.02</td>
<td>privat</td>
<td>branch predicate</td>
<td></td>
</tr>
<tr>
<td>BDNA</td>
<td>14 processors</td>
<td>dail</td>
<td>small triangular loop accesses a vector indexed by a subscript array computed outside loop (5% Tseq)</td>
<td>data accesses</td>
</tr>
<tr>
<td>ACTFOR</td>
<td>spec. 10.65 1.09</td>
<td>dail</td>
<td>replicates loop</td>
<td></td>
</tr>
<tr>
<td>loop 240</td>
<td>1/e 7.72 1.04</td>
<td>privat</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRFD</td>
<td>spec. 8.53 2.17</td>
<td>dail</td>
<td>accesses array indexed by subscript array computed outside loop, access pattern guarded by loop computed predicates (36% Tseq)</td>
<td>not applicable</td>
</tr>
<tr>
<td>INTGRL</td>
<td>sched 1.05 1.74</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop 140</td>
<td>1/e 2.10 1.74</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TRACK</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>NFILRT</td>
<td>spec. 4.21 1.01</td>
<td>dail</td>
<td>accesses array indexed by subscript array computed outside loop, access pattern guarded by loop computed predicates (36% Tseq)</td>
<td>not applicable</td>
</tr>
<tr>
<td>loop 300</td>
<td>8 processors</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ADM</td>
<td>14 processors</td>
<td>dail</td>
<td>accesses privatizable array thru aliases, array re-dimensioned, access pattern control flow dependent (44% Tseq)</td>
<td>not applicable</td>
</tr>
<tr>
<td>RUN</td>
<td>spec. 9.01 1.02</td>
<td>privat</td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop 20</td>
<td>14 processors</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>OCEAN</td>
<td>8 processors</td>
<td>dail</td>
<td>kernel-like loop accesses a vector with run-time determined strides 26K invocations account for 43% Tseq</td>
<td>data accesses</td>
</tr>
<tr>
<td>FTRVMT</td>
<td>spec. 2.28 1.43</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop 109</td>
<td>1/e 2.14 1.50</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SPICE</td>
<td>LOAD</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>loop 40</td>
<td>1/e 2.75 1.09</td>
<td>reduct</td>
<td>traverses linked list terminated by a NULL pointer, all referenced arrays equivalenced to a global work array</td>
<td>data accesses</td>
</tr>
</tbody>
</table>

Table 5.1 Summary of experimental results. Notation: Sp – speedup, SI – potential slowdown, i/e – inspector/executor, spec. – speculative, Tseq – sequential execution time of the entire program.

The parallel and sequential instantiations (Fig. 5.4). This case in which some of the invocations of the loop are parallel while others are not is a relevant example of the use and power of the LRPD test and hints to other possible applications In Chapter 9 we elaborate more on this issue.

**BDNA–ACTFOR–Loop 240.** This loop selects certain elements from a large array, and processes the selected elements later in the loop. The speedup, $S_{pi/e}$, obtained for this loop using the PD test is almost 2/3 of the ideal speedup (see Fig. 5.2). This speedup cannot be accurately predicted at compile time because the number of selected elements is not known until run-time, and the inspector executes the selection phase (computing the subscripts), but not the subsequent processing phase. The speculative speedup ($S_{p_{spec}}$) is much better because the inspector loop replicates a significant portion of the computation of the loop.

**MDG–INTERF–Loop 1000.** This loop calculates inter-molecular interaction forces. In order to avoid false dependences, our inspector computes the branch predicates and has an estimated complexity $T_{mark} \approx 2T_{dail}$ and a minimal $T_{mark}$. Since $T_{analysis}$ is small (the shadow vector has only 14 elements), we expect the LRPD test to obtain $S_{p_{i/e}} \approx 0.7S_{p_{id}}$ and a much better $S_{p_{spec}}$. The results shown in Fig. 5.1 display a pretty good fit to this estimation.

**SPICE–LOAD–Loop 40.** It is representative of the type of loops contained in the LOAD subroutine, which accounts for 70% of the sequential execution time. Since all the arrays are equivalenced
to a global work array, all accesses in the loop were shadowed in the LPRD test, i.e., each array element was proven to be either a reduction variable, read-only, or independent (i.e., accessed in only one iteration). For this loop we used only an inspector/executor version of the LPRD test because we have not yet implemented complex memory management for shadow structures in the presence of highly irregular and sparse access patterns. The ideal speedup of loop 40 is not very large since the loop is small, imbalanced between iterations, and traverses a linked list. The linked list traversal was parallelized using techniques we developed for automatically parallelizing while loops and described in Chapter 8. Thus, although the obtained speedup is modest, it represents a significant fraction of the ideal speedup (see Fig. 5.7). Therefore, since loop 40 is one of the smallest loops in the LOAD subroutine, we expect to obtain better speedups on the larger loops (since they have larger ideal speedups).

**OCEAN–FTRVMT–Loop 109.** This loop is utilized in the computation of a 2-dimensional FFT. It is invoked 26,000 times, and accounts for 40% of the sequential execution time of the program. For this loop we estimate $T_{mark} \approx T_{analysis} \approx T_{save} \approx T_{doall}$, $T_{mark} \ll T_{mark}$, $T_{save}$ small and predict $S_{pi/c} \approx \frac{T_{seq}}{T_{doall}} = \frac{1}{7} S_{pid}$ and $S_{ppec} < \frac{T_{seq}}{T_{doall}} = \frac{1}{7} S_{pid}$. The speedups obtained are relatively small because the loop has all the characteristics of a kernel (a loop containing only shadowed data movement). Applying the 'schedule-reuse' technique was simple and very beneficial. The five scalars that determine the access pattern are checked against a record – if they are found then the previous test result is used. The test had to applied only 20 times over all the 26,000 invocations of the loop resulting in an essentially ideal speedup.

**TRFD–INTGRL–Loop 140.** For this small triangular loop, $S_{pid} \approx 4.5$ on the Alliant FX/80 because of load imbalance. Statically, we would predict $T_{doall} \approx T_{mark} \approx T_{analysis}$, and $S_p \approx \frac{7}{5} S_{pid}$. However, since our implementation does not use hash tables, the access pattern of our shadow arrays is rectangular (versus the triangular pattern of the loop) and $T_{mark} \approx T_{mark} \approx T_{analysis} \approx 2T_{doall}$, so that $S_{pi/c} \approx \frac{1}{5} S_{pid} < 1$. In the case of speculative execution the $T_{save}$ is very large due to the large array that has to be check-pointed and can be approximated by $T_{doall}$. This is why the loop from TRFD is the only case in which speculative execution proved to be inferior to the inspector/executor method. We were able to reuse the schedule and improve our results significantly. Loop 140 is executed seven times (within an outer loop) and uses a larger subscript array on each subsequent execution. In order to apply the schedule reuse technique (Section 4.5.2) we have to prove that the current subscript array is a subset of the one from the previous invocation. We have extracted an inspector for the largest subscript array and verified at run-time this subset relation.

56
5.3 Conclusion

The experimental results obtained show that is usually more advantageous to use the speculative method unless the checkpointing is overly expensive. We believe that this effect (of time-consuming saving/restoration phases) may become more important as we move to larger data sets and the number of processors remains constant. For this reason checkpointing mechanisms have to be improved – through both hardware and compiler techniques.

We should note again the importance of the TRACK_NLFILT_300 loop experiment. It exemplifies how a dynamic access pattern can influence the character of a loop and how an aggressive strategy can pay off. Even if the loop would have been sequential in more than 10% of the cases we could have still obtained a speedup. In case of larger parallel machines this result would be tilted even more in favor of speculative parallelization.
Figure 5.1 Speedup and potential slowdown in MDG

Figure 5.2 Speedup and potential slowdown in BDNA

Figure 5.3 Speedup and potential slowdown in TRFD

58
Figure 5.4 Speedup and potential slowdown in TRACK

Figure 5.5 Speedup and potential slowdown in ADM

Figure 5.6 Speedup and potential slowdown in OCEAN
Figure 5.7 Speedup and potential slowdown in SPICE
Chapter 6

Run–Time Methods for Parallelizing Partially Parallel Loops

6.1 Introduction

The majority of the previous work [BS90, CYT94, KS88, LZ93, MP87, Pol88, SM91, SMC89, SMC91, WSHB91, ZY87] in the domain of run-time parallelization has concentrated on developing run-time methods for constructing execution schedules for partially parallel loops, i.e., loops whose parallelization requires synchronization to ensure that the iterations are executed in the correct order. Given the original, or source loop, most of these techniques generate inspector code that analyzes, at run-time, the cross-iteration dependences in the loop, and scheduler/executor code that schedules and executes the loop iterations using the dependence information extracted by the inspector [SMC91].

While in the previous chapters we have presented an algorithm to parallelize loops that are, or can be made fully parallel (doall), in this chapter we give a new inspector/scheduler/executor method for finding an optimal parallel execution schedule for a partially parallel loop. The inspector is fully parallel, uses no synchronization, and can be applied to any loop (from which an inspector can be extracted). In addition, this inspector can implement at run-time the two most effective transformations for increasing the amount of parallelism in a loop: array privatization and reduction parallelization (element-wise). The ability to identify privatizable and reduction variables is very powerful since it eliminates the data dependences involving these variables. Thus, in addition to increasing the available parallelism in the loop, the removal of these dependences also reduces the work required of the scheduler, i.e., it need not consider the affected variables. We describe a scheme for constructing an optimal parallel execution schedule for the iterations of the loop. The schedule produced is a partition of the set of iterations into subsets called wavefronts, so that the iterations in each wavefront can be executed in parallel, i.e., there are no data dependences between iterations in a wavefront. Although the wavefronts themselves are constructed one after another, the computation of each wavefront is fully parallel and requires no synchronization. The scheduling can be dynamically overlapped with the parallel execution of the loop iterations in order to utilize the machine more uniformly. Therefore, our new method has advantages over all the previous techniques cited above since none of them has all of these desirable properties (a comparison to previous work is contained in Section 7.2).
First, in Section 6.2, we describe a new inspector scheme that in many cases should prove superior to previously proposed schemes. Next, in Section 6.3, we present a scheduler that can use the dependence information found by the inspector to construct an optimal parallel execution schedule for the loop iterations; in addition, we mention how the scheduler might be interleaved with the executor to more efficiently utilize the machine. After describing the basic components of our methods, in Section 6.4 we discuss some strategies for applying them most effectively. Experimental results will be shown in Section 6.5. A comparison to other run-time parallelization schemes is done in Section 7.2.

In the following we will assume that the reader is already familiar with the background information about data dependence analysis and loop transformations presented in the previous chapters.

6.2 The Inspector

In this section we describe a new inspector scheme that processes the memory references in a loop and constructs a data structure which the executor can use to efficiently assign iterations to wavefronts. In addition, our inspector can implement at run-time two important transformations: (element-wise) array privatization and reduction parallelization (see Section 2.2). The ability to identify privatizable and reduction variables eliminates the data dependences involving these variables and increases the parallelism of the loop. Additionally it reduces the work required of the scheduler since it need not consider dependences involving such variables when it constructs the parallel execution schedule for the loop iterations.

The basic strategy of our method is for the inspector to preprocess the memory references and determine the data dependences for each memory location accessed. Later, the scheduler will use this memory-location dependence information to determine the data dependences between the iterations. We describe the method as applied to a shared array A that is accessed through subscript arrays that could not be analyzed at compile-time (see Figure 6.1(a)). For simplicity, we first consider only the problem of identifying the cross-iteration dependences for each array element (memory location). After describing this inspector, we then discuss how the dependence information it discovers can be used to identify the array elements that are read-only, privatizable, or reduction variables. The inspector has two main tasks.

1. For each array element A[x], the inspector collects all the references to it into an array (or list) \( R_x \) and stores them in order of iteration number. For each reference it stores the associated iteration number and access type (i.e., read or write) (see Figure 6.1(b)).
\[
\text{do } i = 1, 8 \\
\quad A(W(i)) = ... \\
\quad A(\text{work}(i)) \\
\text{enddo } (a)
\]

\[
W(1:8) = [1 \ 3 \ 2 \ 4 \ 3 \ 5 \ 6 \ 3] \\
R(1:8) = [3 \ 7 \ 3 \ 3 \ 8 \ 3 \ 3 \ 3]
\]

\[
\begin{array}{ccccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

\[
\begin{array}{ccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\end{array}
\]

2. For each array element \(A[x]\), the inspector determines the data dependences between all its references and stores them in a data structure \(H_x\) for later use by the scheduler.

In Section 6.2.1, we discuss how the references to each array element can be collected and stored in the array (or list) \(R_x\). Thus, assuming that \(R_x\) is available, we now describe how the inspector determines the dependences among the references to \(A[x]\) and computes the data structure \(H_x\).

The relations between the references to \(A[x]\) can be organized (conceptually) into an array element dependence graph \(D_x\). If adjacent references in \(R_x\) have different access types, then a flow or anti dependence exists, and if they are both writes, then an output dependence is signaled. These dependences are reflected by parent-child relationships in \(D_x\). If adjacent references are both reads, then there is no dependence between the elements, but they may have a common parent (child) in \(D_x\): the last write preceding (first write following) them in \(R_x\). For example, the dependence graph \(D_3\) for \(A[3]\) is shown in Figure 6.1(c).

Our goal is to encode the predecessor-successor information of the (conceptual) dependence graph \(D_x\) in a hierarchy vector \(H_x\) so that the scheduler can easily look-up the dependence information for the references to \(A[x]\). First, we add a level field to the records in \(R_x\), and store in it the reference’s level in the dependence graph \(D_x\) (see Figure 6.1(b)). Then, for each level, we store
in \( H_x \) the index (pointer to location) in \( R_x \) of the first reference at that level. Specifically, \( H_x \) is an array and \( H_x[i] \) contains the index in \( R_x \) of the first reference at level \( i \), i.e., \( H_x \) will serve as a look-up table for the first reference in \( R_x \) at any level (see Figure 6.1(d)). Note that this implies that \( H_x \) records the position in \( R_x \) of every write access and of the first read access in any run of reads.

We now give an example of how the hierarchy vector serves as a look-up table for the predecessors and successors of all the accesses. Consider the read access to \( A[3] \) in the 6-th iteration, which appears as the sixth entry in \( R_3 \). Its level is 5, and thus it finds its successor by looking at the \( 5 + 1 = 6 \)th element of the hierarchy vector \( H_3 \), which contains the value 8 indicating that its successor is the 8th element in \( R_3 \). Similarly, its predecessor is found by looking in the \( 5 - 1 = 4 \)th element of \( H_3 \), which indicates that its predecessor is the 5th element of \( R_3 \).

### 6.2.1 Implementing the Inspector

We now consider how to collect the accesses to each array element \( A[x] \) into the arrays \( R_x \). Regardless of the technique used to construct these arrays, to ensure the scalability of our methods we must process (mark) the references to the shared array \( A \) in a detail (see Figure 6.2(a) and (b)). The computation performed in the marking operations will depend upon the technique used to construct the arrays \( R_x \). In any case, note that since we are interested in cross-iteration data dependences we need only record at most one read and write access in \( R_x \) for any particular iteration, i.e., subsequent reads or writes to \( A[x] \) in the same iteration can be ignored.

#### 6.2.1.1 Lexicographic Sort

Perhaps the simplest method of constructing the element arrays \( R_x \) is to first place a record for each memory reference into an array \( R_A \), and then sort these records lexicographically by array element (first key) and iteration number (second key). After this sort, each array \( R_x \) will occupy a contiguous portion (a sub-array) in the sorted array \( R_A \). In this case the marking operations will simply record the information about the access into \( R_A \). After the lexicographic sort, the level of each reference in \( D_x \) can be computed by a prefix sum computation.

#### 6.2.1.2 Bucket Sort

Fortunately, since the range of the values to be sorted is known in advance (it is given by the dimension of the shared array \( A \)), a linear time bucket or bin sort can be used in place of the more general \( O(n \log n) \) lexicographic sort. Moreover, if the inspector’s marking phase is chunked
Figure 6.2 Result of the marking phase. An example of how the private element arrays $pR$ and hierarchy vectors $pH$ (c) when two processors are used in the inspector $doall$ loop (b) for the source $do$ loop (a).

(i.e., statically scheduled), then further optimization is possible. In this case, processor $i$ will be assigned iterations $i[n/p]$ through $(i + 1)[n/p] - 1$, where $p$ is the total number of processors, $n$ is the number of iterations in the loop, and $0 \leq i < p$. The basic idea is as follows. First, in a private marking phase, each processor marks the references in its assigned iterations, and constructs element arrays $R_x$ and hierarchy vectors $H_x$ as described above, but only for the references in its assigned iterations. Then, in a cross-processor analysis phase, the hierarchy vectors for the whole iteration space of the loop are formed using the processors’ hierarchy (sub)vectors.

The private marking phase proceeds as follows. Let $A[1 : s]$ be the shared array under scrutiny, and suppose each processor has a separate array $pR[1 : s, 1 : 2n/p]$ in which to store the records of the references in its set of iterations. Each record contains the iteration, type of reference, and
level as described above. (The second dimension of $1 : 2n/p$ follows since, as noted above, at most
one read and write to any element need to be marked in each iteration, and each processor has $n/p$
iterations.) Assuming a processor marks its iterations in order of increasing iteration number, it
can immediately place the records for the references into its array $pR$ in sorted order of iteration
number. In addition to the array $pR$, each processor has a separate array $pH[1 : s, 1 : 2n/p]$ used
to store the hierarchy vectors for the references in its assigned set of iterations. Again, assuming
that iterations are processed in increasing order of iteration number, the hierarchy vectors can be
filled in at the same time that the references are recorded in $pR$ (see Figure 6.2(c)).

In the analysis phase we need to find for each array element $A[x]$ the predecessor, if any, of the
first reference recorded by each processor, i.e., we need to fill in the value in processor $i$’s hierarchy
vector for the reference that immediately precedes (in the dependence graph $D_x$) the first reference
to $A[x]$ that was assigned to processor $i$. Similarly, we must find the immediate successor of the last
reference to $A[x]$ that was assigned to processor $i$. Processor $i$ can find the predecessors (successors)
needed for its hierarchy vectors by scanning the arrays of the processors less than (larger than) $i$.
For example, the “?” at the end of $pH[3]$ for processor 1 in Figure 6.2 would be filled in with a
pointer to the first element in the array $pR[3]$ of processor 2. Hence, the initial and final entries
in the hierarchy vectors also need to store the processor number that contains the predecessor and
successor. These scans can be made more efficient by maintaining some auxiliary information, e.g.,
for each array element, each processor computes the total number of accesses it recorded, and the
indices in $pR$ of the first and last write to that element. In any case, we note that filling in the
processors’ hierarchy vectors requires a minimal amount of interprocessor communication, i.e., it
requires only a “connecting” and not a full “merging” of the different hierarchy vectors.

There are several ways in which the above sketched analysis phase can be optimized. For
example, in order to determine which array elements need predecessors and successors (i.e., the
elements with non-empty arrays $R_x$), the processor needs to check each row of its array $pR$ (row $i$
of $pR$ corresponds to the array $R_i$). This could be a costly operation if the dimension of the original
array is large and the processor’s assigned iterations have a sparse access pattern. However, the
need to check each row in $pR$ can be avoided by maintaining a list of the non-empty rows. This list
can be constructed during the marking phase, and then traversed in the analysis phase – thereby
avoiding the need to check every row. Another source of inefficiency for machines with many
processors is the search for a particular predecessor (or successor) since each processor might need
to look for a predecessor in all the preceding (succeeding) processors’ iterations. The cost of these
searches can be reduced from $p$ to $O(\log p)$ using a standard parallel divide-and-conquer “pair-wise”
merging approach [Lei92], where $p$ is the total number of processors.
do i = 1, n
S1: A(K(i)) = .......
S2: ............ = A(L(i))
S3: A(R(i)) = A(R(i)) + exp()
endo

private integer j
do j=start(p,niter),end(p,niter)
markwrite(K(i))
markredux(K(i))
markread(L(i))
markredux(L(i))
markwrite(R(i))
endo
doall
enddoall

Figure 6.3 Example of inspector loop with privatization and reduction detection. The inspector of the do loop in (a) is shown in (b). The markwrite (markread) operation adds a record to the processor’s array pR (if its not a duplicate), and updates the hierarchy vector pH appropriately. The marknoredx operation invalidates the indicated array element as a reduction variable since it is accessed outside the reduction statement S3.

6.2.2 Privatization and Reduction Recognition at Run–Time

The basic inspector described above can easily be augmented to find the array elements that are independent (i.e., accessed in only one iteration), read-only, privatizable, or reduction variables. We first consider the problem of identifying independent, read-only, and privatizable array elements. During the marking phase, a processor maintains the status of each element referenced in its assigned iterations with respect to only these iterations. In particular, if it finds than an element is written in any of its assigned iterations, then it is not read-only. If an element is accessed in more than one of its assigned iterations, then it is not independent. If an element was read before it was written in any of its assigned iterations, then it is not privatizable. Next, the final status of each element is determined in the cross-processor analysis phase as follows. An element is independent if and only if it was classified as independent by exactly one processor, and was not referenced on any other processor. An element is read-only if and only if it was either determined to be read-only by every processor that referenced it. Similarly, an element is privatizable if and only if it was either privatizable on every processor that accessed it. Thus, the elements can be categorized by a similar process to the one used to find the predecessors and successors when filling in the processors’ hierarchy vectors. Finally, if we maintain a linked list of the non-empty rows of pR as mentioned above, then the rows corresponding to elements that were found to be independent, read-only, or privatizable are removed from the list, i.e., accesses to these elements need not be considered when constructing the parallel execution schedule for the loop iterations.

We now consider the problem of verifying that a statement is a reduction using run-time data dependence analysis. Recall, as mentioned in Section 2.2, that potential reduction statements are
generally identified by syntactically matching the statement with the generic reduction template
\[ x = x \otimes \text{exp}, \]
where \( x \) is the reduction variable, and \( \otimes \) is an associative and commutative operator.
The statement is validated as a reduction if it can be shown through dependence analysis that \( x \) is
not referenced in \( \text{exp} \) or anywhere in the loop body outside the reduction statement. Sometimes
the necessary dependence analysis cannot be performed at compile-time. This situation could
arise if the reduction variable is an array element accessed through subscripted subscripts, and
the subscript expressions are not statically analyzable. For example, although statement \( s3 \) in the
loop in Fig. 6.2.2(a) matches a reduction statement, it is still necessary to prove that the elements
of array \( A \) referenced in \( s1 \) and \( s2 \) do not overlap with those accessed in statement \( s3 \), i.e., that:
\[ K(i) \neq R(j) \text{ and } L(i) \neq R(j), \quad \text{for all } 1 \leq i, j \leq n. \]
It turns out that this condition can be tested in the same way that read-only and privatizable array elements are identified. In particular,
during the marking phase, whenever an element is accessed outside the reduction statement the
processor invalidates that element as a reduction variable. Again, the final status of each element
is determined in the cross-processor analysis phase, i.e., an element is a reduction variable if and
only if it was not invalidated as such by any processor.

This strategy can also be used when the \( \text{exp} \) part of the RHS of the reduction statement contains
references to the array \( A \) that are different from the pattern matched LHS and cannot be statically
analyzed, i.e., the elements referenced in \( \text{exp} \) are invalidated during the marking phase. A more
complicated situation is when the loop contains several reduction statements that refer to the same
array \( A \). In this case the type of the reduction operation performed on each element must be the
same throughout the loop execution, e.g., a variable cannot participate in both a multiplicative
and an additive reduction since the resulting operation is not commutative and associative and is
therefore not parallelizable. The solution to this problem is to also maintain the reduction type
with each potential reduction variable. Whenever a reference in a reduction statement is marked,
the current reduction type (e.g., summation, multiplication) is checked with with previous one. If
they are not the same, the corresponding element is invalidated as a reduction variable.

6.2.3 Complexity of the Inspector

The worst case complexity of the inspector is \( O(a \log p) \), where \( a \) is the maximum number of
references assigned to each processor and \( p \) is the total number of processors. In particular, using
the bucket sort implementation, each processor spends constant time on each of its \( O(a) \) accesses
in the marking phase, and the analysis phase takes time \( O(a \log p) \) using a parallel divide-and-
conquer pair-wise merging strategy [Lei92]. We remark that since the cost of the analysis phase is
proportional to the number of distinct elements accessed (i.e., the number of non-empty rows in
the \( pR \) array) the complexity of this phase could be significantly less than \( O(a \log p) \) if there are many repeated references in the loop. Also, if \( a \log p > s \), then the merge among the processes can be improved to \( O(s + \log p) \) time by chunking the \( pR \) arrays.

### 6.3 The Scheduler

We now consider the problem of finding an execution schedule for the iterations of the loop. We assume that the inspector described in Section 6.2 has been used on the loop. The scheduler derives the more restrictive iteration-wise dependence relations from the memory location dependence information found by the inspector. Formalizing this, the memory location dependencies define a directed acyclic graph (dag) \( D = (V,E) \) describing the cross-iteration dependencies in the loop: there is a node \( v_i \in V \) for each iteration \( i \) in the loop, and there is a directed edge \( (v_i,v_j) \in E \) if some memory location has a dependence from iteration \( i \) to iteration \( j \). Note that \( D \) is implicit in the reference arrays \( pR \) and their hierarchy vectors \( pH \). A valid parallel execution schedule for a loop is a partition of the set of iterations into ordered subsets called wavefronts, so that all dependences go from an iteration in a lower numbered wavefront to an iteration in a higher numbered wavefront. We say that a valid parallel execution schedule is optimal if it has a minimum number of wavefronts, i.e., is has as many wavefronts as the longest path (the critical path) in the dag.

We remark that the schedulers described below can be used to construct the full iteration schedule in advance (which is how we describe them for simplicity), or alternatively, they can be interleaved with the executor, i.e., the iterations could be executed as they are found to be ready.

#### 6.3.1 A Simple Scheduler

A simple scheduler that finds an optimal schedule is sketched in Figure 6.4(a). In the figure, an array \( \text{wf}(i) \) stores the wavefront found for iteration \( i \), the global variable \( \text{done} \) flags if all iterations have been scheduled, \( \text{rdy}(i) \) signals if iteration \( i \) is ready to be executed, lower case letters \( (a,b) \) are used for references to array elements, \( \text{a.iter} \) is the iteration which contains reference \( a \), and \( \text{Pred}(a) \) is the set of immediate predecessors of \( a \) in the array element dependence graphs. The scheduling is performed in \( \text{cpl} \) phases (line 4) so that in phase \( i \) the iterations belonging to \( i \)th wavefront are identified. In each phase, all the references recorded in the \( pR \) arrays are processed (lines 7–13), and the predecessors of all references whose iterations have not been scheduled (line 8) are examined. An iteration is found not ready if the iterations of any of its reference’s predecessors were not assigned to previous wavefronts (line 10). After all the references are processed, all
4 do while (done.eq..false.)
    rdy(1:numiter) = .false.
    done = .true.
7 doall i = 1,numaccess
      a = access(i)
8 if (wdz(a.iter).eq.0) then
9      for each (b in Pred(a))
10 if (wdz(b.iter).eq.0) done,rdy(a.iter) = .false.
11 endif
12 enddoall
13 doall i = 1,numiter
14 if (rdy(i).eq..true.) wf(i) = cpl
15 enddoall
16 cpl = cpl + 1
17 enddo while

Figure 6.4 A simple scheduler. In (a), \( \text{wf}(i) \) stores the wavefront found iteration \( i \), the global variable \( \text{done} \) flags if all iterations have been scheduled, \( \text{rdy}(i) \) signals if iteration \( i \) is ready to be executed, lower case letters \( (a,b) \) are used for references to memory locations, \( a\.iter \) is the iteration which contains reference \( a \), and \( \text{Pred}(a) \) is the set of immediate predecessors of \( a \) in the memory location dependence graphs. The dependence graph for one of the memory locations accessed in the loop is shown in (b).

the iterations are examined (lines 14–17) to see which can be added to the current wavefront: an iteration \( i \) is ready (line 15) if none of its references set \( \text{rdy}(i) \) to false. Advantages of this scheduler are that it is conceptually very simple and quite easy to implement.

**Optimizing the simple scheduler.** There are some sources of inefficiency in this scheduler. First, since a write access could potentially have many “parent” read accesses it could prove expensive to require such a write to check all of its “parents” (line 9). Fortunately, this problem is easily circumvented by requiring an unscheduled read access to inform its successor’s iteration (the successor, if any, is a write to the same address) that it is not ready. Then, a write access only needs to check its predecessor if the (single) predecessor is also a write.

Another source of inefficiency arises from the fact that each inner doall (lines 7–13) requires time \( O(n_a/p) \) to identify unscheduled iterations (line 8), where \( n_a \) is the total number of accesses to the shared array and \( p \) is the number of processors. Thus, the scheduler takes time \( O((n_a/p) \cdot cpl) \), where \( cpl \) is the length of the critical path. Thus, if \( p = O(cpl) \), then it cannot be expected to offer any speedup over sequential execution, and even worse, it could yield slowdowns for longer critical paths. However, note that in any single iteration of the scheduler, the only iterations that could potentially be added to the next wavefront must have all their accesses at the lowest unscheduled
level in their respective element-wise dependence graphs. For example, consider the dependence graph shown in Figure 6.4(b). If iteration 2 (level 1) has not been scheduled yet, then none of the iterations with accesses in higher levels could be added to the current wavefront.

Thus, in each of the CPL iterations of the outer do while loop, we would like to examine only those references that are in the topmost unscheduled level of their respective dependence graph. First note that we can easily identify the accesses on each level of the array element dependence graphs since references are stored in increasing level order in the pR arrays and the pH arrays contain pointers to the first access at each level. However, to process only the accesses on the lowest unscheduled level it is useful to have a count of the total number of (recorded) accesses in each iteration. This information can easily be extracted in the marking phase and stored in an array indexed by iteration number. Then, in the scheduler, a count of the number of ready accesses for each iteration can be computed on a per processor basis in the first doall (lines 7–13). In the second doall (lines 14–17), the cross-processor sum of the ready access counts for each unscheduled iteration is compared to its total access count, and if they are equal the iteration is added to the current wavefront.

In summary, we would expect this optimized version to outperform the original scheduler if there are multiple levels in them array element dependence graphs, i.e., because it only examines the accesses at the lowest unscheduled level in any iteration of the outer do while. However, note that if there are not many repeated write accesses (and thus few levels), then it is possible that this version could in fact prove inferior to the original (due to the cross-processors summation of the counts). Therefore, the determination of which version to use should be made using knowledge gained about the access pattern by the inspector. These issues are discussed in more detail in Section 6.4.

**Overlapping scheduling and execution.** As mentioned above, the scheduler can construct all the wavefronts in advance or it can be interleaved with the executor so that wavefronts are executed as they are found. A third alternative is to overlap the computation of the wavefronts with the execution of the loop. First, all the processors compute the first wavefront. Then, some processors are assigned to execute the iterations in that wavefront, and the rest of the processors compute the next wavefront. The strategy is carried out repeatedly until all wavefronts are computed. The number of processors assigned to each task would depend upon the amount of work contained in the wavefront. Thus, this approach “fills out” the wavefronts that cannot employ all the processors, i.e., in effect we dynamically merge the parallelism profiles of the wavefront computation and the loop execution to more fully utilize the machine.
Remark: In this section we are mainly concerned with constructing a parallel execution schedule for the iterations of the loop. However, we would like to note that the array element dependence information extracted by the inspector could also be used for producing schedules that overlap iterations or for creating multiple threads of execution.

6.4 Strategies for Applying Run–Time Parallelization

In this section we outline the basic strategy of using the methods in a real application environment.

At Compile-Time.

1. A cost/performance analysis is performed to evaluate whether a speedup can be obtained by these methods (which is not always the case).

2. If the compiler decides to perform run-time parallelization, then an inspector for the marking phase is extracted from the source loop and any other code needed for the methods is generated.

Cost/Performance Analysis. The cost/performance analysis is primarily concerned with evaluating the amount of available parallelism in the loop. Since the data dependence relations between the loop iterations cannot be analyzed statically, an estimate of the available parallelism in the loop can only be made at compile-time using meaningful statistics from previous runs of the program. If the loop is instantiated several times in the same program, then an estimate on the available parallelism in a future instantiation could be made at run-time using statistics from previous invocations of the loop within the same run. For every given (estimated) amount of parallelism, the potential speedup is a function of the ratio between the work of the loop body and the the number of accesses that are shadowed using our methods. The smaller this ratio, the more difficult it will be to obtain a speedup, with the worst case being what we call a “kernel,” i.e., a loop that performs only data movement and no computation. Therefore, in order to obtain a speedup, a substantial amount of parallelism, and sufficient processors to exploit it, are needed.

Instrumentation and Code Generation. For the marking phase, the compiler needs to extract a marking loop, i.e, a parallel loop that traverses the access pattern of the source loop without side effects (without modifying the original data). It is imperative that the marking loop be parallel, for otherwise it defeats the purpose of run-time parallelization [LZ93, SM91]. (Below, we mention some special circumstances in which speedups might still be obtained using a sequential marking loop.) A parallel marking loop can be obtained if the source loop can be distributed into a loop computing
the addresses of the array under test and another loop which uses those addresses (i.e., when the address computation and data computation are not contained in the same strongly connected component of the dependence graph). Unfortunately, in some cases such a marking loop does not exist. In particular, when the data computation in the loop affects future address computations in the loop. After extracting a marking loop, if possible, the compiler augments it with the code for the marking operations, and generates the code for the analysis phase, and for the scheduling and execution of the loop iterations. If a marking loop cannot be extracted, then the compiler must choose between sequential execution and a speculative parallel execution [RP94b].

At Run-Time.

1. At run-time (and possibly also at compile-time) an evaluation of the storage requirements of the methods is performed. If these requirements are prohibitive for the full iteration space of the loop, then the marking loop can be strip-mined and the method (i.e., marking, analysis and scheduling) can be applied to each strip. Even in the case of strip-mining, an optimal schedule can be obtained since the scheduling method can easily be modified to assign iterations in each strip to a single wavefront structure.

2. The marking phase is executed.

3. Using information gathered during the marking phase, the compiler decides whether to continue with run-time parallelization. A lower bound on the length of the critical path is the maximum level (across processors) assigned to any individual array element. If this lower bound is too high, then parallelization should be abandoned and the source loop should be executed sequentially since speedups are unlikely.

4. The analysis phase is executed. Recall that the analysis phase identifies all elements that are independent, read-only, privatizable, or reduction variables, and that accesses to these elements are removed from consideration by the scheduler. If all elements fall into one of these categories, then the loop can be executed as a doall and the scheduling step is omitted.

5. Execute an appropriate scheduler (overlapping it with ready iterations of the source loop). The optimized simple scheduler should prove superior to the original version unless the element-wise dependence graphs have large average degree (see Section 6.3). Since the optimal parallel schedule may be imbalanced (the number of iterations in a wavefront can vary significantly between wavefronts), it is desirable to interleave the scheduler and the executor, i.e., overlap the scheduler’s wavefront computations with the actual execution of the ready iterations.
This can either be achieved with a dynamic partition of the processors among these two tasks (see Section 6.3) or with a dynamic ready queue [MP94, PBK93].

Schedule reuse and decoupling the inspector/scheduler and the executor. Thus far, we have assumed that our methods must be used each time a loop is executed in order to determine a parallel execution schedule for the loop. However, if the loop is executed again, with the same data access pattern, the first schedule can be reused amortizing the overhead of the methods over all invocations. This is a simple illustration of the schedule reuse technique, in which a correct execution schedule is determined once, and subsequently reused if all of the defining conditions remain invariant (see, e.g., Saltz et al. [SMC91]). If it can be determined at compile time that the data access pattern is invariant across different executions of the same loop, then no additional computation is required. Otherwise, some additional computation must be included to check this condition, e.g., for subscripted subscripts the old and the new subscript arrays can be compared. Although a parallel marking loop is always desirable, if schedule reuse can be applied then it may still be possible to obtain speedups with a sequential marking loop since its one sequential execution will be amortized over all loop instantiations.

Another method to reduce the cost associated with these methods is to hide their overheads by executing them as soon as all the necessary data is available. If this type of decoupling is possible, then the inspector phase could be overlapped with other portions of the program—thereby more fully exploiting the processing power of the machine (of course support for MIMD execution is highly desirable in this case).

6.5 Experimental Results

In this section we present experimental results obtained on two modestly parallel machines with 8 (Alliant FX/80 [All86]) and 14 processors (Alliant FX/2800 [All91]). However, we remark that the results scale with the number of processors and the data size and thus they may be extrapolated for massively parallel processors (MPPs), the actual target of our run-time methods.

To demonstrate that the new methods can achieve speedups, we applied them to three loops contained in the PERFECT Benchmarks [BCK+89] that could not be parallelized by any compiler available to us. In addition, in order to analyze the overhead incurred by the methods, we applied them to different access patterns taken from loops in the PERFECT Benchmarks and to synthetic access patterns generated to test their behavior in various situations.

The methods were implemented in Cedar Fortran [GPHL90]. The inspector was essentially as described in Section 6.2. In particular, we implemented the bucket sort version using separate $pR$
and pH data structures for each processor. To avoid checking each row in $pR$ during the analysis phase of the inspector and in the scheduler, each processor constructed a linked list of the non-empty rows in its $pR$ array during the marking phase. Checks for independent, read-only, and privatizable elements were implemented in the inspector (we did not yet incorporate the test for reduction variables). In the analysis phase, these elements are classified at the same time that the predecessors and successors are found for each row. One optimization that we did not yet implement was the “pair-wise” merge across processors when searching for predecessors or successors in the analysis phase (or when classifying elements as independent, read-only, or privatizable). However, this is an important optimization since, as previously noted, without it the analysis phase of the inspector may fail to scale with the number of processors. Since we implemented the optimized version of the simple scheduler described in Section 6.3, a count of the total number of accesses in each iteration was computed in the marking phase (no inter-processor communication is needed to determine these counts since each iteration is assigned to a single processor). For simplicity, the scheduler and the executor were completely decoupled in the implementation. In general, however, better speedups should be obtainable by interleaving these two tasks (see Section 6.3).

6.5.1 Synthetic Access Patterns

Using synthetic loops, we now study the sensitivity of the overhead of the methods to two characteristics of the source do loop: its average parallelism (the number of iterations divided by the number of wavefronts in an optimal parallel execution schedule) and its hotspot degree (the maximum number of repeated accesses to any array element). To simplify the generation of the synthetic workloads, we did not identify independent, read-only, or privatizable elements in the analysis phase. This should not affect our conclusions, however, since these computations can be folded into the searches for predecessors and successors (with little extra work).

**Average parallelism.** To isolate the affect of the average parallelism in the source loop on the overhead of the methods, we generated access patterns that were as similar as possible in all aspects except for the average parallelism. In particular, there were two accesses in every iteration (a read followed by a write), and every array element was accessed approximately twice (at some boundary conditions some elements are accessed either once or three times).

First, we would not expect the inspector execution time to be dependent on the average parallelism in the loop. In the marking phase each processor marks $n_a/p$ accesses in its private shadow array (to isolate the effects of the average parallelism, we assume that the marking phase is balanced). The overhead of the analysis phase is primarily dependent upon the number of distinct
array elements marked in its $pR$ array (since it must find successors and predecessors for each non-empty row). Thus this overhead might vary inversely with the hotspot degree, but it is not necessary dependent on the average parallelism because for the same critical path length the hotspot degree can be anywhere between 2 and the number of iterations. In Figures 6.15 and 6.16 we display results from a loop with 2048 iterations run on 10 processors. The plot shows the overhead incurred for a loop with a critical path length of “Step” (the average parallelism is the number of iterations divided by the critical path length). As expected, the overhead of the inspector is invariant with the length of the critical path, and that of the scheduler grows linearly with this length.

We now consider how the speedup of the overheads relates to the average parallelism. Since the execution time of the inspector is independent of the average parallelism, its speedup should not depend on it either. Even though the scheduling time does depend on the average parallelism, its speedup is not necessarily similarly correlated. This is because each wavefront is calculated in doall loops, i.e., each of iteration of the scheduler can be expected to obtain reasonable speedups, and thus the overhead of the scheduler as a whole can be expected to obtain good speedups as well. In Figures 6.17 and 6.18 we show the speedup obtained for the inspector and executor, respectively, on a loop with 2048 iterations and three different values of average parallelism. In both cases the similar speedups are obtained for the sequential loop (average parallelism 1) and the loop that is almost fully parallel (average parallelism 1024). In Figures 6.19 and 6.20 we show analogous results on a loop with 1024 iterations. Recall that in our implementation we did not use a “pair-wise” merge among the processors, i.e., in our implementation each processor checks all $p - 1$ other processors for predecessors and successors whereas in the pair-wise merge only $O(\log p)$ operations would be needed. This fact is most likely the cause of the slightly diminished slope of the speedup curve after about 10 processors for the overhead of the inspector.

**Hotspots.** To isolate the effect of the hotspot degree in the source loop on the overhead of the methods, we generated access patterns that were as similar as possible in all aspects except for the hotspot degree. In particular, all loops had 2048 iterations, two accesses in each iteration, and an average parallelism of 51 (a critical path length of 40). Also, a loop with hotspot value $h$ contained $h$ references to each of $n/h$ array elements, where $n = 2048$ is the number of iterations in the loop. In principle, we would not expect our methods to be negatively affected by the hot spot degree. In fact, a larger the hotspot degree implies fewer non-empty rows in the $pR$ array, and thus we might see improved results in the analysis and scheduling phase since few rows would need to be accessed. The results in Figure 6.14 show that in fact the total overhead (inspector + scheduler) is nearly the same for all hotspot degrees.
6.5.2 Real Access Patterns

Now we would like to look at access patterns arising in real applications to demonstrate the diversity of partially parallel access patterns and their associated parallelism profiles. By applying the new methods to such access patterns, we can re-confirm the conclusions reached above using synthetic reference patterns. For this purpose we have chosen a loop out of MA28, a blocked sparse UN-symmetric linear solver [Duf77]. Loop MA30cd/DO_120 performs the forward-backward substitution in the final phase of the blocked sparse linear system solver (MA28). We selected this loop because it can generate many diverse access patterns when using the Harwell-Boeing matrices as input. Unfortunately, however, the loop itself is not a good candidate for parallelization since it performs very little work and is highly imbalanced due to the blocked nature of the algorithm employed by MA28.

We will limit our discussion below to two input sets: gemat12, which generates 4929 iterations, and bp_1600, which generates 822 iterations. After extracting and precomputing the linear recurrences from the source loop (based on the methods described in [RP94c]), we generated a fully parallel inspector and applied our methods to compute an optimal parallel execution schedule for the loop.

From the data obtained we constructed the parallelism profiles depicted in Figures 6.5 and 6.6. These profiles depict the size of the wavefronts of the optimal parallel execution schedule. As we can see from the figures, the same loop can have vastly different dependence relations between its iterations. These figures clearly point out both the need for run-time analysis techniques and for dynamic and adaptive scheduling schemes capable of overlapping scheduling and execution. Figure 6.5 shows that most of the iterations of the loop can be executed in the initial wavefronts (the critical path length is 114). This suggests that in this case it would be more beneficial to interleave the parallel wavefront computation with the execution of previous wavefronts than it would be to overlap them, so that parallelization (and its associated overhead) can be abandoned when the sequential tail of the profile is reached. Although in Figure 6.6 most of the iterations are also executed in the initial wavefronts, in this case it appears that some benefit could be gained by overlapping, i.e., we can take advantage of the “pauses” in parallelism to compute future (hopefully larger) wavefronts. The histograms in Figures 6.7 and 6.8 underscore the need for scheduling and execution strategies that can be dynamically adapting depending upon the type of parallelism encountered to more fully utilize the machine.

Despite the differences in the parallelism profiles Figures 6.9 and 6.10 show that the overhead of the run-time methods described in this paper achieve similar performance. The reason that larger
speedups were not obtained is that the loop is heavily imbalanced due to the blocked nature of the algorithm used in MA28.

6.5.2.1 Parallelizing Benchmark Loops

We applied the methods to three loops contained in the PERFECT Benchmarks [BCK+89] that could not be parallelized by any compiler available to us. In the analysis phase of the inspector it was found that one of the loops was fully parallel, and that the other two could be transformed into doalls by privatizing the shared array under test. We show in Figures 6.11 through 6.13 the speedup measured for each loop as a function of the number of processors used. As a reference, we give the ideal speedup, which was measured using an optimally parallelized (by hand) version of the loop. These graphs show that the speedup scales with the number of processors and is a significant percentage of the ideal speedup. Below, we discuss each loop in more detail.

We remark here that these loops could also be identified by the LRPD test [RP94a, RP94b], a run-time test for identifying fully parallel loops, or loops that can be transformed into doalls using privatization and reduction parallelization. An advantage of the LRPD test is that it has a smaller overhead than the methods we present here. The disadvantage of the LRPD test is that if the loop cannot be transformed into a doall, then the overhead of applying the method is added to cost of the sequential execution, i.e., a slight “slowdown” may be incurred. Ideally, in order to exploit the relative advantages of the two methods, one would like to apply them both simultaneously.

BDNA–ACTFOR–Loop 240. This loop selects certain elements from a large array, and processes the selected elements later in the loop. The shared array is accessed through a subscript array that is computed inside the loop (and thus cannot be analyzed at compile-time). Although there are repeated accesses in this loop, it is determined in the analysis phase of the inspector that the entire shared array is privatizable, i.e., that the loop can be transformed into a doall by privatizing the array. As shown in Figure 6.11, the obtained speedup scales with the number of processors and is a significant percentage of the ideal speedup.

MDG–INTERF–Loop 1000. This loop calculates inter-molecular interaction forces. In the marking loop, to avoid introducing false dependences we computed the branch predicates that guard accesses to the shared array under scrutiny. As with the array in the loop from BDNA, it is found in analysis phase of the inspector that the entire shared array is privatizable. The speedup obtained scales with the number of processors and is a significant fraction of the ideal (see Figure 6.12).
OCEAN–FTRVMT–Loop 109. This kernel-like loop is utilized in the computation of a 2-dimensional FFT and accesses a vector with run-time determined strides. During the analysis phase of the inspector it is found that all accesses in the loop are unique, i.e., it is a fully parallel loop. Since this loop is invoked 26,000 times, and accounts for 40% of the sequential execution time of the program, it is an excellent candidate for schedule reuse (see Section 6.4). The access pattern for each instantiation of the loop is determined by a set of five scalars. In order to apply schedule reuse, we checked whether the current set of scalars matched a previously analyzed set. If not, then we applied the parallelization techniques, and if they did match then we simply executed the loop as a doall. As can be seen in Figure 6.13, with schedule reuse we obtain scalable speedups that are comparable to the ideal speedup.

6.6 Future Improvements

In this chapter we have presented a method for the parallelization of partially loops that applies a fully parallel inspector, uses no synchronization, and can be applied to any loop (from which an inspector can be extracted). In addition, it can implement at run-time the two most effective transformations for increasing the amount of parallelism in a loop: array privatization (element-wise) and reduction parallelization. The scheduler/executer constructs an optimal parallel execution schedule for the iterations of the loop. Although the wavefronts of the schedule are constructed in sequence, the computation of each wavefront is fully parallel and requires no synchronization. These new methods improve on all previously proposed techniques since none of them simultaneously has all these features (Section 7.2).

Although Section 6.5 illustrates through experiments the potential benefits of run-time parallelization, there is still much work left to be done. For example, there are many potential scheduling strategies that need to be studied such as decoupling the inspector/scheduler and the executor in order to hide the overheads, dynamically overlapping scheduling and execution, or, constructing parallel threads of execution (as opposed to wavefronts). Further investigation is needed to determine the relative performance of the various scheduling techniques for different dependence structures of a loop. Another important task is to devise effective, automatable strategies for determining when and how to combine the LRPD test with the technique presented in this chapter. Both methods can detect parallelism but their cost is quite dissimilar. It remains to be studied whether first testing for full parallelism and then for partial parallelism is better than trying only one of the methods.
Finally we think that some related technique can be devised that could overcome the necessity of an inspector for partially parallel loops (for example through a speculative approach) and therefore become applicable to any loop.
Figure 6.5 Parallelism profile

Figure 6.6 Parallelism profile

Figure 6.7 Parallelism histogram

Figure 6.8 Parallelism histogram

Figure 6.9 Overhead speedup

Figure 6.10 Overhead speedup
Figure 6.11 Speedup

Figure 6.12 Speedup

Figure 6.13 Speedup

Figure 6.14 Overhead with, without hotspots

Figure 6.15 Marking, analysis phase overhead

Figure 6.16 Scheduling overhead
Figure 6.17 Marking, analysis phase speedup

Figure 6.18 Scheduling phase speedup

Figure 6.19 Marking, analysis phase speedup

Figure 6.20 Scheduling phase speedup
Chapter 7

Previous Work in Run–Time Concurrency Detection

7.1 Hardware Implementations

Run–time techniques have been used practically from the beginning of parallel computing. During the 1960s, relatively simple run–time techniques, used to detect parallelism between scalar operations, were implemented in the hardware of the CDC 6600 and the IBM 360/91 [Tho71, Tom67]. Later more aggressive designs have incorporated a full-empty bit [Smi87] to enforce data dependences on the fly. This type of memory structure has been exploited by the data-flow computing model [AC86, AI86, AN86] in quite an original way. It is probably also the most fundamental model of exploiting parallelism.

Another hardware approach that is useful for run-time parallelization was taken by the Cedar project with its hardware synchronization primitives [ZYL83, ZY84, TZ84].

Today, microprocessors are all implementing hardware schemes that permit out-of-order execution while enforcing data dependences. Some of the latest ones (e.g. PA-RISC from Hewlett Packard) are using a tagged register file that allows speculative loads and enforce flow-dependences.

Finally we should mention the new approach taken by G. Sohi at the University of Wisconsin [SBV95]. It is a speculative approach for executing iterations of a loop in parallel across multiple functional units. If an incorrect out-of-order memory access occurs (checked through hardware mechanisms) then all iterations and their effects are squashed and work restarts from that point.

7.2 Software Methods for Run–Time Parallelization of Partially Parallel Loops

Simple run-time compiler techniques for using multi-version loops have been in production for quite some time. Many of today’s parallelizing compilers postpone part of their analysis for the run-time phase by generating two-version loops. These consist of an if statement that selects either the original serial loop or its parallel version. The boolean expression in the if statement typically tests the value of a scalar variable.

As mentioned in Section 2.1, most previous approaches to run–time parallelization have concentrated on developing methods for constructing execution schedules for partially parallel loops, i.e., loops whose parallelization requires synchronization to ensure that the iterations are executed in the correct order. Most of the developed schemes partition the set of iterations into subsets called stages, so that the iterations in each stage can be executed in parallel, i.e., there are no data
dependences between iterations in a stage. Stages formed by a regular pattern of iterations are named wavefronts. The wavefronts themselves are executed sequentially by placing a synchronization barrier between each them.

In the following, we briefly describe some of the previous methods, placing particular emphasis on the differences from and similarities to our methods. A high level comparison of the various methods is given in Table 7.1.

7.2.1 Methods Utilizing Critical Sections

One of the first run–time methods for scheduling partially parallel loops was proposed by Zhu and Yew [ZY87]. It computes the wavefronts one after another using a method similar to the simple scheduler described in Section 6.3.1. During a phase, an iteration is added to the current wavefront if none of the data accessed in that iteration is accessed by any lower unassigned iteration; the lowest unassigned iteration to access any array element is found using atomic compare–and–swap synchronization primitives and a shadow version of the array. Midkiff and Padua [MP87] extended this method to allow concurrent reads from a memory location in multiple iterations. Due to the compare–and–swap synchronizations, this method runs the risk of a severe degradation in performance for access patterns containing hot spots (i.e., many accesses to the same memory location). However, when there are no hot spots and the critical path length is very small, then this method should perform well. An advantage of this method is reduced memory requirements: it uses only a shadow version of the shared array under scrutiny whereas all other methods (except [Pol88, RP94a, RP94b]) unroll the loop and store all the accesses to the shared array.

Krothapalli and Sadayappan [KS88] proposed a run–time scheme for removing anti and output dependences from loops. Their scheme includes a parallel inspector that determines the number of accesses to each memory location using critical sections as in the method of Zhu and Yew (and is thus sensitive to hotspots). Using this information, for each memory location, they place all accesses to it in a dynamically allocated array and then sort them according to iteration number. Next, the inspector builds a dependence graph for each memory location (similar to our arrays \( R_x \)), dynamically allocates any additional global storage needed to remove all anti and output dependences (using renaming), and explicitly constructs the mapping between all the memory accesses in the loop and the storage, both old and new, thereby inserting an additional level of indirection into all memory accesses. The loop is executed in parallel using synchronization (full/empty bits) to enforce flow dependences. To our knowledge, this is the only other run–time privatization technique except [RP94a, RP94b].
Recently, Chen, Yew, and Torrellas [CYT94] proposed an inspector that has a private phase and a merging phase. In the private phase, the loop is chunked and each processor builds a list of all the accesses to each memory location for its assigned iterations. This is similar to the private marking phase of our inspector except that they serialize read accesses (i.e., they have a list instead of the dependence graph). Next, the lists for each memory location are linked across processors using a global Zhu/Yew algorithm [ZY87]. Their scheduler/executor uses doacross parallelization [SM91], i.e., iterations are started in a wrapped manner and processors busy wait until their operands are ready. Although this scheme potentially has less communication overhead than [ZY87], it is still sensitive to hot spots and there are cases (e.g., doalls) in which it proves inferior to [ZY87]. For example, consider a loop with \( cpl = p \) and dependence distance \( p \) as well, i.e., \( r_a^i = p \) and \( u_a = n/p \) so that each processor’s iterations access the same set of \( n/p \) distinct memory locations. In this case the new inspector requires time \( O(np) \), the original Zhu/Yew scheme uses time \( O(p^2 + n) \), and, as usual, the sorting-based inspector needs time \( O((n/p) \log p) \). Although this example may appear a bit contrived, it is actually a quite realistic possibility. Consider for example a nested loop in which the inner loop loop is fully parallel and moreover, it always accesses the same memory locations. Thus, if, as would be natural, each processor were assigned one iteration of the outer loop, we would have precisely the situation described above.

7.2.2 Methods for Loops Without Output Dependences

The problem of analyzing and scheduling loops at run-time has been studied extensively by Saltz et al. [BS90, SM91, SMC89, SMC91, WSHB91]. In most of these methods, the original source loop is transformed into an inspector, which performs some run-time data dependence analysis and constructs a (preliminary) schedule, and an executor, which performs the scheduled work. The original source loop is assumed to have no output dependences. In [SMC91], the inspector constructs stages that respect the flow dependences by performing a sequential topological sort of the accesses in the loop. The executor enforces any anti-dependences by using old and new versions of each variable. Note that the anti dependences can only be handled in this way because the original loop does not have any output dependences, i.e., each variable is written at most once in the loop. The inspector computation (the topological sort) can be parallelized somewhat using the DOACROSS parallelization technique of Saltz and Mirchandaney [SM91], in which processors are assigned iterations in a wrapped manner, and busy-waits are used to ensure that values have been produced before they are used (again, this is only possible if the original loop has no output dependences).
Recently, Leung and Zahirjan [LZ93] have proposed some other methods of parallelizing the inspector of Saltz et al. These techniques are also restricted to loops with no output dependences. In sectioning, each processor computes an optimal parallel schedule for a contiguous set of iterations, and then the stages are concatenated together in the appropriate order. Thus sectioning will usually produce a suboptimal schedule since a new synchronization barrier is introduced into the schedule for each processor. In bootstrapping, the inspector of Saltz et al. (i.e., the sequential topological sort) is parallelized using the sectioning method. Although bootstrapping might not optimally parallelize the inspector (due to the synchronization barriers introduced for each processor), it will produce the same minimum depth schedule as the sequential inspector of Saltz et al.

Other methods In contrast to the above methods which place iterations in the lowest possible wavefront, Polychronopoulos [Pol88] gives a method where wavefronts are maximal sets of contiguous iterations with no cross-iteration dependences. Dependences are detected using shadow versions of the variables, either sequentially, or in parallel with the aid of critical sections as in [ZY87].

Another significant contribution to this field has been done by Alex Nicolau in [Nic89]. The idea of run-time disambiguation has been more recently used in optimizing codes for instruction level parallelism [HSS94, GCM+94]. Their idea is to speculatively execute code very aggressively (out of order) despite the fact that some memory locations (few) could cause unsatisfied data dependences. The offending addresses which are used out of order are stored until all potential hazards have been cleared. If an error is detected repair code will backtrack and restart from that point.

In summary, the previous run-time methods for parallelizing loops rely heavily on global synchronizations (communication) [CYT94, KS88, LZ93, MP87, Pol88, SM91, SMC91, ZY87], are applicable only to restricted types of loops [LZ93, SM91, SMC91], have significant sequential components [Pol88, SM91, SMC91], and/or do not extract the maximum available parallelism (they make conservative assumptions) [CYT94, LZ93, Pol88, SM91, SMC91, ZY87].

Table 7.1 gives a more graphic overview of the previous work in run-time parallelization and compares it to our method.

7.2.3 Related Work

Hazard Detection Significant amount of work has been invested in the research of hazards (race conditions), access anomalies for debugging parallel programs. Generally, access anomaly detection techniques seek to identify the point in the parallel execution in which the access anomaly occurred.
Table 7.1 A comparison of run-time parallelization techniques for do loops. In the table entries, $P$ and $R$ show that the method identifies privatizable and reduction variables, respectively. The superscripts have the following meanings: 1, the method serializes all read accesses; 2, the performance of the method can degrade significantly in the presence of hotspots; 3, the scheduler/executor is a doacross loop (iterations are started in a wrapped manner) and busy waits are used to enforce certain data dependences; 4, the inspector loop sequentially traverses the access pattern; 5, the method is only applicable to loops without any output dependences (i.e., each memory location is written at most once); 6, the method only identifies fully parallel loops.

Padua et al. [AP87, EGP92] discuss methods that statically analyze the source program, and methods that analyze an execution trace of the program. Since not all anomalies can be detected statically, and execution traces can require prohibitive amounts of memory, run-time access anomaly detection methods that minimize memory requirements are desirable [Sch89, DS90, NR88]. In fact, a run-time anomaly detection method proposed by Snir, and optimized by Schonberg [Sch89], bears similarities to the version of the doall test presented in Chapter 2 (i.e., the version without the privatization). However, in order to identify the point in the execution in which the anomaly occurred, their methods [Sch89] require much more memory than the doall test, e.g., viewed in the framework of the doall test, a separate shadow array for each iteration in a loop must be maintained. Another difference is that the doall test is optimized especially for loops, and access anomaly detection methods must handle any type of concurrent thread in a parallel program.

**Optimistic Execution** A concept related to the speculative approach proposed in this thesis is virtual time first introduced in [Jef85] and defined as ” ... paradigm for organizing and synchronizing distributed systems.... [It] provides a flexible abstraction of real time in much the same way that virtual memory provides an abstraction of real memory. It is implemented using the Time Warp
mechanism, a synchronization protocol distinguished by its reliance on look-ahead rollback, and by its implementation of rollback via antimesages.” We think that the granularity and overhead associated with this method is more applicable to such problems as discrete event simulation and database concurrency control rather than loop parallelization. In fact this concept has been applied in database design.
Chapter 8

Parallelizing While Loops for Multiprocessor Systems

8.1 Introduction

Most current parallelizing compilers treat while loops and do loops with conditional exits as sequential constructs. Since these types of loops arise frequently in practice, techniques for extracting their available parallelism are highly desirable.

In the most general form, we define a while loop as a loop that includes one or more recurrences that can be detected at compile time, a remainder, whose dependence structure can be either analyzed statically (as being parallel or sequential) or is unknown at compile time, and one or more termination conditions. Sometimes the termination conditions form part of one of the recurrences, but they can also occur in the remainder, e.g., conditional exits from do loops. Assuming, for simplicity, that there are no cross-iteration data dependences in the remainder, there are two potential problems in the parallelization of while constructs:

- *Evaluating the recurrences.* If the recurrences cannot be evaluated in parallel, then the iterations of the loop must be started sequentially, leading in the best case to a pipelined execution (also known as a doacross).

- *Evaluating the termination conditions.* If the termination conditions (loop exits) cannot be evaluated independently by all iterations, the parallelized while loop could continue to execute beyond the point where the original sequential loop would stop, i.e., it can overshoot.

Although the concurrent evaluation of recurrences is in general not possible, some special cases lend themselves to either full or partial parallelization. There are parallel algorithms to solve simple inductions (the case of do loops) [WL90] and associative recurrences [CKS78, LF80, Kru85, Kru86] but the evaluation of general recurrences has always been of a sequential nature. The concurrent evaluation of the while loop termination condition has been dealt with only in the case when it is loop invariant with respect to the remainder (a property we define later as remainder invariant). In other words, the exit conditions that have been dealt with so far are those dependent on the terms of the recurrence, and since these recurrences are executed sequentially, the exit conditions do not pose a problem for parallelization.

The task of parallelizing a while loop becomes even more difficult if the data dependence structure in the remainder cannot be determined statically. For example, there may exist additional
recurrences in the the remainder that cannot be detected by the compiler. For input data dependent irregular access patterns this problem is intractable with traditional compile-time methods and has not been addressed so far.

In this chapter we present a general framework for the automatic transformation of any while loop for parallel execution, provided that its remainder is indeed parallel. The basic strategy of our methods is to evaluate in parallel the recurrences that can be statically identified and speculatively execute remainder concurrently, and then later, to "undo" the effects of any iterations that overshot the termination condition, i.e., iterations that would not have been performed by the original sequential version of the loop. We describe techniques for parallelizing loops involving linked list traversals. This is an important problem since linked list traversals arise frequently in loops with irregular access patterns, such as sparse matrix computations. In many cases, our parallelization of loops involving linked lists can be done without overhead or side effects.

Our framework for while loop parallelization can be viewed as a step toward providing while loop counterparts for the existing constructs for parallel execution of do loops, e.g., doall, doacross, doany. These new parallel constructs could be called while-doall, while-doacross, and while-doany and could prove useful in the parallel programming (manual parallelization) of applications.

The methods described here extend previous works [Har86, WL90] in that they:

1. can handle remainder variant termination conditions,
2. can test at run-time for cross-iteration data dependences in the remainder,
3. do not require work and storage for saving the values computed in the recurrence,
4. support both static and dynamic scheduling, and
5. present a comprehensive analysis and solution package for parallelizing while loops for multiprocessors.

The techniques are capable of extracting a substantial fraction of available parallelism in a loop. In particular, we show that in the worst case our techniques will extract at least 20-25% of the parallelism inherent in the loop, which can amount to significant speedups on massively parallel processors. Therefore, we conclude that as long as there is sufficient available parallelism in the loop, our techniques will prove to be beneficial. We have obtained experimental results on loops from the PERFECT Benchmarks and sparse matrix packages on the Alliant FX/80 which substantiate this conclusion.

We begin in Section 8.2 by introducing a taxonomy of while loops based on the difficulties they present for parallelization. Then, in Sections 8.3 and 8.4, for each case in the taxonomy we give
the necessary transformations and methods for automatically parallelizing the while loop, under
the assumptions that there are no cross-iteration data dependences in remainder, and that there is
only one recurrence controlling the loop. In Section 8.5, these methods are augmented to include
loops whose access patterns cannot be analyzed at compile-time, and in Section 8.6 we describe
how loops with an arbitrary number of recurrences can be handled. In Section 8.7, we discuss a
cost/performance analysis that can be used to determine when our methods should be applied, and
in Section 8.8 we give some strategies for reducing the overhead of our methods. We present some
experimental results in Section 8.9. In Section 8.10 we discuss related work.

8.2 Transforming While Loops for Parallel Execution

While loops have often been treated by parallelizing compilers as an intrinsically sequential
constructs because their iteration space is unknown [Har86]. A related case which is generally
also handled sequentially by compilers is the do loop with a conditional exit. In this paper we
propose techniques that can be used to execute such loops in parallel. In order to clarify our
presentation we first consider loops which (a) contain a single statically detectable recurrence, and
(b) have no cross-iteration data dependences except those in this recurrence. Later, we relax these
constraints and show how to deal with loops with multiple recurrences and unknown cross-iteration
data dependences.

Figure 8.1 While loop examples

In this case, a while loop can be considered as a parallel loop controlled by a recurrence. In
general while loops can exhibit several dependent or independent (of one another) recurrences.
We call the dominating recurrence, which precedes the rest of the computation in the dependence
graph, the dispatching recurrence, or simply the dispatcher (see Figure 8.1(a)). In the most general
case, the terms of the dispatcher must be evaluated sequentially. An example of this case is a
pointer used to traverse a linked list; since the values of the dispatcher (the pointer) must be
evaluated in sequential order, iteration \( i \) of the loop cannot be initiated until the dispatcher for
iteration \( i - 1 \) has been evaluated (see Figure 8.1(b)). However, sometimes the evaluation of
the terms of the dispatching recurrence can be parallelized. In particular, if the dispatcher is an
associative recurrence, then the computation of its terms can be parallelized using techniques such
as parallel prefix computations (see Figure 8.1(c)). Finally, in the best case, the dispatcher has the
simpler form of an induction, and each point in the dispatcher’s domain can be independently and
concurrently evaluated using the closed form solution of the induction. In this case, all iterations
of the while loop can be executed simultaneously since aside from the dispatching recurrence we
assumed no other dependences. An example of a dispatcher with a closed form solution is a do
loop (see Figure 8.1(d-e)).

Another difficulty with parallelizing a while loop is that the termination condition (terminator)
of the loop may be overshot, i.e., iterations could be executed that would not be executed by the
sequential version of the loop. In the context of our analysis we define the terminator as remainder
invariant or RI if it is only dependent on the dispatcher and values that are computed outside the
loop; if it is dependent on some value computed in the loop then it is considered to be remainder
variant or RV. If the terminator is RV, then iterations larger than the last valid iteration could
be performed in a parallel execution of the loop, i.e., iteration \( i \) cannot decide if the terminator is
satisfied in the remainder of some iteration \( i' < i \). Overshooting may also occur if the dispatcher
is an induction, or an associative recurrence, and the terminator is RI. An exception in which
overshooting would not occur is if the dispatcher is a monotonic function, and the terminator
is a threshold on this function, e.g., \( d(i) = i^2 \), and \( tc(i) = (d(i) < V) \), where \( V \) is a constant,
and \( d(j) \) and \( tc(j) \) denote the dispatcher and the terminator, respectively, for the \( j \)th iteration.
Overshooting can also be avoided when the dispatcher is a general recurrence, and the terminator
is RI. For example, the dispatcher \( tmp \) is a pointer used to traverse a linked list, and the terminator
is \( (tmp = null) \) (see Figure 8.1(b)). In the most general case, the exit from a while loop may be
caused by one of many termination conditions; this situation which will require a combination of
several solutions.

From the discussion above we conclude that the techniques needed to parallelize a while loop
depend on the type of its dispatcher and terminator. We can therefore summarize our discussion
through the taxonomy of while loops given in Table 8.1.
<table>
<thead>
<tr>
<th>Loop Terminator</th>
<th>Monotonic Induction</th>
<th>Not Monotonic Induction</th>
<th>Associative Recurrence</th>
<th>General Recurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Overshoot</td>
<td>Parallel</td>
<td>Overshoot</td>
<td>Parallel</td>
</tr>
<tr>
<td>RI</td>
<td>NO</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
<tr>
<td>RV</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
<td>YES</td>
</tr>
</tbody>
</table>

Table 8.1 A taxonomy of while loops and their dispatcher’s potential for parallel execution. The notation PP implies parallelizable with a parallel prefix computation.

In the next section we discuss methods for parallelizing while loops under the two previously mentioned assumptions:

1. There is one and only one recurrence (dispatcher), which can be detected statically

2. The only cross-iteration data dependences in the loop are carried by the controlling recurrence (the dispatcher)

Later, in Sections 8.5 and 8.6, we show how our methods can be generalized when these two restrictive conditions are relaxed.

8.3 Parallelizing the Dispatcher

Clearly, the most important factor affecting the amount of available parallelism in a while loop (assuming no cross-iteration dependences) is the amount of parallelism available in its dispatching recurrence. To aid our analysis of the dispatching recurrence, it is convenient to extract, at least conceptually, this recurrence from the original while loop by distributing [Wol89] the original loop into two do loops with conditional exits:

1. A loop that evaluates the terms of the dispatcher (recurrence) and any termination condition that is strongly connected to the dispatcher.

2. A loop consisting of the remainder loop which uses the values of the recurrence (computed by the first loop), and its associated termination condition.

Note that the original set of termination conditions is distributed appropriately to the two loops. Thus, the first loop may or may not have a termination condition, and the second loop is either a simple do loop, or a do loop with a conditional exit.

In order to perform the data dependence analysis necessary for loop distribution all array references in the while loop have to be associated with a loop counter. We remark that a proper
distribution is not possible ([KM90]) if the dependence structure of body of the loop consists of a
single strongly connected component. In either case, for the purposes of parallelizing the dispatcher,
the techniques proposed in the sequel remain the same.

Once the dispatcher has been extracted we can attempt to parallelize it. As discussed in the
previous section, the extent to which this is possible depends upon the form of the recurrence itself.
In its most general form a recurrence can be evaluated only through a sequential method. However,
if the recurrence is associative, then parallel algorithms like parallel prefix computations can speed
the task of computing its terms by a significant factor, and if the recurrence is an induction, then
its evaluation can be done in fully concurrent mode by evaluating its closed form.

In the remainder of this section we present techniques that can be used to extract the maximum
available parallelism from while loops with one dispatching recurrence. For the cases in which
the dispatcher is not an induction, our methods assume that the dispatching recurrence is fully
determined before loop entry (e.g., if the dispatcher is traversing a linked list, no list elements may
be inserted or deleted during loop execution). Although not all of our methods are fully parallel,
they can yield very good speedups – especially if a significant amount of work is performed in the
loop body.

We describe the methods without addressing the overshooting problem, and then discuss in
Section 8.4 how they can be augmented to “undo” any iterations that overshot the termination
condition. We also assume that there are no cross-iteration dependences in the loop other than
those associated with the dispatcher. This restriction is removed in Section 8.5 where we describe
how while loops can be speculatively executed in parallel by combining our methods with run-time
techniques for detecting the presence of cross-iteration dependences in the loop.

Finally we should consider the case when the loop evaluating the recurrence does not contain a
termination condition and therefore does not in itself impose an upper limit on the number of terms
to be computed. In this case an upper bound can be inferred from the body of the while loop. If
that is not possible then the two distributed loops can be executed in a strip-mined fashion until
the termination condition is reached, effectively limiting the number of precomputed recurrence
terms to the length of the strip.

8.3.1 The Dispatcher is an Induction

In this section we consider a while loop in which the dispatcher $d(i)$ is an induction of the
generic form:

\[ d(i) = c \times i + b \]
where \( c \) and \( b \) are constants. To simplify our discussion, we assume that the dispatcher of the \( i \)th iteration is \( i \), i.e., \( d(i) = i \). The fact that all processors can evaluate the dispatcher simultaneously from a closed form solution of the induction relation makes loop distribution and precomputation of the recurrence terms unnecessary.

\[
\text{*while Loop: induction*} \\
\text{integer } i = 1 \\
\text{while } ((f(i)) \\
\quad \text{work}(i) \\
\quad i = i + 1 \\
\text{endwhile}
\]

\[
\text{*Induction-1*} \\
\text{integer } L[0:nproc-1] = u \\
\text{doall } i = 1, u \\
\quad \text{if } (L[vpn] > i) \text{ then} \\
\quad \quad \text{if } (\text{not } f(i)) \text{ then } L[vpn] = i \\
\quad \quad \text{else work}(i) \\
\quad \text{endif} \\
\text{enddo} \\
L_i = \min(L[1:nproc])
\]

\[
\text{*Induction-2*} \\
\text{integer } L[0:nproc-1] = u \\
\text{doall } i = 1, u \\
\quad \text{if } (\text{not } f(i)) \text{ then} \\
\quad \quad L[vpn] = i \\
\quad \text{QUIT} \\
\quad \text{endif} \\
\quad \text{work}(i) \\
\text{enddo} \\
L_i = \min(L[1:nproc])
\]

**Figure 8.2** While Loop with inductive dispatcher. In the doalls, \( nproc \) is the number of processors, \( u \) is an upper bound on the number of iterations of the while loop, and \( vpn \) is the virtual processor number of the processor executing the iteration.

In this method, referred to as Induction-1, the loop is run as a doall and a test of the termination condition of the while loop is inserted into the loop body (see Figure 8.3.1). During the parallel execution, each processor keeps track of the lowest iteration it executed that met the termination condition. Then, after the doall has terminated, the last iteration that would have been executed by the sequential version of the while loop is found by taking the minimum of the processor-wise minima. This iteration must be found so that any iterations that need to be undone can be identified.

On computers, such as the Alliant [All86], in which iterations are issued in order, the test \( L[vpn] > i \) is unnecessary. In order to terminate the parallel loop cleanly before all iterations have been executed, a QUIT operation similar to the one on Alliant computers [All86] could be used. Once a QUIT command is issued by an iteration, all iterations with loop counters less than that of the issuing iteration will be initiated and completed, but no iterations with larger loop counters will be begun. If multiple QUIT operations are issued, then the iteration with the smallest loop counter executing a QUIT will control the exit of the loop. An optimized version of the method is shown as Induction-2 in Figure 8.3.1.

### 8.3.2 The Dispatcher is an Associative Recurrence

We now consider a while loop in which the dispatcher is an associative recurrence. Examples of such dispatchers can have the form:

96
Figure 8.3 While loop with associative dispatcher. The loop in (a) is distributed into the two loops in (b), which are then transformed into the parallel prefix computation and the doall shown in (c).

\[ x(i) = a \cdot x(i - k) + b \]  
or  
\[ x(i) = a \cdot x(i - k)^b \]

where \( i = 1, n \) and \( a, b \) and \( k \) are constants. The terms of this relation can be evaluated for \( i = 1, n \) using a parallel prefix computation. This technique has been well documented in the literature ([Lei92]) and gives a logarithmic speedup, i.e., it can be done in \( O(n/p + \log p) \) time, where \( p \) is the number of processors and \( n \) is the number of terms to be computed. Thus, after loop distribution, the first loop can be transformed into a parallel prefix computation, and the second loop can be executed as a doall using the terms of the dispatcher which were computed by the first loop. An example is shown in Figure 8.3.

In the example, no overshooting occurs because the terminator is a threshold function and the dispatcher is monotonic increasing. However, if this had not been the case, then overshooting might have occurred and we would have also needed to find the last valid iteration in order to decide which iterations to undo. Unfortunately, this cannot be done in loop 1 (of the distributed loop) without bringing additional computation from the loop body into loop 1. In this situation, it is probably best to strip-mine the loop, and to find the last valid iteration inside loop 2 (in the same manner as in Figure 8.3.1 for Induction-I). The drawback of this approach is that loop 1 could potentially calculate a large number of superfluous values of the dispatcher.
8.3.3 The Dispatcher is a General Recurrence

This section presents several methods for parallelizing loops with inherently sequential dispatchers. These techniques do not attempt to parallelize the dispatcher itself since, as we have mentioned before, it can be represented by a continuous chain of flow dependences.

*General-1*
ptr tmp = head(list)
doall i = 1,u
  ptr pt
  lock(list)
  pt = tmp
  tmp = next(tmp)
  unlock(list)
  if (pt .eq. null) QUIT
  work(pt)
endoall

*General-2*
doall i = 1, nproc
  integer j
  ptr pt
  pt = head(list)
do j = 1,vpn
  pt = next(pt)
  if (pt .eq. null) goto 2
endo

*General-3*
doall i = 1, u
  integer j, prev
  ptr pt
  prev = 1
  pt = head(list)
loop
  do j = 1, i - prev
  pt = next(pt)
  if (pt .eq. null) QUIT
endo
work(pt)
prev = i
endoall

Figure 8.4 While loop with linked list dispatcher. In the doalls, u is an upper bound on the number of iterations of the while loop, nproc is the number of processors, and vpn is the virtual processor number of the processor executing the iteration. In General-3, the operations before the loop label are executed just once by each processor, and the operations after the loop label are executed for every iteration.

Instead we are trying to speed up the while loop as a whole by taking advantage of the parallelism of the remainder of the while loop body, i.e., we attempt to maximize the overlap between iterations. Thus, if there is not sufficient parallelism available in the loop remainder, then the original while
loop should be executed sequentially. For simplicity, we describe the methods as applied to a while loop that traverses a linked list.

We first notice that when loop distribution is applied the evaluation of the dispatcher is completely sequential, i.e., we cannot perform a parallel prefix computation. In this case, since the parallel execution of the remainder cannot be started before all the terms of the dispatcher have been computed sequentially, it is not clear that the restructuring of the while loop into a sequential dispatcher loop and a parallel remainder will be beneficial. This is especially true if the terminator is RV since the loop distribution scheme would either involve moving portions of the parallel remainder containing the termination condition to the sequential recurrence loop, or entail the sequential computation of unneeded terms of the dispatcher (those beyond the last iteration) which are stored in additional memory. It is possible that strip-mining the loop could improve these costs for RV-type terminators. However, this option would drastically increase the overhead of parallelization since the strips would then be executed as a doalls, separated by synchronization barriers. In fact, it is even possible that a slowdown could occur due to this increased overhead.

We now describe several methods that enable iterations of the loop body to be executed concurrently but do not use loop distribution. One simple method, referred to as General-1, is to serialize the accesses to the next() operation. This technique is equivalent to hardware pipelining which has been well studied in the literature [HP90]. The cost of synchronization and the limited amount of parallelism may make this scheme unattractive. A method, General-2, which avoids explicit serialization, is to compute the whole recurrence in each processor in private storage and assign to processor $i$ the privatized values $k$ of the recurrence such that $k = i \mod nproc$, where $nproc$ is the total number of processors. A third method, General-3, dynamically assigns iterations to the processors like General-1, and also avoids explicit serialization like General-2. In this method, each processor records the last iteration that it processed (prev) and the value of the recurrence at that point (pt). Then, when it is assigned a new iteration $i$, it calculates the values of the recurrence between prev and $i$. Examples of all three methods for the while loop of Figure 8.1(b) are shown in Figure 8.4.

We first contrast the loop distribution approach with the general strategy of embedding the sequential evaluation of the dispatcher inside the parallel execution of the loop as is done in the other methods described above. Notice that the performance of both strategies is likely to be similar for loops in which the terminator is RI, i.e., when overshooting is not possible. However, when the terminator is RV, the loop distribution approach would prove to be inferior due to the reasons mentioned above, i.e., the extra sequential computation performed in loop 1, or when strip-mining
the costs of the associated doall's and synchronizations, none of which are concerns for the other methods.

We now consider the relative advantages/disadvantages of the methods that do not use loop distribution. In addition to the fact that General-1 explicitly serializes accesses to next(), and no such serialization is used in General-2 or General-3, there are some other differences between the methods. First, in General-1 the recurrence is computed (the list is traversed) just once by all processors cooperatively, but in General-2 and General-3 each processor computes the entire recurrence. Second, in General-1 and General-3 the values of the recurrence are dynamically allocated to the processors, but in General-2, processor \( i, 0 \leq i < nproc \), is statically assigned values congruent to \( i \mod nproc \). Another point, related to this last difference, is that the iteration execution pattern of the methods that dynamically assign iterations to processors (General-1 and General-3) may be quite different from that of General-2 that statically assigns iterations to processors. In particular, the span of iterations (i.e., difference between the lowest and smallest iteration numbers) that are executing at any given time might be larger for the static assignment method than for the dynamic assignment method. If the termination condition of the loop is RV, then it is likely that more iterations would need to be undone in the static assignment method than in the dynamic assignment methods.

In the example while loop from Figure 8.1(b), no overshooting occurs because the termination condition is RI. However, if the termination condition had been RV, then overshooting might have occurred and in order to determine which iterations needed to be undone, we would have also needed to find the last valid iteration. This could be done in the same manner as shown in Figure 8.3.1 for Induction-1. With the loop distribution method the performance would be taxed additionally with the serial computation of the overshot values of the dispatcher.

8.4 Undoing Iterations that Overshoot the Termination Condition

Perhaps the easiest method for "undoing" iterations that overshot the termination condition is to checkpoint prior to executing the doall, and to maintain a record of when (i.e., iteration number) a memory location is written during the loop. Note that since all iterations of the while loop have been assumed independent, each memory location will be written during at most one iteration of the loop. Then, after the doall has terminated and the last valid iteration is known, the work of iterations that have overshot can be undone by restoring the values that were overwritten during these iterations. This solution may require as much as three times the actual memory needed by the original while loop: one copy for checkpointing, one for the actual loop data, and one for the time-
stamps. It is possible that this increase in memory requirements might degrade the performance of the parallel execution of the \texttt{while} loop. However, in some situations the memory usage is quite reasonable and, as we will show in Section 8.9, this scheme performs very well.

It might be possible to reduce the cost of checkpointing by identifying and checkpointing a point of minimum state in the program prior to the parallel execution of the \texttt{while} loop. Alternatively checkpointing could be avoided by privatizing all variables in the loop, copying in any needed values, and copying out only those values that are live after the loop and have time-stamps less than or equal to the last valid iteration. Privatized variables need not be backed up because the original version of the variable can serve as the backup since it is not altered during the parallel execution. If the access pattern of any array in the loop is known to be sparse, then the memory requirements could be reduced by using hash tables for the private version of the array. Less memory would be needed in this case since only the elements of the array accessed in the loop would be inserted into the hash table.

A simple way to reduce the memory requirements is to \textit{strip mine} the loop, i.e., execute the first $s$ iterations, then the next $s$ iterations, etc., for some suitable value of $s$. Then, the memory needed to maintain the time-stamps would be bounded by the product of $s$ and the number of write accesses performed in an iteration. However, this method would introduce global synchronization points in the \texttt{doall} and potentially reduce significantly the amount of obtainable parallelism. A better method of controlling memory usage at the application level is discussed in section 8.8.2.

Finally, we mention that time-stamping can be avoided completely if one is willing to execute the parallel version of the \texttt{while} loop twice. First, the loop is run in parallel to determine the number of iterations (using one of the methods of Sections 8.3.1-8.3.3). Then, since the number of iterations is known, the second time the loop can simply be run as a \texttt{doall}.

\section{While Loops with Unknown Cross-Iteration Dependences}

We now consider \texttt{while} loops for which the compiler cannot statically determine the access pattern of a shared array $A$ that is referenced in the loop. The dependences between the statements referencing the shared array may be difficult and/or impossible for the compiler to analyze for a number of reasons: very complex subscript expressions which could only be computed statically through deeply nested forward substitutions and constant propagations across procedure boundaries, nonlinear subscript expressions (a fairly rare case) and, most frequently, subscripted subscripts.
The iterations of such a loop can be executed in parallel, without synchronization, if and only if the desired outcome of the loop does not depend in any way upon the execution ordering of the data accesses from different iterations. In order to determine whether or not the execution order of the data accesses affects the semantics of the loop, the data dependence relations between the statements in the loop body must be analyzed [PW86, KKP+81, Ban88, Wol89, Zim91] (see Chapter 2 for a brief introduction to data dependence issues).

\[
\begin{align*}
\text{do } i &= 1, n \\
\text{if } (f(i) \text{ .eq. true}) &
\begin{cases}
\text{then exit} & \text{ s6: } A[2\ast i-1] = \text{tmp}
\end{cases}
\text{ enddo}
\end{align*}
\]

\[
\begin{align*}
\text{do } i &= 1, n/2 \\
\text{if } (f(i) \text{ .eq. true}) &
\begin{cases}
\end{cases}
\text{ enddo}
\end{align*}
\]

\[
\begin{align*}
\text{do } i &= 2, n \\
\text{if } (f(i) \text{ .eq. true}) &
\begin{cases}
\end{cases}
\text{ enddo}
\end{align*}
\]

(a) (b) (c)

**Figure 8.5** While loops with data dependences: loop in (a) is independent, loop in (b) is independent if privatization is applied, loop in (c) is dependent.

Figure 8.5 shows a few examples of while loops with different data dependence structure. There are no dependences in the loop in Figure 8.5(a) because values that are computed (produced) in some iteration of the loop are used (consumed) during some later iteration. The loop in Figure 8.5(c) has a flow dependence between statement s4 of iteration \(i-1\) and statement s4 of iteration i. The anti-dependences between statement s6 in iteration \(i-1\) and statement s4 in iteration \(i+1\) in the loop shown in Figure 8.5(b) can be removed by privatizing the temporary variable tmp.

Now, consider a while loop for which the cross-iteration dependences cannot be analyzed at compile-time. Instead of executing the while loop sequentially, the compiler could decide to speculatively execute the loop in parallel as a doall (using one of the techniques described in the previous section) and to test at run-time whether there were any cross-iteration dependences in the loop. If the test finds that there were cross-iteration dependences in the loop, then it will be re-executed sequentially. In addition, if it is suspected that some memory-related dependences could be removed by privatization, then the compiler may further elect to privatize those arrays that need it in the speculatively parallelized while loop. In order to speculatively parallelize a while loop as outlined above we need an error (hazard) detection method to test the validity of the speculative parallel execution.
8.5.1 Detecting Errors in the Parallel Execution

There are essentially two types of errors (hazards) that could occur during the speculative parallel execution of the while loop: (i) exceptions and (ii) the presence of cross-iteration dependences in the loop. A simple way to deal with exceptions is to treat them like an invalid parallel execution, i.e., if an exception occurs, abandon the parallel execution, restore the values of any altered program variables, and execute the loop sequentially. In Chapters 2 and 3, we have proposed a run-time technique, called the Privatizing doall test (PD test) [RP94a] and the more powerful LRPD test [RP95], for detecting the presence of cross-iteration dependences in a loop. This test was originally presented as a method to test at run-time whether a do loop was fully parallel, i.e., whether it could be executed as a doall. However, the LRPD test can also be adapted to detect cross-iteration dependences in a while loop since such a loop is essentially a generalization of a do loop, i.e., it is just a do loop with an unknown iteration space.

Because we have already extensively presented the LRPD test in Chapters 2 and 3 we will now discuss how to use the test on while loops. The general strategy is to combine the LRPD test (applied on the remainder loop) with the techniques described in Sections 8.3.1 – 8.3.3 for transforming while loops into doall loops.

If it is known that the parallel execution of the while loop will not overshoot, then the shadow variable accesses of the LRPD test can be inserted directly into the doall loop that is created for the while loop. When overshooting may occur, a simple solution is to initially assume that there are no cross-iteration dependences, and execute the loop twice. First, the loop is run in parallel to determine the number of iterations (using one of the methods of Sections 8.3.1-8.3.3), and once the number of iterations is known the resulting do loop can be speculatively parallelized using the LRPD test as mentioned above.

In order to avoid executing the parallel version of the while loop twice, the LRPD test can be incorporated directly into the while loop methods as follows. Suppose that some shared array A in the while loop will be privatized and tested using the LRPD Test, and assume that it is known that A is not live after the loop. In this case, all writes to the shadow arrays used for the LRPD Test will be time-stamped (just like all other variables), and for each shadow element we will maintain the minimum iteration that marked it. Everything proceeds as before, except that in the post-execution analysis of the LRPD test, those marks in the shadow arrays with minimum time-stamps greater than the last valid iteration will be ignored.

If a privatized shared array under test is live after the loop, then the backup method for the privatized array must be more sophisticated. The reason for this is that it is possible for a private
variable to be written in more than one iteration of a valid parallel loop. In order to handle this problem, we can keep a time-stamped (by iteration number) trail of all write accesses to the privatized array. If the test passes, the live values need to be copied out: the appropriate value would be the value with the latest time-stamp that was not larger than the last valid iteration number, and could be found in the time-stamped trail of write accesses. Methods for reducing the memory requirements are given in Section 8.8.

If the termination condition of the \texttt{while} loop is dependent (data or control) upon a variable with unknown dependences, then special care must be taken. If it turns out that there are no cross-iteration dependences in the loop, then the techniques mentioned above would work as before. However, if there is a dependence between statements in different iterations accessing some variable, and the termination condition is dependent upon that variable, then some difficulties may arise if the loop is executed in parallel. For example, the last valid iteration of the loop might be incorrectly determined, or, even worse, the termination condition might never be met (an infinite loop). In this situation, the best solution is probably to strip-mine the loop, and to run the LRPD test on each strip.

\section{Transforming Arbitrary \texttt{while} Loops for Parallelization}

In the previous sections we made the simplifying assumption that the \texttt{while} loop had only one recurrence (dispatcher) and proposed methods for parallelizing it. In this section we extend our methods to the case when the loop under consideration contains an arbitrary number of statically detectable recurrences.

After constructing the data dependence graph of the loop, we distribute the initial loop into a loop formed by the hierarchically top level recurrences and a second loop containing the remainder of the statements. The recurrence(s) extracted are then parallelized (if possible) with the methods described in Section 8.3. The second loop is then treated in the same way, with the difference that we have already computed its data dependence graph. This means that, for analysis purposes, we now extract the top level recurrence and attempt to parallelize it. Essentially this method amounts to a recursive application of the techniques described in the previous sections. This process stops after all recurrences have been extracted from the original \texttt{while} loop. The remaining loop may take several forms:

\begin{itemize}
  \item A fully parallel loop
  \item A sequential loop whose dependence structure is not of the form of any recurrence detectable by the compiler
\end{itemize}
Figure 8.6 General algorithm for loops with multiple recurrences applied to two different loops. The \( P \) and \( S \) sections refer to the loops obtained after initial distribution and the \( L \) the loops obtained after fusion.

- A loop whose access pattern cannot be analyzed statically

Once this loop distribution is completed we will fuse all the contiguous loops of the same type, i.e., sequential, parallel. The resulting sequence of loops will be combined (fused) according to the following criteria:

- Maximize granularity

- Maximize the code to be executed in parallel

- Balance the overhead of parallelization with the potential speedup, especially in the case when pseudo-parallel code is generated (see Section 8.3.3)

The method proceeds bottom-up (based on the data dependence graph) and analyzes the nature of the loops. If the first loop is sequential, we fuse it with all following contiguous sequential loops. When the first parallelizable loop is found, we generate a distinct, new loop to which all next contiguous parallel loops are fused. If a new sequential loop is encountered, it is fused to the existing block. The decision of whether to generate parallel code for a parallelizable loop (using one of the methods of Section 8.3.3) depends on its potential parallel performance. In particular, if the overhead of parallelization is not offset by the parallel execution, then sequential code should be generated and fused to the immediately preceding sequential block, if any. In particular, if the overhead of parallelization is not offset by the parallel execution, then sequential code should be generated and fused to the immediately preceding sequential block, if any. In the end this algorithm
produces a parallel loop if enough parallelism is available followed by a sequential loop if any. We can exploit the availability of a dependence graph by scheduling the sequential loops in a doacross fashion. Figure 8.6 shows two examples of loops that have been distributed and then fused based on this general algorithm.

We remark that fusing associative recurrences evaluated by parallel prefix computations must be done carefully if there is data flow between the recurrences. Similarly, loops paralleled with the LRPD test should be fused with care – if at all – to loops that they dominate in the data dependence graph since the cost of a failed test will be increased for the resulted loop.

8.7 Predicting Performance

Although it is not strictly necessary for the compiler to perform any cost/performance analysis, the overall usefulness of techniques for parallelizing while loops will be enhanced if their overhead can be avoided when they are unlikely to yield speedups. The main factors that the compiler should consider when deciding whether to parallelize a while loop are: the probability that the iterations of the loop are independent, the speedups that could be obtained using the techniques, and any potential slowdown that might result. In order to perform this analysis and to predict the parallelism of the loop, the compiler should use both static analysis and run-time statistics (collected on previous executions of the loop); in addition, directives about the parallelism of the loop might prove useful.

In this paragraph we will evaluate the ratio between the ideal speedup obtainable by hand parallelization and the real speedup obtained using the techniques presented in this paper. The ratio computed here is independent of the actual number of iterations in the while loop. If an absolute execution time needs to be predicted then an estimation of the loop counts has to be made. Given a while loop L, the ideal speedup, $S_{pid}$, of L is the ratio between its sequential execution time, $T_{seq}$, and its ideal parallel execution time, $T_{ipar}$ (i.e., the time required by an optimal parallel execution of the loop). If there is not enough parallelism available in the loop, i.e., $S_{pid}$ is small, then it should not be parallelized. For convenience, we partition $T_{seq}$ into $T_{rem}$ and $T_{rec}$, where $T_{rec}$ is the time to compute the entire dispatching recurrence, and $T_{rem}$ is the time spent in the remainder of the loop. In general, $T_{ipar} = T_{rem} / p + T_{rec}$, where $p$ is number of processors, i.e., the recurrence is evaluated sequentially and all other work is done in parallel. However, if the dispatcher is an induction or an associative recurrence, then the dispatching recurrence can be evaluated in parallel as well, i.e., $T_{ipar} = (T_{rem} + T_{rec}) / p$, with an additional term of $\log p$ in the case of the associative recurrence. An example in which there is not enough parallelism available in the loop
to justify its parallelization is when the dispatcher is a more complex recurrence and $T_{rem} < T_{rec}$, i.e., the loop essentially consists of evaluating the dispatcher, which must be done sequentially.

When our run-time techniques are applied, the attainable speedup, $S_{pat}$, will be reduced by the overhead of the methods. This overhead can be divided into $T_b$, $T_d$, and $T_a$, representing the overhead incurred before, during or after the parallel execution of the loop, respectively.

$$S_{pid} = \frac{T_{rem} + T_{rec}}{T_{ipar}}$$

$$S_{pat} = \frac{T_{rem} + T_{rec}}{T_{ipar} + T_b + T_d + T_a}$$

The overhead incurred before the parallel execution, $T_b$, represents the cost of any checkpointing needed to maintain the original values so that iterations can be undone, or, if the LRPD test is applied, and fails, so that the loop can be re-executed sequentially. The overhead during the parallel execution, $T_d$, includes the cost of time-stamping so that invalid iterations can be undone, and the accesses to the shadow variables if the LRPD test is applied. The overhead after the parallel execution, $T_a$, includes the time needed to undo any iterations found to be invalid, and the post-execution analysis of the LRPD test, if it is applied.

Assuming that the LRPD test is not applied, let $a$ denote the number of accesses made during the parallel execution of the loop, excluding those inserted by our techniques. Since all operations contributing to the overhead incurred before and after the parallel execution can be executed in parallel, in the worst case we have $T_b \approx T_a = O(\frac{a}{p})$. The number of operations of the overhead incurred during the parallel execution is also proportional to $a$, but the extent to which these operations can be parallelized is dependent upon $S_{pid}$, the maximum parallelism available in the original loop, i.e., $T_d = O(\frac{a}{S_{pid}})$. However, the worst case for the obtained speedup is when $S_{pid} \approx p$, so that $S_{pat} = O(\frac{1}{p} S_{pid})$. If the LRPD test is applied, then in the worst case $S_{pat} = O(\frac{1}{p} S_{pid})$ because the post-execution analysis might add another term of $a$ to $T_a$. Note that 20-25% of the ideal speedup could be an excellent performance – especially when compared to the alternative of sequential execution.

We remark that in many cases the expected speedup of our techniques will be larger than the worst case estimates given above. For example, if it is known that variables are not live after the loop, then often the overhead of time-stamping and restoring can be avoided. A case when all of the overhead can be avoided is when it is known that no overshooting will occur in the parallel execution (e.g., a linked list traversal with a RI termination condition). In this case, we would have $S_{pid} = S_{pat} = T_{work}/p + T_{rec}$. Another case in which our speedup estimates might prove overly conservative is when the iterations are initiated in order. In this case, on the average we would expect to undo $p/2$ iterations – not all of them as in our worst case estimate of $T_a$.
If the LRPD test is used on the loop, and it fails, then a slowdown will be incurred, i.e., the total execution time includes both the failed parallelization attempt and the sequential re-execution of the loop. Since, when using the LRPD test, in the worst case the failed attempt could require time \(O(\frac{2}{p} T_{seq})\), the total execution time could be \(O(T_{seq} + \frac{2}{p} T_{seq})\). Therefore, the slowdown incurred is proportional to \(T_{seq}/p\).

From our analysis above, it is clear that as long as there is enough parallelism available in the loop, a speedup can be obtained by parallelizing it. This is true even if the dependence relations in the loop are not known, and the LRPD test must be applied — the speedup that would be obtained is significant and the potential slowdown is small. However, if it is known a priori with a high degree of confidence that the loop is not parallel, then parallelization should probably not be attempted. There are essentially two cases in which the maximum available parallelism in the loop may not be sufficient to justify parallelization. The first case is when the dispatcher is not an induction and \(T_{work} < T_{rec}\), i.e., the loop essentially consists of evaluating the dispatcher, which must be done sequentially. The second case in which speedups might not be obtained is if there are not enough iterations in the loop. For a do loop, it is sometimes possible to determine at compile-time whether there are enough iterations to justify parallelization. Unfortunately, this is not true for while loops. However, in this case the compiler could predict the number of iterations using branch statistics, where the branch is on the termination condition of the while loop. Although the application is different, this is not a new idea since branch speculation has been used effectively in superscalar compilers [MCH+92, SLH90, TLS90]. Since branch statistics have already been collected for many benchmarks, these collection mechanisms are available.

8.8 Strategies for Applying the Techniques

In the previous section we discussed the speedups and potential slowdowns that can be expected when using our techniques for parallelizing while loops. In this section, we discuss some strategies that might be used to help bias the results in our favor, i.e., to help to insure or improve speedups and reduce the possibility of slowdowns.

Thus far we have not addressed the fact that our techniques might cause an increase in the working set size which could lead to performance degradation. Motivated by this fact, most of the methods discussed in this section are targeted at managing the size of the working set.
8.8.1 Statistics Enhanced Strip-Mining

The additional memory required by our techniques is for the checkpointing prior to the parallel execution, and the time-stamps made during the parallel execution for the write accesses so that work done by iterations later found to be invalid can be undone.

Perhaps the simplest way to reduce the memory requirements is the strip-mine the loop, as discussed in Section 8.4. In this case, time-stamps would only need to be maintained for values written during the current strip. Therefore, only $O(sa)$ memory would be needed for these values, where $s$ is the size of the strip and $a$ is the number of writes per iteration. However, strip-mining introduces global synchronization points and might potentially reduce significantly the amount of obtainable parallelism.

Suppose that, as discussed in Section 8.7, the compiler has supplied an estimate $n_i$ of the number of iterations in the loop. If this is a good estimate, then the time-stamps for values written in iterations smaller than $n_i$ are not likely to be needed since these iterations will likely be valid. Therefore, in this case, a good strategy might be to only time-stamp the values written in iterations larger than $n_i'$, for some value $n_i'$ close to, but less than, $n_i$, e.g., $n_i' = .9n_i$. The value $n_i'$ could be chosen based on the degree of confidence placed on the compiler's estimate of $n_i$, e.g., if the confidence in $n_i$ is about $x\%$, then $n_i'$ is selected to be about $x\%$ of $n_i$.

8.8.2 Resource Controlled Self-Scheduling

A way in which the memory requirements could be reduced without introducing rigid synchronization points is to maintain a sliding window of some predetermined size $w$: at any given time, the difference between the minimum iteration $l$ that has not been completely executed and the maximum iteration $h$ that has been, or is currently being, executed is at most $w$, i.e., iterations $1$ through $l - 1$ have been completely executed, and $h - l \leq w$. Similar to the strip mining solution, this method would bound the memory needed to maintain the time-stamps by the product of window size $w$ and the number of write accesses performed in an iteration.

Now, suppose that the window size is dynamically determined at the application level based on the current memory usage: the window size is increased if more memory can be used without degrading performance, and is decreased if less memory should be used to improve performance. The window size can be dynamically adjusted by the program itself, since the program could easily monitor how much memory is used by its data structures. The working set size of the application can be user pre-set, or dynamically adjusted according to system usage information available through system calls. Note that we are suggesting that the application monitor its own memory usage and
dynamically adjust its actions accordingly, which is different from operating system monitors that watch such things as network traffic, i/o requests, or paging activity.

### 8.8.3 The 1 Processor/(p - 1) Processor Solution

As a final remark, we note that the method described in Section 4.5.4.2 can be used also here to minimize the risks of parallelizing a while loop: one processor executes the loop sequentially, and the rest of the processors execute the loop in parallel. Of course, the sequential and the parallel executions would need separate copies of the output data for the loop. As long as the cost of creating these copies is not too great, this technique should maximize the potential gains attainable from parallel execution, while, at the same time, minimizing the costs.

### 8.9 Experimental Results

In this section we present experimental results obtained on a modestly parallel machine with 8 processors (Alliant FX/80 [All86]) using a Fortran implementation of our methods. It should be pointed out that our results scale with the number of processors and the data size and that they should be extrapolated for MPPs, the actual target of our methods.

We considered five while loops that could not be parallelized by any compiler available to us; two loops are from the PERFECT Benchmarks [BCK+89], two loops are from MA28, a sparse non-symmetric linear solver [Duf77], and one loop is extracted from MCSPARSE, a parallel version of a non-symmetric sparse linear systems solver [GMW89, GMW91]. Our results are summarized in Table 8.9. For each method applied to a loop, we give the speedup that was obtained, and, mention whether backups and time-stamping were necessary. Whenever necessary, we performed a simple preventive backup of the variables potentially written in the loop. In some cases, the cost of saving/restoring might be significantly reduced by using another strategy. In addition to the summary of results given in Table 8.9, we show in Figures 8.7 through 8.15 the speedup measured for each loop as a function of the number of processors used.

Overall, our results show that significant speedups can be obtained by parallelizing while loops using our methods. We now make a few remarks about individual loops for which Table 8.9 does not give complete information.

Loop 40 in subroutine LOAD from SPICE loads the device models for capacitors. Since our interest is in measuring the performance of our linked list traversal techniques, the run-time overhead associated with run-time dependence testing has not been included in the reported results.

---

2 All benchmarks are from the PERFECT Benchmark Suite, with the exception of MCSPARSE.
<table>
<thead>
<tr>
<th>Benchmark\m Subroutine Loop</th>
<th>Experimental Results</th>
<th>Description of Loop</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPICE LOAD Loop 40</td>
<td>General-1 (locks)</td>
<td>traverses linked list terminated by a NULL pointer loop counter: recurrence</td>
</tr>
<tr>
<td></td>
<td>General-3 (no locks)</td>
<td>termination condition: RI no backups or time-stamps</td>
</tr>
<tr>
<td>TRACK FPTRAK Loop 300</td>
<td>Induction-1</td>
<td>accesses array indexed by run-time computed subscript array loop counter: induction termination condition: RV backups and time-stamps</td>
</tr>
<tr>
<td>MCSPARSE DFACT Loop 500</td>
<td>Induction-1</td>
<td>processes an array loop counter: induction termination condition: RV no backups and no time-stamps</td>
</tr>
<tr>
<td>MA28 MA30AD Loop 270</td>
<td>Induction-1 and General-3 (no locks)</td>
<td>processes an array loop counter: induction termination condition: RV backups and time-stamps</td>
</tr>
<tr>
<td>MA28 MA30AD Loop 320</td>
<td>Induction-1 and General-3 (no locks)</td>
<td>processes an array loop counter: induction termination condition: RV backups and time-stamps</td>
</tr>
</tbody>
</table>

Table 8.2 Summary of experimental results for while loop parallelization

Even though the body in Loop 40 does little work, we obtained a very good speedup (Figure 8.7). Note that although each processor traversed the entire linked list, the General-3 method significantly outperformed the General-1 method, in which the processors cooperatively traversed the list (by placing the next() operation in a critical section). Since the structure of Loop 40 is identical to those for the evaluation of transistor models (subroutines BJT and MOSFET), the same parallelization techniques can also be used on these loops. We remark that approximately 40% of the sequential execution time of SPICE is spent in subroutine LOAD, which calls subroutines BJT and MOSFET.

Loop 300 in subroutine FPTRAK from TRACK is a do loop with a conditional exit which is taken if an error condition is detected. The speedup obtained is shown in Figure 8.8. For this loop we also show the ideal speedup, which was obtained from a hand parallelized version of the loop.

Loops 270 and 320 in subroutine MA30AD from MA28 cooperatively search for a pivot. Since MA28 is a sequential program, any parallelization must guarantee sequential consistency. In order to accomplish this we time-stamped the pivots found during the parallel execution. Then, after loop termination, we found the pivot with minimum cost by performing a time-stamp ordered reduction operation (minimum) on the (privatized) pivots selected by each processor. For each input set,
the speedups for both Loop 270 and 320 are shown on the same graph (see Figures 8.13 through 8.15). We remark that the speedups shown for the loops from MA28 are not as big as for the other programs. This is largely due to the fact that there was less available parallelism in these loops.

8.9.1 While-Doany

MCSPARSE is, as mentioned before, a sparse solver that has been manually programmed as a parallel code. Loop 500 in subroutine DFACT from MCSPARSE searches for a pivot in a non-deterministic manner. In other words the program is designed to be insensitive to the order in which the columns and rows of the matrix are searched for the pivot. Originally, only the row search was parallelized by applying a technique equivalent to a doany construct [Wol92], leaving the traversal of columns in a sequential while loop. We fused the two loops, effectively implementing a new While Doany parallel construct. Through this technique we were able to parallelize the pivot search across the whole matrix. Since the order of the searching iterations is not important, we did not need to perform backups or maintain time-stamps for back-tracking, even though the termination condition is RV and we do overshoot. We report speedups for four different input data sets from large sized Harwell-Boeing matrices (see Figures 8.9 through 8.12). Note that the available parallelism, and therefore our obtained speedup, is strongly dependent on the data input.

8.10 Related Work

We can find in the literature several efforts in improving the performance of the while loop execution. In [TLS90] the authors have proposed some methods for achieving vector-like performance on multiple issue pipelined machines. They do not try to address the problem for large multiprocessors.

Some techniques for solving certain types of recurrences in parallel were proposed by Harrison in [Har89] for Lisp-like languages. His main goal was to parallelize list operations (e.g., traversing a linked lists). Generally, his methods assume that the terminator is RI and it is known that there are no cross-iteration dependences in the loop. In the context of his proposed framework ([Har86]), lists consist of linked chunks of contiguously allocated memory locations, and each chunk has a header that stores the number of memory locations in that chunk. In this way the evaluation of the dispatcher (i.e., the traversal of the list) can be optimized by using a sequential prefix computation (on the chunks) to assign portions of the recurrence (chunks) to processors for parallel evaluation. We note that this optimization requires the dynamic memory allocation scheme proposed by the author (in which list elements are allocated contiguously). Therefore, for languages such as
FORTRAN which rely mainly on static memory allocation (i.e., each list element is contained in a separate chunk), this method could not be used to parallelize the evaluation of the dispatcher, i.e., it would degenerate to the naive loop distribution method mentioned for general recurrences in Section 8.3.3. In fact, the author mentions that if the chunk sizes become too small, then the result might be an “inefficient restructured version of the loop that contains too little parallelism to recover the expense [invested]” [Har86]. We note that when the entire list resides in a single chunk (i.e., an array), then this method is equivalent to the method we describe in Section 8.3.2 for associative recurrences, i.e., loop distribution together with a parallel prefix computation to evaluate the dispatcher in parallel.

The only previous work of which we are aware (except some early work by [Wol89]) for parallelizing while loops in languages such as FORTRAN for multiprocessors is due to Wu and Lewis [WL90]. One method they propose is to pipeline the loop by executing it in doacross fashion, and to enforce any cross-iteration data dependences with explicit synchronization operations. When the terminator is RI and it is known that there are no cross-iteration data dependences in the loop, they suggest using the naive form of loop distribution mentioned in Section 8.3.3 (also implicit in [Har86]), i.e., first a sequential while loop evaluates the dispatcher and stores its values in an array, and then the loop iterations are performed in parallel using this array.

For the case of RV termination conditions no methods have been proposed in the past. Also, the problem of testing for cross-iteration data dependences has not been addressed before.

8.11 Conclusion

In this chapter we have shown that lack of knowledge about the iteration space of a loop does not preclude parallelization. We have demonstrated this by giving techniques for concurrently executing while loops and do loops with conditional exits. Our methods can even be used to obtain significant speedups for loops that involve linked list traversals without using global synchronization or explicitly sequential code – something that has not been done before. This is an important problem since linked list traversals arise frequently in loops with irregular access patterns, such as sparse matrix computations. In many cases, these methods have no associated overhead or side effects. Our techniques can be applied even when the dependence relations between the iterations of the loop cannot be analyzed at compile-time. In this case, an efficient run-time test for cross-iteration dependences is inserted into the parallel version of the loop, and the outcome of the test determines whether the parallel execution was valid, or if the loop must be re-executed sequentially.
We feel our framework for while loop parallelization represents a step toward providing while loop counterparts for the existing constructs for parallel execution of do loops: While-Doall, While-Doacross, and While-Doany. Based on our experience, these new constructs would be useful extensions to present parallel languages.

Our experimental results show that our techniques yield significant speedups for real loops – even on a modestly parallel machine like the Alliant FX/80. The performance gain (speedup) from our techniques ranges from a minimum of 20 – 25% of the ideal speedup to nearly 100% of the ideal speedup. If the target architecture is an MPP with hundreds or, in the future thousands, of processors, then even the minimum expected speedup could easily reach into the hundreds. We have also shown that the potential payoffs remain large when the cross-iteration dependencies are analyzed at run-time. Therefore, our conclusion is that as long as there is enough parallelism available in the while loop, our techniques should be applied.

To bias the results even more in our favor, we would like to avoid parallelizing loops that do not have enough available parallelism. For this reason it would be useful to estimate the number of iterations in the loop using information such as branch statistics – data which can easily be obtained for any program. Also, in order to decrease the probability of attempting to parallelize a loop that is in fact sequential, our methods should make use of run-time collected information about the parallel/not parallel nature of the loop. In all cases, specialized hardware features could greatly reduce the overhead introduced by the methods.
Figure 8.7 Speedup for loop in SPICE

Figure 8.8 Speedup for loop in TRACK

Figure 8.9 Speedup for loop in MCSPARSE

Figure 8.10 Speedup for loop in MCSPARSE

Figure 8.11 Speedup for loop in MCSPARSE

Figure 8.12 Speedup for loop in MCSPARSE
Figure 8.13 Speedup for loop in MA28

Figure 8.14 Speedup for loop in MA28

Figure 8.15 Speedup for loop in MA28
Chapter 9

Extensions, Significance of this Work, Future Research

9.1 Further Applicability of Run-Time Methods

Thus far we have discussed how run-time techniques can be used in the detection of parallelism in the case of input or run-time dependent access patterns in the framework of automatic parallelization. However these methods can be also used in the context of parallel programming for a variety of purposes that will be sketched in the next sections.

9.1.1 Verifying Parallelized Loops

As we have mentioned in Chapter 7 there has been a significant amount work done in the area of debugging and testing parallel programs. Some of this work answers the question whether an access anomaly has occurred rather than if an anomaly can occur for a given input data. The run-time data dependence tests presented in this thesis can extract the parallelism structure of a loop for any given access pattern - whether it is statically or dynamically defined. The answers they provide are of a deterministic nature: they do not show whether a memory access was unsafe but whether it could be could be unsafe for any given input set.

It follows that it would be quite beneficial to use the run-time dependence test as debugging and program verification tools. In some cases we think it would be more important to check if there is a potential problem rather than if a problem actually occurred during one experiment. We believe that a compiler assisted debugging tool where the user would selectively test proper parallelization by calling the run-time test could prove to be useful. Additionally, this approach would neither incur the large storage overhead nor the execution time expansion typical of current debuggers and tracers.

9.1.2 Run-Time Check as a Parallel Language Extension

In the previous chapters we have discussed how to detect parallelism and how to transform do and while loops into one or more parallel doall loops with the purpose of automatically parallelizing existing applications. Writing explicitly parallel programs requires the user to have a priori knowledge of the data dependence structure of the loops (assuming it uses loop level parallelism). However, in the case of dynamic codes, where the data distribution is input dependent or changes during program execution it is not possible to guarantee statically when loops are parallel.
and therefore conservative decisions are made. In order to overcome this problem we propose to extend any existing language with constructs that will automatically test and exploit parallelism. For example we can devise a construct named check-doall that can be used in cases when a loop is not certain to be parallel. The implementation could actually use one of our run-time techniques in a user transparent way. Furthermore the potentially dependent variable can be specified thus reducing the necessary amount of compiler and run-time analysis.

In Chapter 8 we have presented the transformation necessary to execute while loops concurrently. As mentioned in Section 8.9.1 we can extend the concept of doall in the case of an unknown iteration space. While loops known to be independent can be coded as while doany if they do not need to be sequentially consistent (i.e., always execute the same number iterations for a given input set) and the user is willing to accept a certain amount of non-determinacy. If a deterministic outcome is required then it can be encoded as a while doall possibly paying a performance penalty due to additional bookkeeping and backtracking. Finally, if the while loop is only potentially parallel then it can be specified with check-while doall and check-while doany constructs which will incorporate one of the previously described run-time tests.

We believe that such language extensions, if implemented efficiently, could be of great use in extracting and exploiting statically un-specifiable parallelism in a safe manner.

9.2 The Contribution of this Work

In this thesis we have presented several original techniques that together sketch how automatic compilation can go beyond statically analyzable, well-behaved codes. The methods described are true to the principles enunciated in the introduction: they are fundamentally efficient, scalable and general, i.e. their characteristics are not based on heuristics with a wide performance distribution across their input domain but are algorithms that can be analytically proven to produce speedups given the necessary resources and available parallelism. We have introduced the idea of testing only for full parallelism in the presence of run-time transformations rather than computing an execution schedule. We have introduced the very aggressive strategy of speculative parallel execution that can be scaled to any parallel system (from micros to multiprocessors). Additionally we have presented a framework for the parallelization of loops which contain recurrences and have an unknown iteration space and we have sketched a global criterion for fusing and distributing loops within a program based on their concurrency characteristics.

We believe that the significance of the methods presented here will only increase with the advent of massively parallel processors (MPPs) for which the penalty of not parallelizing a loop could be
a massive performance degradation and where the costs (overhead) of our techniques will become a very small fraction of the sequential execution time.

We believe that within the domain of automatic parallelization the true importance of this work is in breaking the barrier at which automatic parallelization had stopped: regular, well-behaved programs. The use of aggressive, dynamic techniques can extract most of the available parallelism from even the most complex programs, making parallel computing attractive.

9.3 More Work Ahead

Although our experimental results indicate that our approach is promising more research needs to be done before we can claim victory. In order to build a comprehensive framework for run-time detection and exploitation of parallelism we need solve several more problems:

New Run-Time Algorithms

- Our library of run-time techniques needs a method that can compute an execution schedule for any partially parallel loop, not only from those from which a proper inspector can be extracted.

- The 'pure run-time' algorithms presented in this thesis need to be complemented with run-time extensions of static compilation techniques. For example symbolic analysis has been proven to be crucial to automatic parallelization. Although the available static techniques are limited, they could easily be extended at execution time (providing multiple code versions). The overhead of such 'extensions' will most likely be minimal.

Integration in a Compiler and Optimization

- Efforts are needed to perfect the seamless integration of run-time techniques into the compiler. For example, deciding the best hierarchical level at which to apply the LRPD test and determining the amount of parallelism expected from an un-analyzable loop (whether it is fully or partially parallel).

- Compiler (static) heuristics for finding a 'good guess' of the data dependence structure of a loop are necessary if we decide to speculate on it. (Some of these ideas have been mentioned in Section 4.5)

- The transfer of all statically available data about the program to the run-time system can greatly reduce overheads. For example, if we know exactly which section of an array is
compile-time undecidable, we can transfer this information to the run-time system and test only a minimal number of memory locations rather than a whole array.

- Predicting the execution time of the code generated by different alternative transformations is essential for making good decisions. For that purpose a high level architectural model should be incorporated into the compiler. The model need not be highly accurate but just sufficient to differentiate between the work and execution time of different code alternatives that a compiler could generate: decisions such as which loop to parallelize and how to schedule are heavily influenced by the amount of work and its granularity and can be optimized using such a model.

- Distributing the different levels of parallelism among the architectural levels that can exploit it is an important decision that can greatly affect the obtained speedups. Through compiler (static and run-time) analysis we can find several nested levels of parallelism (e.g., nested parallel loops) and we have to find a policy based on which we can decide at which physical level it can be exploited – from instruction level on a superscalar node [RDN93] to groups of nodes (clusters) and reaching the entire computer system.

Optimization through Feedback

- The use of statistics for feedback during current execution or future invocations of the program has to be studied. Speculations about parallelism can be biased favorably if previous experience could be used. However the statistical significance of the collected data has yet to be proven. A static/dynamic approach of data selection and collection can be used to break up the data dependence structure of a program into a static and dynamic part. This would allow us to draw the correct conclusion about the usefulness of run-time collected data.

- A more modest goal is to find a relationship between data set size and parallelism structure. However, for complex applications data set size is sometimes difficult to define.

Architectural Support

- It would be desirable to adapt our software-only approach to increase the parallelism in current and future microprocessors, while taking advantage of their built-in speculative load mechanisms.

- Architectural support for accelerating certain typical operations related to marking, checkpointing, and data cross-processor merging would be very useful.

120
An ever increasing domain of application for run-time parallelization is its application to programs written in more modern languages like C and C++. It is quite reasonable to assume that because of their extensive use of indirection and dynamic memory allocation, static analysis of such codes is very difficult and dynamic analysis will become indispensable.

9.4 A Final Word

We believe the philosophy of making decisions on-the-fly is a general strategy for optimizing program performance that has potential uses and benefits of a much larger scope than that of detecting and exploiting parallelism. As the number of dynamic applications is increasing, the information that can be obtained statically from the body of all programs is decreasing. At the same time the characteristics of an initial data distribution can greatly affect the behavior of a program even if we leave the data dependence issue aside. We view the run-time methods presented here as a stepping stone into the creation of an adaptive computing environment. Data distribution, scheduling and load balancing, and task granularity affect execution time greatly and are quite often impossible to analyze at compile time. This thesis hopes to have convinced its readers that making decisions dynamically adds extra work but reduces overall execution time – Provided it is done properly.
References


[EHLP91] Rudolf Eigenmann, Jay Hoeflinger, Zhiyuan Li, and David Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs. Lecture Notes in Com-


[RP95] Lawrence Rauchwerger and David A. Padua. The LRPD Test: Speculative Run-Time Parallelization of Loops with Privatization and Reduction Parallelization. In *Proceed-

126


Vita

After graduating from the German School in Bucharest and spending a year in reserve officer school he was admitted to the Department of Electronics and Telecommunications of the Polytechnic Institute Bucharest. In June 1980 he received the degree of Engineer and joined the “Felix” Computer Company in Bucharest as an R&D engineer. Less than two years later he defected from communist România, spent seven months in Italy and immigrated to the United States. From early 1983 until July 1984 Lawrence worked as an electrical engineer for Beckmann Instruments Inc., Scientific Instruments Division, in Irvine, CA. He then moved to Varian Associates Inc, Thin Film Technology Division, in Palo Alto, CA and became a member of the Research and Development Staff. In 1985 he began graduate studies at Stanford University where he received a M.S. in Electrical Engineering with a specialization in Integrated Circuit Manufacturing Science. In the fall of 1988, he began his Ph.D. studies in Computer Science at the University of Illinois at Urbana-Champaign. Until 1994 he was a research assistant at the Center for Supercomputing Research and Development and for a brief time a teaching assistant in the Department of Computer Science. He started his research in computer architecture under the guidance of Dr. P. Michael Farmwald who left the University after 2 years. He then continued with Prof. Daniel Reed in the performance evaluation area. In May 1989 he married Nancy M. Amato in Bellevue, Washington. In the summer of 1992 he worked with Dr. Ravi Nair and Dr. Pratheep Dubey at the IBM, T. J. Watson Research Center. In the spring of 1993 he joined Prof. David Padua’s compiler group. Since July 1994 he has held the Intel Foundation and the NASA HPCC Graduate Fellowships. He received the Ph.D. in Computer Science in 1995 from the University of Illinois at Urbana-Champaign. His doctoral research was directed by Professor David Padua.