DYNAMIC PROGRAM ANALYSIS ALGORITHMS TO ASSIST PARALLELIZATION

A Dissertation
Presented to
The Academic Faculty

by

Minjang Kim

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in
Computer Science

School of Computer Science
Georgia Institute of Technology
December 2012

Copyright © 2012 by Minjang Kim

DYNAMIC PROGRAM ANALYSIS ALGORITHMS TO ASSIST PARALLELIZATION

Approved by:

Dr. Hyesoon Kim, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Chi-Keung Luk
Technology Pathfinding and Innovations
Intel Corporation

Dr. Hsien-Hsin S. Lee
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Santosh Pande
School of Computer Science
Georgia Institute of Technology

Dr. Richard Vuduc
School of Computational Science and Engineering
Georgia Institute of Technology

Date Approved: August 16th, 2012

ACKNOWLEDGEMENTS

My Ph.D. would not have been successfully finished without support and encouragement from my wife, Kyung Im. Besides this dissertation, our two kids, Dowan and Jiu, are our big accomplishment and great pleasure during the graduate study. My father, my mother, father-in-law, and late mother-in-law have been always support for me and my family.

This work literally cannot be completed without guidance of my advisor, Prof. Hyesoon Kim. From the very first project in Prospector to the last moment of this thesis writing, she always have advised to the right direction and complement what I missed. I have learned so many valuable lessons from her, not only how to find good research topics and solve them analytically, but how to be a good researcher and mentor.

I am also very fortunate to work with Dr. Chi-Keung Luk, for being a mentor during the internship and a co-advisor to shape my thesis, besides physically participating for my defense talk. He guided me many technical challenges to improve my algorithms.

I am grateful to all the dissertation members: Prof. Hsien-Hsin Lee, Prof. Richard Vuduc, and Prof. Santosh Pande. I particularly thank to Prof. Pande for providing me a view from a compiler perspective and sharing time in Korea. I thank to Prof. Milos Prvulovic for guiding my early graduate research. I am also thankful to Bevin Brett and John Pieper during my internship at Nashua.

I thank to all our lab members: Sunpyo for having the same challenges to finish the degree and find a job; Jaekyu and Nagesh for setting up our lab machines in early days and helping many Linux-related problems; Hyojong and Chayoung for

helping parallelization experiments for the post-analyzer; Pranith for the parallel prophet work and struggles; and Puyan for the challenging and interesting LLVM-Prospector work and enjoying many talks. I also thank to my friends: Changhee and Sangmin for being great colleagues to discuss the details of my and their work; Myungcheol for sharing many tricky situations in Ph.D. life; Min for settling down together in the US; and Donghwan, whose dissertation preface can be shared with mine, for big help during job searching.

I thank to many families who stayed together in the tenth and home apartments for years, especially for Chung Hyuk's, Hyojun's, Hyungie's Jin-Kook's, Kihwan's, Min's, Minsung's, Seung-Joon's, Taesu's, and Tonghoon's families.

TABLE OF CONTENTS

ACKNOWLEDGEMENTS iii

LIST OF TABLES xi

LIST OF FIGURES xii

SUMMARY xvi

I INTRODUCTION 1

1.1 The Problem: Parallelizing Serial Code 1

1.2 The Solution and Contributions: Prospector 4

1.3 Thesis Statement 7

1.4 Organization of This Proposal Document 7

II MOTIVATIONS AND OVERVIEW OF PROSPECTOR 8

2.1 Motivations for an Efficient Loop Profiler 8

2.2 Motivations for Dynamic Data-Dependence Analysis 11

2.2.1 Case Studies: Automatic Parallelization in C/C++ Compilers 14

2.3 Motivations for An Efficient Data-Dependence Profiler 20

2.4 Motivations for Correct Data-Dependence Profiling 23

2.4.1 Experimentation Results of Simple Sampling Techniques . . 26

2.5 Motivations for Dynamic Parallel Speedup Prediction 29

2.6 Motivations for A New Speedup Prediction Algorithm 33

2.7 Background on Scheduling Policies of OpenMP and Cilk Plus 36

2.7.1 Scheduling Policies in OpenMP 37

2.7.2 Recursive and Nested Parallelism in OpenMP and Cilk Plus 37

2.8 Motivations for Post-Analyzer of Dependence Profiler 40

2.9 Overview of Prospector 42

III RELATED WORK 44

3.1 Related Research on Loop Profiling 44

3.2 Related Research on Data-Dependence Profiling 45
3.2.1 Data-Dependence Profiling for Speculative Parallelization . 47
3.2.2 Reducing Time Overhead of Dynamic Analysis 47
3.2.3 Limitations of Previous Compression Techniques 48
3.3 Related Research on Dynamic Speedup Prediction 48
3.4 Related Research on Post Analysis of Dependence Profiling 50
3.4.1 Parallelism Discovery Using Dynamic Analyses 50
3.4.2 Code Transformation to Avoid Dependences 51
3.4.3 Bug-Fixing Algorithms in Concurrent Programming 52
3.4.4 Parallelism Visualization 53
3.5 Tools for Assisting Parallelization 54
3.5.1 Intel Parallel Advisor 55
3.5.2 Vector Fabrics' Pareon 56
3.5.3 Rogue Wave's ThreadSpotter 58
IV AN EFFICIENT LOOP PROFILER 59
4.1 Introduction 59
4.2 An Efficient Loop-Profiling Algorithm 60
4.2.1 Reconstructing CFGs and Loop Structures from Binary . . . 61
4.2.2 Instrumenting Loop-Behavior Instructions 62
4.3 Challenges: Case Studies and Solutions 64
4.4 Implementation 67
4.5 Experimental Results 68
4.6 Summary of This Chapter 69
V AN EFFICIENT DYNAMIC DATA-DEPENDENCE PROFILER 71
5.1 Introduction 71
5.2 The Baseline Algorithm: The Pairwise Method 72
5.2.1 Checking Data Dependences in a Loop Nest 73
5.2.2 Handling Loop-independent Dependences 76

5.2.3 Dependences in Functions and Handling Function Calls . . . 78
5.2.4 Computing Data-Dependence Distance 79
5.2.5 Summary of the Pairwise Method 80
5.2.6 Optimizing the Merge Operation: PC-set Optimization . . . 82
5.2.7 Problems of the Pairwise Method 84
5.3 A Memory-Efficient Algorithm in SD^3 85
5.3.1 Overview of the Algorithm 85
5.3.2 Dynamic Detection of Strides 86
5.3.3 Stride-Based Dependence Checking Algorithm 88
5.3.4 Overview of the Memory-Efficient SD^3 Algorithm 93
5.3.5 Optimizing Stride-Based Dependence Checking 94
5.3.6 Merging Stride Tables for Loop Nests 98
5.3.7 Handling Killed Addresses in Strides 99
5.3.8 Lossy Compression in Strides 100
5.4 Reducing Time Overhead by Parallelization 101
5.4.1 Overview of the Algorithm 101
5.4.2 A Hybrid Parallelization Model of SD^3 101
5.4.3 Event Distribution for Parallel Processing 103
5.4.4 Strides in Parallelized SD^3 105
5.4.5 Details of the Data-Parallel Model 106
5.5 Implementation 108
5.5.1 Basic Architecture 108
5.5.2 Implementation of Analyzer 110
5.5.3 Implementation of Tracers 111
5.6 Experimentation Results 113
5.6.1 Experimentation Methodology 113
5.6.2 Memory Overhead of SD^3 115
5.6.3 Time Overhead of SD^3 117

5.6.4 Input Sensitivity of Data-Dependence Profiling 119
5.6.5 Opportunities for Stride Compression 121
5.7 Summary of This Chapter 122
VI AN EFFECTIVE SPEEDUP PREDICTOR FROM SERIAL CODE 123
6.1 Introduction 123
6.2 The Front-end of Parallel Prophet: Annotation and Profiling 125
6.2.1 Annotating Serial Code 125
6.2.2 Interval Profiling to Build a Program Tree 126
6.2.3 Memory Profiling: Burden Factors 128
6.2.4 Summary of the Profiling 130
6.3 The Backend of Parallel Prophet: The Emulators 130
6.3.1 Fast-Forwarding Emulation Algorithm 131
6.3.2 Challenges and Limitations in Fast-Forwarding Method . . . 137
6.3.3 Program-Synthesis-Based Emulation Algorithm: Basic Idea . 138
6.3.4 Challenges in the Synthesizer 140
6.3.5 Summary of the Emulations 142
6.4 Lightweight Memory Performance Model 143
6.4.1 Assumptions 144
6.4.2 The Performance Model 145
6.4.3 The Definition of the Burden Factor 146
6.4.4 Memory Access Overhead, w_t Prediction 147
6.4.5 Details of the Microbenchmark for w_t Prediction 149
6.4.6 Summary of the Memory Performance Model 154
6.5 Implementation 155
6.5.1 Implementation of Annotation System 155
6.5.2 Implementation of Interval Profiling 156
6.5.3 Compression of the Program Tree 158
6.6 Experimentation Results 159

6.6.1 Experimentation Methodologies 160
6.6.2 Validation of the Prediction Model 160
6.6.3 Prediction Results with Memory Performance Model 165
6.6.4 A Simple Verification for the Memory Performance Model . 167
6.6.5 Detailed Cache and Memory Behaviors of the Benchmarks . 168
6.6.6 The overhead of Parallel Prophet 171
6.6.7 Limitations of Parallel Prophet 172
6.7 Summary of This Chapter 173
VII ALGORITHMS TO EXTRACT PARALLELISM AND TRANSFORMATION ADVICE 174
7.1 Assumptions for the Post-Analyzer 174
7.2 Overview of the Post-Analyzer 175
7.2.1 False Dependences and Privatization 176
7.2.2 Reductions and Mutexes 177
7.2.3 Supporting condition variables and barriers 179
7.2.4 Pipelining 180
7.2.5 Ordering of Transformations 180
7.3 Case Studies of Prospector 182
7.3.1 179.art in SPEC CPU2000 182
7.3.2 Susan in MiBench 185
7.3.3 Dijkstra in MiBench 189
7.3.4 256.bzip2 in SPEC CPU2000 193
VIII CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS 198
8.1 Conclusions 198
8.2 Future Research Directions 200
8.2.1 Future work for SD^3 200
8.2.2 Future work for Parallel Prophet 201
8.2.3 Future work for the Post-Analyzer 202

REFERENCES **204**

LIST OF TABLES

1 OmpSCR Automatic Parallelization Results 16
2 Inaccuracies of simple sampling policies 27
3 Scheduling policies in OpenMP parallel-for construct 37
4 Comparisons of recent dynamic tools to predict parallel speedups . . 49
5 Robustness of our binary-level loop profiler 70
6 Memory traces and discovered dependences of Figure 29. 73
7 Annotations in Parallel Prophet. 126
8 Comparison of the emulation algorithms 143
9 Expected speedup classifications based on memory behavior 144
10 Parameters in the performance model 145
11 Parameters in the burden factor 147
12 System configuration 160
13 CPU affinity configuration for the experimentations 161
14 Classification of dependences and solutions for parallelization 176
15 Parallelized benchmarks with Prospector's post-analyzer 182
16 Post-analyzed result of the loop, `scan_recognize:5`, in 179.art 183

LIST OF FIGURES

1 A simple loop profiling in Intel Parallel Advisor 9

2 A hot loop of 436.cactusADM: `ScheduleTraverse.c` 10

3 LUReduction and FFT6 benchmarks in OmpSCR 17

4 Mandelbrot in OmpSCR 19

5 FFT in OmpSCR 20

6 Overhead of the current data-dependence profilers 21

7 The partitioning algorithm in Quicksort 28

8 From ScheduleTraverse.c in 436.cactusADM 28

9 Memory overhead of Jacobi in OmpSCR 29

10 Comparison of analytical models and a dynamic approach 32

11 Easy-to-predict parallel patterns by dynamic analyses 34

12 Hard-to-predict parallel programming patterns in OmpSCR 35

13 FT in NPB. Input set is 'B' of 850 MB memory footprint 36

14 OpenMP parallel for with schedule clause 37

15 Parallelized QuickSort by Cilk Plus and OpenMP 38

16 Speedups of various parallelization versions of QuickSort 40

17 Raw result of SD^3: The MPEG2 decoder in MediaBench 41

18 Overview of Prospector 42

19 Overview of Intel Parallel Advisor's approach 55

20 Overview of Intel Parallel Advisor's approach 57

21 A screen capture of Rogue Wave's ThreadSpotter 58

22 An example of binary-level loop profiling 59

23 x86-64 code-generation styles of three major compilers for a loop . . 63

24 A complex case of binary-level loop instrumentation 64

25 A complex case due to indirect-branch based switch-case 65

26 A combination of switch-case and goto can be a loop 65

27 Overview of the loop profiler of Prospector 68
28 Time overhead comparison of Prospector and LoopProf 69
29 A simple example of data-dependence profiling 73
30 Snapshots of the pending and history tables for Figure 29 74
31 Loop-independent flow dependences on `pass_flag` in 179.art 77
32 Three kinds of loop-independent dependences in a loop nest 78
33 Finding dependences among functions and loops. 79
34 Data structures in the Pairwise method 80
35 Hit ratio of the union-computation for SPEC 2006 83
36 PC-set optimization for the fast `POINT`-list merging 84
37 A finite state machine for stride detection 86
38 Two different dynamic allocation styles for 2D array 87
39 Interval tree for fast overlapping point/stride searching 89
40 A simple example for DYNAMIC-GCD 90
41 Two strides in Figure 40 90
42 A correct program that will shows Figure 41 and Equation (2). 92
43 Structures of the point and stride tables 93
44 A simple example of DAS-ID 96
45 Pseudo code for the table structure with DAS-ID 97
46 Merging two stride lists from the same PC 98
47 A stride kill example 99
48 The hybrid pipeline of SD^3: pipelining and data-level parallelism . . 103
49 An example of the event distribution step with 3 tasks 104
50 A naive address space division scheme 104
51 Data-parallel model of SD^3 105
52 Broken strides from a single stride in the parallelized SD^3 algorithm . 105
53 Deviations of total number of memory accesses of eight tasks 106
54 When too many tasks work in the hybrid parallelization model . . . 107

55 Total trace size for SPEC 2006 109
56 Average trace transfer rate between a tracer and an analyzer 109
57 A false negative case with 1-byte granularity 111
58 Absolute memory overhead for SPEC 2006 with the reference inputs 115
59 Memory overhead of the pairwise for four SPEC 2006 benchmarks. . 116
60 Slowdowns for SPEC 2006 with the reference inputs 117
61 Speedups of SD^3 in SPEC 2006 benchmarks 118
62 Similarity of the results from different inputs 120
63 Classification of stride detection for SPEC 2006 benchmarks 121
64 Work flow of Parallel Prophet. 123
65 An example of an annotated program and its program tree 127
66 A program tree without and with burden factors 129
67 An example of the fast-forwarding method 132
68 An example of nested parallel loops in the FF 135
69 Simplified emulation code and data structures of the FF 136
70 A counterexample for the FF and Suitability algorithms 137
71 An example of a synthesized program 139
72 A general synthesizer for Cilk Plus 141
73 Examples of the coefficients of MPM 149
74 The kernel loop of the microbenchmark 150
75 The improved kernel loop of the microbenchmark 151
76 Controlling L3 cache miss ratio in the microbenchmark 151
77 Self-adjusting code to create an arbitrary cache miss ratio 152
78 The shape of links to compute $CPI_{\$}$ 153
79 Verifications of the microbenchmark 154
80 An example of the annotation system 155
81 Annotation and profiling overhead 158
82 Solving memory overhead by compression 159

83 Verification testers: `Test1` and `Test2` 162
84 Prediction accuracy results of `Test1` and `Test2` 163
85 Inaccurate prediction without considering scheduling policies 164
86 Predictions of OmpSCR and NPB benchmarks 166
87 A simple verification test for memory performance model 168
88 Memory traffic for NPB-FT measured by Intel VTune 169
89 Memory traffics from the major routines of NPB-FT 169
90 LLC misses (MPKI) of NPB-FT's `evlove` and `cffts2` 170
91 A naive example of privatization for output dependences 177
92 A loop that can be parallelized by pipelining 180
93 Simplified parallelization steps of 179.art 184
94 Hot loop information of Susan in MiBench 185
95 Data-Dependence profiling of Susan in MiBench 186
96 Privatizations for the variables in Susan 186
97 The variable `out` in Susan 186
98 A parallelized version of Susan 187
99 Speedups of parallelized Susan in MiBench 188
100 Loop profiling for Dijkstra in MiBench 189
101 Data dependences of Dijkstra in MiBench 190
102 Parallelizing Dijkstra in MiBench based on the post-analyzer 191
103 Speedups of parallelized Dijkstra in MiBench 192
104 The structure of bzip2's `compressStream` 193
105 Data-dependence graph of `compressStream` in 256.bzip 194
106 The three-stage pipeline in `compressStream` in 256.bzip 196
107 Speedups of `compressStream` in 256.bzip 197

SUMMARY

All market-leading processor vendors have started to pursue multicore processors as an alternative to high-frequency single-core processors for better energy and power efficiency. This transition to multicore processors no longer provides the free performance gain enabled by increased clock frequency for programmers. Parallelization of existing serial programs has become the most powerful approach to improving application performance. Not surprisingly, parallel programming is still extremely difficult for many programmers mainly because programmers are not taught well to think in parallel so far. However, we believe that software tools based on advanced analyses can significantly reduce this parallelization burden.

Much active research and many tools exist for already parallelized programs such as finding concurrency bugs and performance profilers for parallel and multithreaded programs. Instead we focus on program analysis algorithms that assist the actual parallelization steps: (1) finding parallelization candidates, (2) understanding the parallelizability and profits of the candidates, and (3) writing parallel code. A few commercial tools are recently introduced for these steps, and a number of researchers have proposed various techniques to assist parallelization. However, many weaknesses and limitations still exist.

In order to assist the parallelization steps more effectively and efficiently, this dissertation proposes *Prospector*, which consists of several new and enhanced program analysis algorithms.

First, an efficient loop profiling algorithm is implemented. Frequently executed loop can be candidates for profitable parallelization targets. The detailed execution

profiling for loops provides a guide for selecting initial parallelization targets.

Second, an efficient and rich data-dependence profiling algorithm is presented. Data dependence is the most essential factor that determines parallelizability. Thus, understanding dependences is critical in parallelization. Prospector exploits dynamic data-dependence profiling, which is an alternative and complementary approach to traditional static-only analyses. However, even state-of-the-art dynamic dependence analysis algorithms can only successfully profile a program with a small memory footprint. Prospector introduces an efficient data-dependence profiling algorithm to support large programs and inputs as well as provides highly detailed profiling information.

Third, a new speedup prediction algorithm is proposed. Although the loop profiling can give a qualitative estimate of the expected profit, obtaining accurate speedup estimates needs more sophisticated analysis. Prospector introduces a new dynamic emulation method to predict parallel speedups from annotated serial code. Prospector also provides a memory performance model to predict speedup saturation due to increased memory traffic. Compared to the latest related work, Prospector significantly improves both prediction accuracy and coverage.

Finally, Prospector provides algorithms that extract hidden parallelism and advice on writing parallel code. We present a number of case studies how Prospector assists manual parallelization in particular cases including privatization, reduction, and pipelining.

CHAPTER I

INTRODUCTION

1.1 The Problem: Parallelizing Serial Code

"The free lunch is over," a famous quote by Herb Sutter [137], succinctly summarizes the perils of the software development we faced in the multicore era. When processors kept doubling their clock frequencies in every new generation, programmers were able to enjoy the *free lunch*, that is, upgraded processors automatically boosted the performance of the software. Since 2003, however, such steep increases in frequency have been stalled mainly because of the power wall as well as several critical challenges of modern computer architectures and limitations of instruction-level parallelism. As of 2012, the number of transistors has been doubling every two years, as predicted by Moore's law [93], although a number of recent studies project a decreased growth rate [58] and the utilization wall [146, 28]. Traditionally, such transistors were used to increase caches and enhance out-of-order pipelines. However, now all leading processor vendors started to pursue energy-efficient multiprocessors as an alternative to high-frequency processors.

The transition to the multicore processor no longer provides free lunch to programmers. The *parallelization* of existing serial programs has become the most powerful approach to improving application performance by exploiting multicore processors. The parallelization problem is becoming more urgent than ever as multicore processors are ubiquitous in mainstream computing, from smart phones,

tablets, to high-end servers. However, concurrent and parallel[1] programming are still extremely challenging concepts for many programmers. This is not surprising because we are not trained well to write an algorithm in parallel in traditional computer science programs. Instead relying on only programmers' efforts, we need advanced support from software and hardware that can make parallel programming easier.

Perhaps compilers may be the ideal tools for exploiting parallelism, as they can perform automatic parallelization. However, Chapter II presents a number of results in which even state-of-the-art compilers miss many parallelization opportunities in small C/C++ programs due to limitations of static pointer analysis. Automatic parallelization, however, fundamentally has a limited applicable domain because only data-dependence-free loops could be parallelized, except for except some reduction variables.

Given such a situation, we believe that *software tools* powered by advanced analyses targeting current multicore processors may significantly reduce the parallelization burden from programmers. Tools for traditional software development (i.e., serial programs) have been developed for decades. These tools, including compilers, debuggers, profilers, and testing tools, are essential for successful large-scale development and productivity. However, what about tools for parallel and multithreaded programming?

Arguably, tools for finding multithreaded and concurrency bugs may be the most researched domains [127, 122, 154, 107], and a number of tools have been developed in this domain [53, 90, 130]. Many commercial profilers for multithreaded programs, which provide detailed thread utilization information

[1]The differences between concurrent and parallel programming are clarified in [27]. A concurrent program (and multithreaded programming) does not necessarily run in parallel physically, but a parallel programming requires a physical multiprocessor. This dissertation takes parallel programming as a representative model.

to diagnose and debug poor parallel performance, also have been developed so far, such as Microsoft Concurrency Visualizer [91], Rogue Wave's ThreadSpotter [121], and Intel Parallel Amplifier [52]. Although these tools are critical for the successful development of parallel and multithreaded programs, they focus on only *post*-parallelization, not *before* and *during* the parallelization process. As a result, programmers are forced to manually parallelize applications. Tools and analysis algorithms that assist the actual parallelization step are urgently needed.

Let us discuss a typical parallelization process from original serial code.[2] This process would include the following four steps:

- Step 1: Finding candidates for parallelization
  - Which loops or functions would be good targets for parallelization?
- Step 2: Understanding the parallelizability and profits of the candidates
  - Can this loop or function be parallelized? If so, what kind of parallelism is available? What about synchronization for correctness? What would be the projected speedup?
- Step 3: Transforming and parallelizing the targets
  - How can I parallelize this code with a parallel programming paradigm such as OpenMP [138], Thread Building Block (TBB) [56], Parallel Pattern Library (PPL) [111], and Cilk Plus [11, 49]?
- Step 4: Verifying and optimizing the parallelized code
  - Does my parallel code work correctly and efficiently?

The tools for concurrency bugs and performance debugging only support the fourth step, that is, after writing parallel code. We argue that tools for the first three

[2] A programmer could write parallel code from the scratch. Even in such case, the programmer mostly needs a serial program for verification. The assumption of the corresponding serial code in parallelization is reasonable.

steps, the actual parallelization steps, are crucial for programmers and advanced compiler optimizations.

So far, a few commercial tools have been introduced to facilitate the subset of the first three steps: Intel Parallel Advisor [51], CriticalBlue's Prism [19], and Vector Fabrics' Pareon (previously vfAnalyst, vfThreaded-x86, and vfEmbedded) [145]. A number of researchers also attempted to attack these steps: HPCToolkit [88, 1], MAPS [14] Embla [30], DAT(Dynamic Analysis Tool) [123], Alchemist [156], Tournavitis and Franke's work [140], DProf [20], Kremlin [33], and Kismet [62]. However, these tools and algorithms currently have a number of weaknesses and limitations, which are discussed in the later chapters.

Therefore, in order to assist the parallelization steps more effectively and efficiently, this dissertation proposes *Prospector*, which consists of four components that are built on several new and enhanced profilers and program analysis algorithms [69].

1.2 The Solution and Contributions: Prospector

Prospector introduces several key algorithms toward an ideal tool that assists the parallelization steps. In particular, Prospector addresses challenging problems of Steps 1 to 3 by presenting the solutions and preliminary results:

- Chapter IV: An efficient loop profiling algorithm for Step 1;
- Chapter V: An efficient data-dependence profiling algorithm for Step 2;
- Chapter VI: A new speedup prediction algorithm for Step 1 and Step 2;
- Chapter VII: Providing advice on writing parallel code and extracting parallelism for Step 2 and Step 3.

An efficient loop profiling algorithm: Frequently executed loops, *hot loops*, are likely to be profitable parallelization targets. The detailed execution profiling for loops provides a quick guide for selecting initial parallelization targets. Popular current profilers [51, 145] do not provide the details of loop execution. An algorithm for loop profiling [94] incurs high overhead for Prospector. To address such limitations, Prospector statically discovers loop and function structures either in binary or source code and then instruments the minimal instructions to capture loop and function executions efficiently, resulting in reduced overhead. The data-dependence profiler of Prospector is built upon this loop profiler.

An efficient data-dependence profiling algorithm: Data dependence (data dependency) is the most essential factor that will determine the parallelizability of a given serial program. Although data dependence can be analyzed statically by compilers, Prospector and related work [75, 51, 145, 156, 140, 20, 33] use *dynamic* data-dependence profiling, which is an alternative and complementary approach to traditional static analyses. However, even state-of-the-art dynamic dependence analysis algorithms can successfully profile only a program with a small memory footprint. This limitation in the scalability and overhead significantly reduces the usefulness of data-dependence profiling. Prospector introduces a new efficient data-dependence profiling algorithm to support large programs and inputs [70]. The key algorithms are exploiting (1) compressible memory streams to reduce the space overhead and (2) parallelism in the data-dependence profiling itself to minimize the time overhead. The data-dependence profiler provides highly detailed information such as dependences in a loop nest.

A new speedup prediction algorithm: Although a serial code section is amenable to parallelization, its speedup may not be compelling for a number of reasons: primarily load imbalance, parallel overhead, and limited scalability of the memory

performance. The loop profiling gives rough estimates of the expected profit, but more sophisticated analysis is needed to obtain accurate parallel speedup estimates. Recently, a couple of dynamic approaches to the practical speedup prediction were proposed [51, 62]. Prospector basically takes a similar approach to the Suitability analysis in Intel Parallel Advisor [51] but proposes two new analysis algorithms [71] to overcome the current limitations. First, a new emulation algorithm for speedup prediction is proposed to improve both prediction accuracy and coverage by easily supporting more complex and broad parallel programming patterns. Second, we propose a memory performance model to predict memory bandwidth saturation, which is one of the major causes of saturated and degraded parallel performance.

A post-analysis algorithm to guide parallelization: To complete the full picture of Prospector, we finally introduce post-analysis algorithms that extract hidden parallelism and provide advice on writing parallel code from the raw results of the data-dependence profiler. The raw results are enumerations of discovered dependence pairs, which give detailed information of data dependences in serial code. However, these raw results do not directly expose potential parallelism. Programmers must extract information on code transformation in the parallelization steps. We attack this difficult problem by devising new and advanced post-analysis algorithms. A couple of essential parallelism models, including reduction and pipelining, are investigated. Based on the dependence result and post-analysis algorithms, Prospector will provide direct advice on writing parallel code. This information will be very informative for both manual parallelization and compiler-assisted automatic parallelization. To summarize, Prospector will provide integrated advice on the parallelization steps, from finding profitable parallelization targets to providing advice on writing parallel code.

1.3 Thesis Statement

Program analysis algorithms in Prospector can significantly reduce the burdens of manual parallelization from serial code by providing detailed information of loop execution, data dependences, speedup estimates, and parallelization advice.

1.4 Organization of This Proposal Document

This document is organized as follows: Chapter II presents the motivations and overview of the proposed algorithms in Prospector. Chapter III summarizes the related work. Chapter IV describes the details of Prospector's efficient loop-profiling mechanism. Chapter V discusses the algorithms and architecture of Prospector's new data-dependence profiler. Chapter VI elaborates the state-of-the-art algorithm in predicting speedups and then presents a new speedup prediction algorithm and our memory performance model. Finally, Chapter VII describes how Prospector can guide manual parallelization and extract hidden parallelism.

CHAPTER II

MOTIVATIONS AND OVERVIEW OF PROSPECTOR

This chapter discusses technical motivations for the algorithms in Prospector with examples and then sketches the overview of Prospector.

2.1 Motivations for an Efficient Loop Profiler

As a rule of thumb, functions dominating the execution time, *hot functions*, are the main targets of optimizations. The same can be said of parallelization, at least for task parallelism (also known as function parallelism). Traditional profilers for sequential programming, notably GNU gprof [34], OProfile [105], and Intel VTune Amplifier [57], provide detailed information of hot functions, including the percentage of the total execution time spent in a function, cumulative and self-time, and call graphs. This profiling information is sufficient to select initial parallelization targets for task parallelism.[1]

Exploiting data parallelism or loop-level parallelism may need more information than traditional profiling data. This methodology parallelizes loop iterations. Programmers would like not only to understand *hot loops* where the majority of the execution time is spent, but also to have more information regarding the loop execution, such as trip count and variance of iteration length.

A simple modification on traditional profilers would provide the hot loop information easily. The start and termination of a loop are considered to be those of a function. Then, a profiler may measure the aggregated execution time of a loop as

[1]Even if a profiler locates a hot function, this function may need heavy synchronization, resulting in no profit. Although the hot function information is very important in finding parallelization targets, this information gives only an initial hint.

Function Call Sites and Loops	Total Time %	Total Time	Self Time	Source Location
⊟ Total	100.0%	1.7157s	0s	
⊟ RtlInitializeExceptionChain	100.0%	1.7157s	0s	
⊟ RtlInitializeExceptionChain	100.0%	1.7157s	0s	
⊟ BaseThreadInitThunk	100.0%	1.7157s	0s	
⊟ _tmainCRTStartup	100.0%	1.7157s	0s	crtexe.c:555
⊟ main [loop]	100.0%	1.7157s	0s	main.cxx:130
⊟ main [loop]	73.7%	1.2648s	0s	main.cxx:131
⊟ main	73.7%	1.2648s	0s	main.cxx:131
⊞ Puzzle::Solver::Solver [loop]	62.1%	1.0656s	0s	solver.cxx:154
⊞ Puzzle::Solver::solve [loop]	11.0%	0.1892s	0s	solver.cxx:289
⊞ Puzzle::Solver::Solver [loop]	0.6%	0.0100s	0s	solver.cxx:139
⊞ main	24.8%	0.4253s	0s	main.cxx:131
⊞ main [loop]	1.5%	0.0256s	0s	main.cxx:131

Figure 1: A simple loop profiling in Intel Parallel Advisor shows only the total execution time of each loop This capture is the sudoku program in the example programs of Parallel Advisor.

shown in Figure 1. For example, the profiler of Intel Parallel Advisor reports both hot loops and functions. The traditional call stack is also combined with loops as if a loop were a function.

However, this hot loop information is not enough to find good initial targets for loop-level parallelism. Our experience shows that the following profiling data is very helpful: (1) invocation count (how many times a loop is executed); (2) average trip count per a loop invocation (the total iteration number of a loop divided by the invocation count); and (3) statistics of iteration lengths (e.g., average, minimum, maximum, and variance).

First, the invocation count for a loop gives a rough idea of synchronization overhead for the parallel loop (e.g., the overhead of OpenMP's `#pragma omp parallel for`). In particular, this information is important when an inner loop is to be parallelized. If an inner loop is invoked excessively while its average loop length is small, a good speedup may not be expected due to high spawn/join overhead. Second, average trip count is an effective indicator of a good parallelization target. A hot loop with small trip count (e.g., less than the number of cores) may not able to utilize all multiple execution units. A

```
307   for(item = 0 ; item < group->n_scheditems; item++)
308   {
309     switch(group->scheditems[group->order[item]].type)
310     {
311       case sched_function:
312         ScheduleTraverseFunction(...);
...
321         break;
322       case sched_group:
323         ScheduleTraverseGroup(...);
...
335         break;
338     }
339   }
```

Figure 2: A hot loop of 436.cactusADM, `ScheduleTraverse.c`, has a trip count of six and performs computation by the switch. However, the last iteration has an actual computation while the other iterations do not perform any work.

sufficiently large trip count has more potential to achieve a decent speedup and better scheduling opportunity. Third, basic statistics for iteration lengths can be useful to predict workload imbalance. We present an example for this case.

Figure 2 shows one of the hottest loops of 436.cactusADM in SPEC CPU2006 [134]. For the train input, the trip count of the `for` is profiled as six, and the invocation count is only one. This profiling information infers that parallelizing this loop seems to be good. However, we soon discovered that only the last iteration actually performed the computation. Simply parallelizing this loop (e.g., by `omp parallel for`) does not give any speedup. Although this is an extreme case of the load imbalance in parallelization, understanding such deviations of the lengths of loop iterations is highly informative to programmers.

The loop profiler can give such quick performance estimation, but this is not for precise parallel speedup prediction. Prospector also provides a separate speedup predictor, *Parallel Prophet*, which is presented in Chapter VI. Parallel Prophet performs more sophisticated loop profiling per iteration granularity (interval profiling, Section 6.2.2) and emulating the expected parallel execution. This prediction mechanism may require significantly more space to store the

profiling information. In contrast, the loop profiler in this chapter collects only single-dimensional statistics for each loop with negligible memory overhead. The purpose of the loop profiler is to provide factual execution profiling information like gprof rather than predicting accurate speedups.

More important, the loop profiler is the platform of the data-dependence profiler of Prospector, SD^3. Note that the loop profiler is a degenerated form of our data-dependence profiling: If a loop iteration does not have any memory load and store (except for a loop induction variable), the dependence profiling on this loop is identical to the loop profiling. In this sense, achieving optimal performance on the loop profiling is particularly important. Furthermore, our dependence profiler itself is parallelized to minimize the time overhead. Each parallel task in the profiler performs its own loop profiling. Hence, an optimal loop profiler is a prerequisite for the efficiency of the dependence profiler.

Parallel Advisor currently does not provide the loop execution information such as average trip counts. Moseley et al. proposed LoopProf [94], which is similar to our loop profiler, providing the details of loop execution. However, on average, LoopProf is 22 times slower than the native execution of SPEC CPU2000 benchmarks without any sampling. Such overhead is unacceptable for our dependence profiler. We present an optimal loop profiler that minimizes the instrumentation and runtime overhead. Our loop profiler has only a 6.2 times slowdown on average. Chapter IV details the mechanism.

2.2 Motivations for Dynamic Data-Dependence Analysis

One of the key components in Prospector is *dynamic data-dependence analysis*, which we call the *data-dependence profiler*. This section discusses why the data-dependence profiler is needed to assist parallelization effectively.

Data dependence indicates whether two instructions (either statements or

tasks) access the same memory location while at least one of them is a write operation.[2] Data dependence has three cases: (1) flow dependence (read-after-write, RAW), (2) anti-dependence (write-after-read, WAR), and (3) output dependence (write-after-write, WAW). Among them, flow dependence is considered true dependence while the others are false dependences (or name dependences). False dependence can be easily avoided using register-renaming-like techniques, but most true dependence are the semantics of the program.

The implication of data dependence in parallelization is clear. Data-independent instructions can be safely executed in parallel without the need for synchronization. Dependent instructions, especially true dependence, should be correctly handled by synchronization such as barriers, condition variables, and mutexes. In a word, the existence and patterns of data dependences in serial code decide the parallelizability of the code.[3] Hence, understanding the dependence in serial code is the most critical step toward developing tools to assist the parallelization step, particularly for parallelism discovery and parallelization guidance.

Traditionally, data-dependence analysis has been done *statically* by compilers. Algorithms such as the GCD test [97], Banerjee's inequality test [73], and the I Test [73] are used to analyze data dependences in array-based data accesses. These static analyses may not be effective in languages that allow pointers and dynamic allocation because array accesses would become general pointer arithmetics. Pointer analysis is needed to determine dependences statically in such cases.

Theoretically, precise pointer analysis is known as undecidable for languages (notably, C and C++), where arbitrary pointers and dynamic allocations are allowed [15]. There is a large body of research related to pointer analysis and

[2] Data dependence across multiple threads is not considered in this work.

[3] Sometimes data dependence may be too conservative constraint to determine parallelizability. There could be a number of dependences created by artifacts such as compiler-generated code [143]. Commutativity [119] and critical path analysis [33] could also be criteria to judge parallelizability.

its approximations [45]. Notable earlier works are Andersen's algorithm [7] and Steensgaard's algorithm [135], and some recent efforts can be found in [78, 40]. Because of the nature of such challenges in pointer analysis, static compiler-based parallelism discovery and automatic parallelization are limited by their pointer analysis. However, static compilers must be correct on optimizations. They have to take a conservative approach. The correct dependences (either always dependent or always independent) between two pointers are often unknown at compile time. A static pointer analysis has to return a *may* dependence in this case. Consequently, the opportunity for advanced optimizations such as automatic parallelization could be significantly reduced.

Dynamic data-dependence analysis or *data-dependence profiling* is an alternative and complementary approach to static-only dependence analysis, which checks data dependences in the runtime using actual memory addresses [75, 108]. Because real addresses are resolved in the runtime, dependence can be determined without a may answer. Data-dependence profiling has been used in various parallelization efforts, including hardware-based speculative parallelization [136, 81, 16, 25, 82, 151] and profiling-assisted parallelization studies [75, 10, 30, 123, 148, 156, 140, 20, 33]. It is also being deployed in commercial tools [51, 19, 145].

Despite the attractiveness of dependence profiling, two weaknesses exist: (1) profiling overhead, which is detailed in the next section, and (2) the *input-sensitivity* problem, where discovered data dependences by a profiler are only the results of particular input sets. In turn, parallelism extracted from data-dependence profiling is *potential* parallelism, not a proved one. However, this input-sensitivity problem is inevitable for all dynamic program analysis techniques. For instance, the majority of memory leaks and data-race detectors are implemented as dynamic tools such as Valgrind [98] and Intel Parallel Inspector [53]. Fixing all discovered memory leaks and races does not guarantee the safety of the program. This

inherent limitation, however, cannot depreciate the value of such dynamic analysis tools. These tools are critical for successful software development. Profile-based data-dependence analysis of Prospector is also based on the same premise. Even with this weakness, we argue that using data-dependence profiling is very helpful for programmers and aggressive compiler optimizations, and it is worth exploring as a research topic and a practical tool.

Our position is that parallelization is mostly done in frequently executed code. In a hot section, important dependence patterns are highly unlikely to be affected by different input sets. Our experiments presented in Chapter V also support this intuition. Furthermore, when programmers are forced to do manual parallelization because a compiler cannot do automatic parallelization, programmers themselves must prove the safety of their parallelization. Because such formal proof is almost impossible, programmers resort to extensive and thorough testing. Data-dependence profiling can significantly mitigate this painful manual parallelization by providing strong *hints* for data dependences. Recent commercial tools [51, 19, 145] and research that use profiling to assist parallelization [139, 140, 124, 33] also advocate this claim.

Finally, our data-dependence profiler is not a pure dynamic tool. We perform static analysis for better performance and quality (See Section 5.5.3). One of the future works is to investigate more opportunities in static analysis. A data-dependence profiler would be eventually a hybrid form of both static and dynamic analyses that can provide provable parallelism information.

2.2.1 Case Studies: Automatic Parallelization in C/C++ Compilers

This section presents case studies that demonstrate the weaknesses of automatic parallelization and static analysis. Instead of using research parallelizing compilers such as SUIF [150] and Polaris [9], two state-of-the-art compilers (as of

2010) are used for practical reasons: Intel C/C++ compiler 12.0 (ICC) [50] and the Portland C/C++ compiler 8.0 (PGC) [109]. These compilers are used to parallelize the OmpSCR [103, 24] benchmarks, a set of small mathematical benchmarks that are manually parallelized by programmers using OpenMP pragmas. Except for a few reduction variables, the tested benchmarks are embarrassingly parallelized. While these state-of-the-art compilers support automatic parallelization, they often fail to parallelize C/C++ programs.

All available compiler options are calibrated to maximize parallelization opportunities. For example, inter-procedural optimizations, inlining, and alias analyses are enabled. Cost-benefit analysis is disabled (e.g., `-par-threshold0` in ICC) so that the compiler will parallelize loops whenever possible. PGC did not have an option to disable the cost-benefit analysis when we performed the experiment. Our goal is to see how many of the manually parallelized loops can be automatically parallelized.

Table 1 summarizes the results of automatic parallelization with both compilers. The second column shows the number of loops manually parallelized by the programmer. The third and fifth columns show how many of these manually parallelized loops are automatically parallelized by ICC and PGC, respectively. Based on the diagnostic reports from the compilers, the reasons for failure are estimated and briefly noted in the fourth and sixth columns. For example, ICC provides detailed diagnostic reports if the reporting option is enabled. The last column of Table 1 shows the number of loops that are discovered as parallelizable loops by our dependence profiler, SD^3.

Overall, ICC parallelizes four of the 13 manually parallelized loops, while PGC parallelizes none of them. Recall that the cost-benefit analysis of PGC could not be disabled. The PGC used in the experiment did not support a profiling-based optimization for the automatic parallelization to overcome the weakness of the

Table 1: OmpSCR Automatic Parallelization Results: How many loops parallelized by the programmers can be automatically parallelized by the compilers? How many loops can be found as parallelizable by a dependence profiler?

Benchmark	Programmers	ICC #	Reason	PGC #	Reason	SD^3
FFT	2	1	Recursion	0	No benefit	3
FFT6	3	0	Pointers	0	No benefit	16
Jacobi	2	1	Pointers	0	Pointers	8
LUreduction	1	0	Pointers	0	Pointers	4
Mandelbrot	1	0	Reduction	0	Multi-exits	2
Md	2	1	Reduction	0	Pointers	10
Pi	1	1	N/A	0	No benefit	1
QuickSort	1	0	Recursion	0	No benefit	4
TOTAL	13	4	N/A	0	N/A	48

static-only cost-benefit analysis. The compilers also parallelize other loops that are not parallelized by the programmers, but most of them are not profitable.

The numbers in SD^3 are greater than the numbers in "Programmers" because SD^3 reports all parallelizable loops without considering benefit. However, note that SD^3 successfully identifies *all* parallelizable loops that were manually parallelized by the programmers while the compilers were unable to do so. This result clearly shows the effectiveness of dynamic dependence profiling. We now detail the reasons why the compilers were not effective to find parallelism.

Pointer-based Accesses The most critical reason for the failures is the weakness of the pointer analysis of the compilers. C99 [60], which is a new standard of C language, supports the `restrict` keyword to minimize pointer overlapping. Even if we manually inserted the `restrict` keyword whenever possible, the results were the same.

LUreduction in Figure 3(a) is parallelizable at the second-level loop among the three nested loops. The two two-dimensional `double` arrays, `L` and `M`, are dynamically allocated, being accessed via `double**` pointers. ICC cannot

```
1  // Allocate two 2D arrays.
2  double** L = (double**)OSCR_calloc(...);
3  double** M = (double**)OSCR_calloc(...);
4  for (int i = 0; i < N; i++) {
5    M[i] = (double*)OSCR_calloc(size, sizeof(double));
6    L[i] = (double*)OSCR_calloc(size, sizeof(double));
7  }
8
9  ... Initializations
10
11 for(int k = 0; k < N-1; k++) {
12   // This for loop is parallelizable.
13   for (int i = k + 1; i < N; i++) {
14     L[i][k] = M[i][k] / M[k][k];
15     for (int j = k + 1; j < N; j++)
16       M[i][j] = M[i][j] - L[i][k] * M[k][j];
17   }
18 }
```

(a) LUreduction

```
1  int gen_v_table(complex *v, int n) {
2    complex wn;
3    wn.re =  cos(PI * 2.f / (n * n));
4    wn.im = -sin(PI * 2.f / (n * n));
5    // This loop is parallelizable.
6    for (int j = 0; j < n; ++j) {
7      for (int k = 0; k < n; ++k) {
8        v[j * n + k] = complex_pow(wn, j * k);
9      }
10   }
11   return 0;
12 }
```

(b) FFT6

Figure 3: LUReduction and FFT6 benchmarks in OmpSCR: State-of-the-art C/C++ compilers may not able to parallelize these parallel loops.

parallelize the second-level loop due to potential flow dependences and anti-dependences on `M`.

The code has two major challenges for successful pointer analysis: (1) a custom memory allocator `OSCR_calloc` and (2) the allocation style for two-dimensional arrays (each row is separately allocated). To alleviate these two difficulties, we modified the source code. First, a custom memory allocator `OSCR_calloc` is

replaced with the standard `calloc` to reduce the difficulty of considering context-sensitive analysis. Notice that `OSCR_calloc` is a simple wrapper of `calloc`. Second, the structures of `L` and `M`, as well as other accessing code, are converted to one-dimensional arrays. Without such transformation using one-dimensional arrays, an advanced analysis such as shape analysis [126] is needed to detect the memory access pattern on two-dimensional arrays. We speculated that a technique such as heap cloning [78] would solve the pointer analysis problem of LUreduction after the changes. Unfortunately, the compiler was still unable to find the parallelism, reporting potential data dependences on `M`. Only after these two arrays are statically allocated (e.g., `double M[1024][1024]`), does ICC parallelize the loop.

The compilers also failed to parallelize arrays whose bounds are unknown at compile time. Figure 3(b) illustrates this case. ICC had to obey dependences conservatively, resulting in output dependences on the array `v`. If the length of the array, `n`, is set to a constant value, the loop is parallelized.

Pointer-linked data structures such as linked lists and tree structures pose an even greater challenge to compilers. The tested benchmarks in OpenMP do not have such data structures; thus some of the Olden benchmarks [120] are used for this experimentation. No automatic parallelization was observed. ICC reported no diagnostic message. We speculate that the compiler simply ignores parallelization attempts once pointer-linked data structures are accessed in loops.

Complex Control Flows The compilers were unable to parallelize loops that have *irregular control flows* such as branches, breaks, early returns, and function calls. Mandelbrot and Md in OmpSCR were not parallelized for this reason.

Figure 4 shows Mandelbrot, in which the outer loop should be parallelizable by realizing that `outside` is a reduction variable. However, the fact that `outside` is conditionally updated at line 7 as well as the potential early exit at line 8 confuses

```
1  // This loop is parallelizable with a reduction on outside.
2  for(i = 0; i < NPOINTS; i++) {
3    complex z = points[i];
4    for (j = 0; j < MAXITER; j++) {
5      z = z * z + points[i];
6      if (abs(z) > THRESOLD) {
7        outside++;
8        break;
9      }
10   }
11 }
12 inside = NPOINTS - outside;
```

Figure 4: Mandelbrot in OmpSCR: The compilers used in our experimentation were unable to detect these parallel loops due to control flow with a reduction.

the compiler. The outer loop is not automatically parallelized.

FFT shown in Figure 5 is parallelized by exploiting recursion. This pattern is a typical "divide and conquer" in which the sub-tasks can be concurrently processed. The loop at 12 is parallelizable without synchronization. Note that the trip count of the loop is two, which means the task is divided into two. However, the compilers fail to recognize that the data accessed by the two paths are independent. As a result, the loop is not parallelized by the compilers. A similar pattern is also found in Quicksort in OmpSCR. Quicksort spawns two concurrent sub-tasks, but the compilers were unable to detect the parallelism.

Code that contains C++ virtual functions and indirect function calls (e.g., callbacks) was not often parallelizable. A loop is automatically parallelizable if these function calls are inlined. We found this case in an image registration function in ITK [46].

Insufficient Cost-Benefit Analysis The compilers statically analyze the cost and benefit of parallelization. As seen in the results, PGC misjudged many hot loops. PGC concludes that many hot loops provide no benefit. The cost-benefit analysis of ICC also has weaknesses. When internal cost-benefit analysis was enabled, only two loops were parallelized. A lack of dynamic execution information of loops is

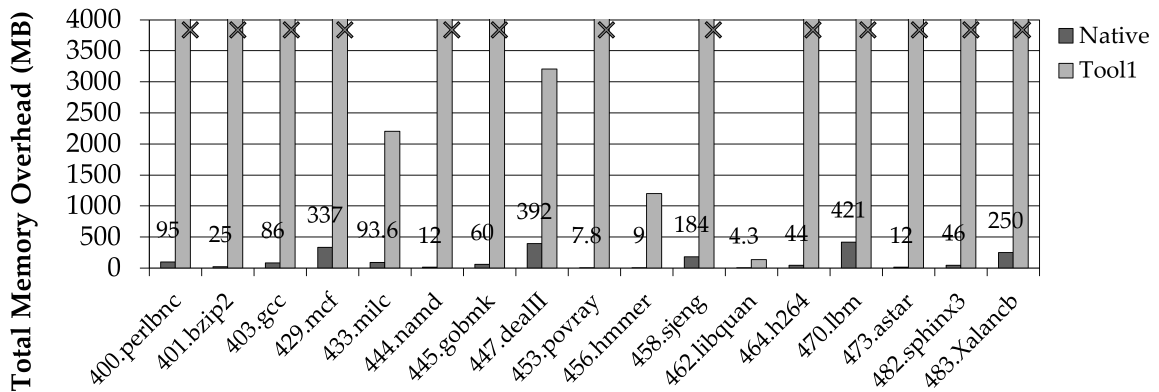
```
1  void FFT(Complex *A, Complex *a, Complex *W,
2    unsigned N, unsigned stride, Complex *D)
3  {
4    if (N == 1) {
5      A[0] = a[0];
6    }
7    else {
8      // Division stage without copying input data.
9      unsigned n = N / 2;
10
11     // This loop is parallelizable.
12     for (int i = 0; i <= 1; i++)
13       FFT(D + i*n, a + i*stride, W, n, stride << 1, A + i*n);
14
15     // This loop is parallelizable.
16     for (int i = 0; i <= n - 1; i++) {
17       Complex* pW = W + i * stride;
18       Complex Aux = pW * (D + n)[i];
19       A[i]     = D[i] + Aux;
20       A[i + n] = D[i] - Aux;
21     }
22   }
23 }
```

Figure 5: FFT in OmpSCR: The compilers used in our experimentation were unable to detect these parallel loops due to complex control flow: recursion.

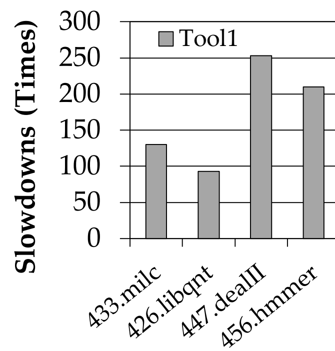
also another major limitation of automatic parallelization. Tournavitis et al. [141] proposed a profile-driven mechanism that may address this problem.

2.3 *Motivations for An Efficient Data-Dependence Profiler*

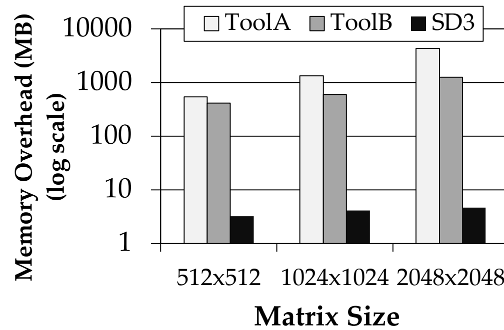
The previous section used a number of examples to emphasize why a data-dependence profiler is an attractive alternative to static-only approaches. This dynamic approach addresses the fundamental limitation of static pointer analysis because all accessed addresses are known at the runtime. However, like other dynamic analyses, significant time overhead is a major obstacle to adapt a dependence profiler for practical usages. Worse, data-dependence profiling has another severe problem: heavy memory consumption, which is often beyond a typical computing resource. The current algorithms for data-dependence profiling incur significant costs of time and memory overhead. Surprisingly, although



(a) Memory overhead of Tool1 (×: Dependence profiling takes 10+ GB memory.)



(b) Time overhead of Tool1



(c) Memory overhead of matrix addition

Figure 6: Overhead of the current commercial data-dependence profilers: "Tool1" and "ToolA" are Intel Parallel Advisor, and "ToolB" is vfAnalyst [145]. "Native" in Figure (a) refers to the memory overhead without the profiler. SD^3 is the new efficient data-dependence profiler in Prospector, presented in Chapter V.

a number of research works have focused on using data-dependence profiling, no work exists on addressing the performance and overhead issues in data-dependence profilers.

As a concrete demonstration, Figure 6 shows the overhead of the dependence profiling algorithms of the two *commercial* tools: Intel Parallel Advisor and Vector Fabrics' vfAnalyst [145].[4] As of 2012, these tools still provide state-of-the-art dependence profilers. Because no detailed profiling algorithms of the tools are

[4]The data in Figure 6 was obtained in May 2010. We used the first version of Intel Parallel Advisor and Vector Fabrics' vfAnalyst, the previous version of Vector Fabrics' Pareon. In early 2012, we still cannot see that these tools improved the overhead problem.

publicly available, we implement our own baseline algorithm calked *the pairwise method*. We believe the pairwise method is very similar to the algorithms of these commercialized tools. Figure 6(a) and 6(b) show the memory and time overhead of a commercial data-dependence profiler, respectively, when profiling 17 SPEC CPU2006 C/C++ applications with the train inputs on a 12 GB machine. Because of the limitation of the tool, the top 20 hottest loops and their children loops were profiled. Among the 17 benchmarks, only four benchmarks were successfully analyzed; the rest of the benchmarks failed because of insufficient physical memory (consumed more than 10 GB memory). Simply doubling the main memory would not be a solution. A couple of programs are profiled on a 24 GB machine, but the tool was still unable to profile them successfully.

Another example that clearly shows the memory overhead problem is a simple matrix addition program that allocates three N $\times$ N matrices for the computation: $A = B + C$. As shown in Figure 6(c), the current tools require significant additional memory as the matrix size increases, while our new method, SD^3 , the new efficient data-dependence profiler in Prospector, needs only very small (less than 10 MB) memory by compressing the memory references.

The runtime overhead is between an 80 times and a 270 times slowdown for the four benchmarks that worked, shown in Figure 6(b). In fact, these numbers are not significantly high. It is well known that many dynamic analyses or profiling tools typically incur slowdowns of a hundred times or worse [125, 148, 107]. The Tool1 in Figure 6(b) does not fully calculate data dependences in loop nests; thus the observed overhead is somewhat low. Our baseline profiler, the pairwise method described in Section 5.2, suffers from a couple of hundred times slowdown because all characteristics of data dependences, including nested loops, are computed. We also solve this time overhead problem.

While both time and memory overhead are severe, the latter will stop further

analysis. The culprit is the dynamic data-dependence profiling algorithm used in these tools, which is very similar to the pairwise method. The pairwise method needs to store all outstanding memory references in order to check dependences, resulting in potentially large memory bloats.

Chapter V addresses these memory and time overhead problems by proposing SD^3, an efficient data-dependence profiling algorithm. For the memory overhead, SD^3 introduces a stride-based *compression* technique and a new dependence calculation algorithm. To minimize the time overhead, SD^3 itself is *parallelized* by data-level parallelization and pipelining.

2.4 Motivations for Correct Data-Dependence Profiling

Regarding the overhead problem, one may consider sampling and other approximation techniques, which are practically proven methods to reduce the overhead in many profiling problems. For example, a number of profilers such as gprof [34], OProfile [105], and VTune [57] are statistical profilers that use sampling with some instrumentation. Data-dependence profilers for speculative optimization, notably Thread-Level Speculation (TLS) [136, 16, 82], typically employ samplings and simplified data structures to minimize the profiling overhead [16]. Time overhead is reduced by changing sampling rate. To reduce space consumption, data dependences are tracked at coarse granularity such as cache line size.

However, these techniques are inadequate to guide parallelization that targets conventional multiprocessors (i.e., non-speculative) for two reasons: (1) any sampling technique may introduce incorrect results, and (2) a simple sampling is not effective to solve the memory overhead problem. We first discuss why the *correctness* of data-dependence profiling is important.

As mentioned in Section 2.2, a dynamic dependence profiler may not discover all existing dependences in a program for all possible inputs. The correctness does

not mean that a profiler should find all inherent dependences. Instead, we define the correctness of a data-dependence profiler as follows:

- We first define the z/correctness of a profiling result from a dependence profiler. Suppose a program P and all available inputs (could be infinite) I. With an ideal and perfect dependence profiling, suppose D is the set of the existing all data dependences in P with I:

  $$D = \{(s, t, d, f) \mid s \subseteq L,\ t \subseteq L,\ d \subseteq \mathbb{Z},\ f \subseteq \mathbb{N},\ \text{and} f > 0\},$$

  where s, t, d, and f are the source, sink, dependence distance, and frequency of a dependence tuple, respectively; L is the set of all executed memory locations in P with I.

  For a given input $I_i \subseteq I$, suppose that the correct profiling result from the same ideal and perfect profiler is D_i:

  $$D_i = \{(s, t, d, f) \mid s \subseteq L_i,\ t \subseteq L_i,\ d \subseteq \mathbb{Z},\ f \subseteq \mathbb{N},\ \text{and} f > 0\},$$

  where L_i is the set of all executed memory locations in P with I_i. Because $L_i \subseteq L$, $D_i \subseteq D$. This implies a single dynamic profiling result may not discover all inherent dependences in a program.

- Now, suppose $\hat{D}_i$ is a profiling result from an implemented dependence profiler with P and I_i. We say that $\hat{D}_i$ is *strictly correct* if and only if:

  - For every dependence element $\hat{d}_i$ in $\hat{D}$, $\hat{d}_i$ exists in D; and
  - For every dependence element d_i in D, d_i exists in $\hat{D}$.

  However, for more flexible implementation and optimization opportunities, we relax the conditions of the correctness by omitting the equivalence of the dependence frequency. We say that $\hat{D}_i$ is *correct* if and only if:

- For every dependence element $\hat{d}_i$ in $\hat{D}$, $\hat{d}_i$ exists in D, but $\hat{f}$ of the $\hat{d}_i$ does not necessarily equal f of the corresponding d; (no false positive) and
- For every dependence element d_i in D, d_i exists in $\hat{D}$, but f of the d_i does not necessarily equal $\hat{f}$ of the corresponding $\hat{d}$ (no false negative).

- Finally, a dependence profiler is correct if and only if, for every possible input $I_i \subseteq I$, this profiler must yield a correct or a strictly correct result for I_i.

In other words, as long as all profiling results preserve the existence and distance of data dependences, we consider this profiler to be correct. The relaxation of the frequency is needed for our optimization (PC-set optimization in Section 5.2.6) and compression (The definition of stride in Section 5.3.8).

We never compromise the existence of the dependences, which is particularly critical to assist non-speculative parallelization. Incorrect data-dependence profiling can lead to either a false negative or a false positive. In the worst case, a dependence pair could be missed due to sampling or other approximation, but this pair can prevent safe parallelization. In some usage models of data-dependence profiling, parallelization based on an incorrect profiling result can be tolerated. One such example is Thread-Level Speculation (TLS). TLS hardware can recover a conflicted parallel execution via the rollback mechanism. This thesis, however, does not assume a speculative parallelization, including TLS and transactional memory. We target conventional multicore processors and non-speculative parallel programming model.

A sophisticated and advanced sampling technique could be applied. In the case of data-race detectors, early detectors did not use sampling [127, 122]. However, 'intelligent filtering is applied to a data-race detector (Intel Thread Checker) [125] without compromising accuracy. Although sacrificing accuracy a little bit, LiteRace [87] further exploits an adaptive sampling method based on a novel observation:

data races are likely to occur in a cold region. However, to the best of our knowledge, for dependence profiling, no advanced sampling is devised to reduce the overhead while maintaining the correctness.

2.4.1 Experimentation Results of Simple Sampling Techniques

We present experimental results that support the claim that simple sampling techniques are not suitable for non-speculative parallelization. In this experimentation, we apply three types of sampling policies to the pairwise method:

(1) Burst sampling: Dependence profiling is triggered every N instructions and then lasts for M consecutive memory instructions. We fix N to be 5,000 and M to be 1,000 in this evaluation.

(2) Early stopping: For any given loop, we profile up to N% of its total invocation count and M% of its total iteration count. We use (100%, 10%), (10%, 100%), and (10%, 10%) in this experimentation.

(3) Major-loops only: We perform dependence profiling only on major loops. A loop is considered as a major loop if its execution time is at least 5% of the total execution time.

Note that we do not consider a simple random sampling of memory accesses. We discuss the accuracy and the overhead issues in these sampling techniques.

Issue with Profiling Accuracy Table 2 shows the numbers of incorrectly reported loops in terms of their parallelizability for an input, among the hot loops in OmpSCR benchmarks [103]. It is clear that the burst-sampling policy should be avoided because of its high inaccuracy. Early-stop samplings are better, but still are not close to 100% accuracy.

Table 2: The numbers of incorrectly reported loops by simple sampling policies (lower is better): For example, among six loops of Jacobi, the parallelizability of four loops was incorrectly reported by burst sampling; the three early-stopping policies also made mistakes.

Policies	Burst	100%/10%	10%/100%	10%/10%	Total Loops
LU	1	0	0	0	3
FFT	2	0	0	0	3
FFT6	6	0	0	0	11
Jacobi	4	2	2	2	6
md	1	0	1	1	5
qs	3	2	0	2	5
Mandel	1	0	0	0	3
TOTAL	18	4	3	5	36

Figure 7 illustrates how early-stop sampling could result in inaccurate dependence profiling. The array v is read by lines 6 and 8 and can be written by lines 12 and 13, depending on the conditional branch at line 10. If the branch always goes to one side before the sampling stops and to the other side afterward, the dependence profile will be incorrect.

We also found another counter example of early-stop sampling in SPEC2006 436.cactusADM as shown in Figure 8. The loop at 307 is mistakenly reported by early-stop sampling as parallelizable because only the last few iterations of the loop exhibit loop-carried dependences.

Issue with Memory Overhead A sampling may reduce the time overhead, but not necessarily the memory overhead. Figure 9 shows the memory overhead of OmpSCR's Jacobi when various profiling mechanisms are applied: (1) "Native" indicates the native run without profiling; (2) "Baseline" corresponds to the pairwise method without sampling; (3) "SD^3" is our proposed memory-efficient mechanism; (4) the rest are pairwise method with different sampling policies.

Jacobi's main data structure is a matrix whose memory requirement increases quadratically with respect to the input parameter (the matrix dimension size).

```
1  void qs(int *v, int first, int last) {
2    int start[2], end[2];
3    start[1] = first, end[0] = last;
4    ...
5    while (start[1] <= end[0]) {
6      while (v[start[1]] < pivot)  // Read v[..]
7        start[1]++;
8      while (pivot < v[end[0]])    // Read v[..]
9        end[0]--;
10     if (start[1] <= end[0]) {
11       int temp = v[start[1]];
12       v[start[1]] = v[end[0]];   // Write v[..]
13       v[end[0]] = temp;          // Write v[..]
14       start[1]++, end[0]--;
15     }
16   }
17   ...
18 }
```

Figure 7: The partitioning algorithm in Quicksort

```
307  for(item = 0 ; item < group->n_scheditems; item++)
308  {
309    switch(group->scheditems[group->order[item]].type)
310    {
311      case sched_function:  // call work function
           ...
322      case sched_group:     // call work function
           ...
338    }
339  }
```

Figure 8: From ScheduleTraverse.c in 436.cactusADM

When the matrix is 3000x3000, the pairwise method requires over 12 GB. Even when sampling techniques are used, the memory overhead is still quite high: an order of gigabytes of memory. When the matrix is 5000x5000, the best sampling result (burst sampling) consumes nearly 4 GB. In contrast, SD^3 requires only 600 MB, which is just 1.1 times the native memory consumption, by completely compressing most of the memory references. In such a perfect compression case, the memory overhead of SD^3 is asymptotically close to the native memory consumption. Simple sampling techniques do reduce memory burdens but may require significant additional memory by an order of magnitude.

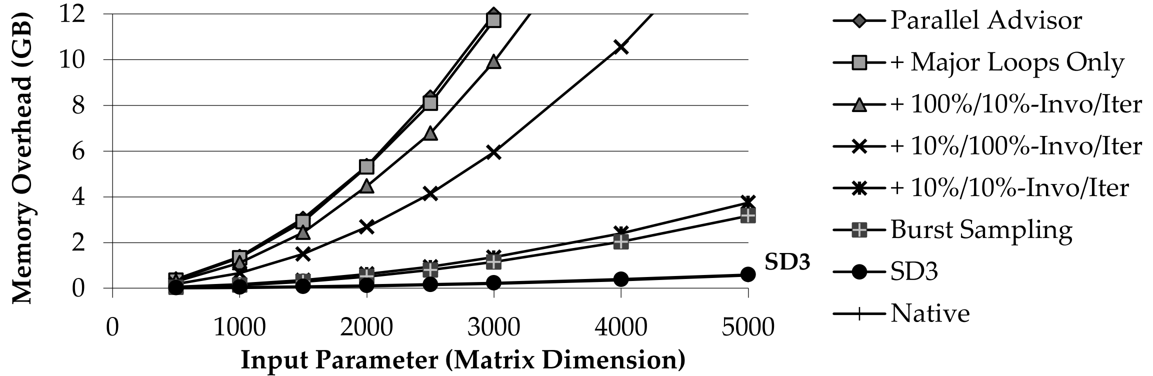


Figure 9: Memory overhead of Jacobi in OmpSCR: Total memory consumptions are measured with different profiling mechanisms and sampling policies.

2.5 *Motivations for Dynamic Parallel Speedup Prediction*

Prospector has a parallel speedup prediction component, *Parallel Prophet*, which is based on *dynamic* profiling and emulations instead of on analytical or static-only approaches. This section reviews the previous analytical approaches for formulating parallel speedups and then claims that a dynamic approach is an attractive solution.

Presumably, Amdahl's law [6] is the first analytical model to estimate parallel speedup. If a programmer knows f, the fraction of perfectly parallelizable sections of a given serial program, then S, the ideal speedup on n processors, is formulated as follows:

$$S = \frac{1}{(1-f)+\frac{f}{n}}. \tag{1}$$

Amdahl's law provides a fundamental insight for parallelization: *The non-parallelizable part of a program eventually limits the parallel speedup.* However, Amdahl's law assumes two conditions:

- All threads execute an equal amount of work (i.e., homogeneous parallel workload); and
- No synchronization among threads exists.

Unfortunately, many parallel programs do not meet the above ideal assumptions. Amdahl's law itself is not an effective method to predict the speedups of realistic programs. There are many extensions of Amdahl's law such as Gustafson's law [36], Karp and Flat metric [66], a model for multicore chips [152], a model for asymmetric multiprocessors [44], and Eyerman and Eeckhout's work for modeling critical sections [29]. We discuss the last work.

Eyerman and Eeckhout recently proposed an extension of Amdahl's law by modeling critical sections with a simple probabilistic model [29]. They approximate the total execution time on a parallel machine based on the following assumptions:

- All threads execute for an equally long time;
- Each thread will execute an equal amount of critical section; and
- Critical sections are entered at random times and threads contend randomly to enter these sections.

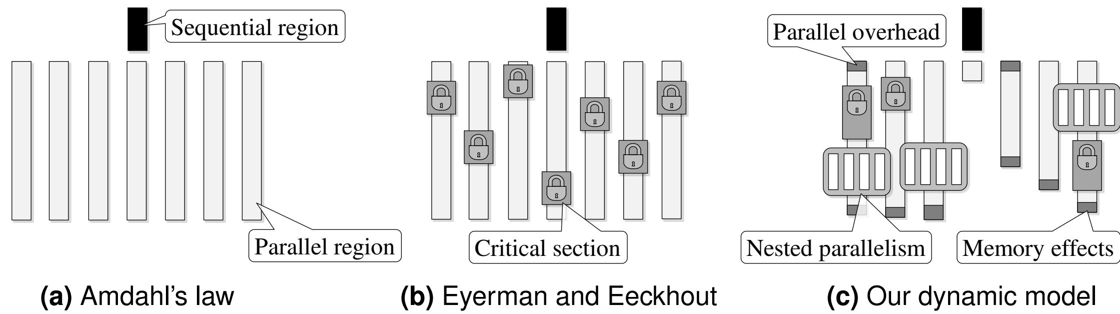
The critical section behavior is modeled with two scenarios: (1) On low-lock contention, the total execution time would be the average of the per-thread execution time; (2) on high-lock contention, the slowest thread will determine the execution time. The final formula is shown as follows (the two parts in the formula represent each case, respectively), where (a) f_{seq} is the totally sequential fraction, (b) $f_{par,ncs}$ is the purely parallelizable fraction that does not need any critical section, (c) $f_{par,cs}$ is the parallelizable fraction, but executed inside critical sections, (d) P_{ctn} is the probability for two critical sections to contend, and (e) P_{cs} is the probability for entering a critical section during parallel execution:

$$S = \frac{1}{f_{seq} + \max\left(f_{par,cs}P_{cs}P_{ctn} + \frac{f_{par,cs}(1-P_{cs}P_{ctn})+f_{par,ncs}}{n}; f_{par,cs}P_{ctn} + \frac{f_{par,cs}(1-P_{ctn})+f_{par,ncs}}{2n}\right)}.$$

This model proves a common understanding of programmers and computer architects: *Parallel speedup is both limited by the sequential part and by synchronization*. However, the assumptions of this extension may still be too restrictive to model realistic programs. The authors also clearly stated in their paper that the goal of the model was not to present accurate quantitative performance numbers and predictions [29].

Likewise, all simple analytical models with a few parameters are based on ideal assumptions. However, these assumptions are often broken. Many parallel programs do not have a homogeneous workload, and the length of a critical section may vary. Speedups may be heavily affected by scheduling policies and implementations of a specific parallel library, which cannot be easily modeled analytically. Furthermore, obtaining the inputs to the models is not straightforward. For example, the equation of Eyerman and Eeckhout's model requires challenging parameters such as P_{cs} and P_{ctn} that mostly need dynamic profiling. No doubt these concise models provide insight and a broader understanding for programmers and architects. However, they are not powerful to predict speedups from realistic serial programs.

As part of the continuous efforts to build a more powerful analytical model, a number of researchers have proposed more sophisticated analytical models to predict speedups practically including stochastic approaches (Refer to the related work in [2]). Among them, Adve and Vernon's work [2] is similar to one of our prediction algorithms, the Fast-Forward method (Section 6.3.1). Given a task graph (a sort of dependence graph), the task scheduling function (which defines how parallel tasks will be executed on processors), and other inputs, their analytical model performs a two-level analysis. The high-level analysis tracks the execution states of processors by traversing the task graph and evaluating the scheduling function. Although this two-level analysis itself is analytical, the model demands



(a) Amdahl's law **(b)** Eyerman and Eeckhout **(c)** Our dynamic model

Figure 10: Comparison of the simple analytical models that need a few parameters and our dynamic approach: our profiling and dynamic emulation model: (1) unequal thread execution time (load imbalance), (2) differences in scheduling policies, (3) multiple critical sections with arbitrary lengths and contentions, (4) nested parallelism, and (5) parallel overhead and effects of caches and memory.

a task graph that mostly requires dynamic analysis. Unfortunately, because obtaining the inputs to their model is a separate step and non-trivial, it is not easy for programmers to use this approach practically.

To address the fundamental weaknesses of these analytical models, we use a *dynamic* approach using low-overhead runtime *profiling* and *emulations* to predict parallel speedup. Figure 10 summarizes the comparison of our dynamic approach with the two static approaches. Our dynamic approach can model many realistic parallel programming patterns and characteristics as follows:

1. Arbitrary workload imbalance;

2. Differences in scheduling policies (e.g., a simple static scheduling like `omp schedule(static,1)` in OpenMP and an advanced dynamic scheduling such as work-stealing [11]);

3. Complex parallel patterns such as nested and recursive parallelism;

4. The overhead of operating system-level scheduling and synchronizations;

5. The overhead and detailed semantics of a parallel library (e.g., TBB [56] and Boost.Thread [12], MPI [35]) or parallel language extensions (e.g., OpenMP

[138], Cilk Plus [49]; Task Parallel Library [80], OpenCL [67]; and

6. Predicting negative effects on parallel speedups due to increased traffic of caches and memory.

Chapter IV introduces our dynamic approach for speedup prediction. Our predictor currently requires manually inserted *annotations* on a serial program. Annotations are used to specify potentially parallelizable and protected sections. Note that this approach using annotation, profiling, and emulation is first used in the Suitability analysis of Intel Parallel Advisor [51]. However, we contribute a new speedup prediction algorithm and a memory performance model to enhance the prediction ability. The following section discusses the motivation.

2.6 Motivations for A New Speedup Prediction Algorithm

A couple of recent works for the dynamic prediction of speedups were proposed: Cilkview [42], the Suitability analysis in Intel Parallel Advisor (denoted by Suitability) [51], and Kismet [62]. Among them, Suitability is the most closely related to our work, as Parallel Prophet also takes the same basic steps: annotation, profiling, and emulation. To the best of our knowledge, Suitability is the state-of-the-art technique in predicting speedups in helping the parallelization steps. The others focus on a different perspective, which is discussed in Chapter III.

As summarized in Figure 10 and the previous section, a dynamic approach resolves many problems in realistic speedup prediction. First, both our algorithm and Suitability predict well the basic patterns of Figure 11, which illustrates a single top-level parallel loop, arbitrary workload lengths, and a single lock[5] with arbitrary lengths and contentions.

[5]Current Suitability supports only a single global lock while Parallel Prophet supports multiple locks. We do not speculate that supporting multiple locks requires significant efforts.

```
1  for (int i = 0; i < N; ++i) {  // To be parallelize.
2    Compute(...);                // Computation length varies.
3    if (do_lock1)                //
4      Compute(...);              // To be protected by a mutex.
5    Compute(...);                //
6  }
```

Figure 11: Easy-to-predict parallel programming patterns by dynamic analysis: A single top-level parallel loop + workload imbalance + a single lock.

However, Suitability currently (as of 2012) is not effective in capturing other complex characteristics of programming patterns, various parallel overhead, and differences in parallel programming paradigms. After surveying OmpSCR [103, 24] and NPB [64], we present the cases that provide the motivation for our new prediction algorithm:

- *Scheduling policies*: In Figure 12(a), the amount of work for each iteration of the `for-i` varies. Scheduling policies can affect speedups significantly in this case. A good example is also shown in Figure 67, where speedups on two cores can be different by more than 20% due to OpenMP's scheduling policies. Speedup prediction should consider such policies. Section 2.7.1 summarizes three representative scheduling policies of OpenMP.

- *Inner loop parallelism*: Parallelizing an outer loop yields better speedups by minimizing spawning and joining overhead.[6] However, as shown in Figure 12(a), programmers are often forced to parallelize inner loops when an outer loop is not parallelizable. Predicting the cost and benefit of inner loop parallelism would be very informative to the programmer. Thus, precise modeling and estimation of the parallel overhead is necessary.

- *Nested and recursive parallelism*: Speedups of nested or recursive parallel

[6]In general, a parallel library or extension does not create physical threads on every parallel loops. Instead, it pre-creates threads and parks them. However, there are still overhead of dispatching logical tasks, deallocating the tasks, and the final implicit barrier.

```
1  for (k = 0; k < size - 1; k++) {
2    #pragma omp parallel for schedule(static,1)
3    for (i = k + 1; i < size; i++) {
4      L[i][k] = M[i][k] / M[k][k];
5      for (j = k + 1; j < size; j++)
6        M[i][j] = M[i][j] - L[i][k]*M[k][j];
7    }
8  }
```

(a) Workload imbalance and inner loop parallelism in *LUreduction*: The trip count of inner loops varies. Scheduling policies such as `(static,1)` affect the speedup.

```
1  void FFT(...)
2  {
3    ...
4    // Divide the task and run in parallel and recursively.
5    cilk_spawn FFT(D,   a,      W, n, strd/2, A);
6               FFT(D+n, a+strd, W, n, strd/2, A+n);
7    cilk_sync;
8
9    cilk_for (i = 0; i <= n - 1; i++) {
10       ...
11   }
12 }
```

(b) Recursive and nested parallelism in *FFT*: For efficient implementation of recursion, the original OpenMP-based (version 2.0) implementation is replaced by Cilk Plus [49].

Figure 12: Hard-to-predict parallel programming patterns in OmpSCR.

programs are not easy to predict because their runtime behavior heavily depends on a parallel library implementation. Figure 12(b) shows the recursive parallelism, which is also a form of nested parallelism. We verified that a naive implementation by OpenMP's (version 2.0) nested parallelism yields poor speedups. An experimental result is presented in Section 2.7.2. TBB, Cilk Plus, and OpenMP 3.0's `task` are much more effective in recursive parallelism. The differences in the implementations of parallel libraries should be modeled to predict speedup precisely.

- *Memory-limited Behavior*: Figure 13 (see "Real") shows an example in which the parallel performance of NPB FT does not scale. Our detailed profiling verifies that the increased memory traffic in parallel execution caused the

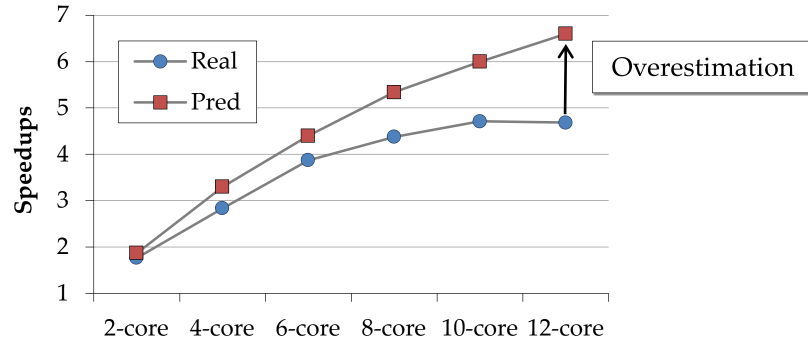


Figure 13: FT in NPB. Input set is 'B' of 850 MB memory footprint. Suitability overestimates speedups. Speedups on this machine are saturated due to increased memory traffic.

saturated speedups. On a higher memory bandwidth computer, the degree of speedup saturation was decreased compared to Figure 13. However, current speedup predictors, such as Suitability and Kismet, cannot model this kind of memory effects. The "Pred" in Figure 13, the prediction results of Suitability, is obviously overestimated. This experiment reveals that current Suitability lacks the ability of predicting parallel memory performance. This motivates us to build a model that predicts such speedup saturation.

From the above observations and motivations, Parallel Prophet proposes two new algorithms: (1) the *program synthesis-based emulation* algorithm (denoted by the synthesizer) to predict a more diverse and complex parallel programming pattern considering the effects of scheduling (both operating systems and parallel libraries), parallel overhead, and differences in parallel libraries; (2) a *memory performance model* to predict saturated speedups based on an analytical model that trained by our microbenchmarks. Chapter VI presents Parallel Prophet.

2.7 Background on Scheduling Policies of OpenMP and Cilk Plus

This section provides brief background knowledge on scheduling policies of OpenMP and Cilk Plus, which is needed to develop Parallel Prophet.

Table 3: Scheduling policies in OpenMP parallel-for construct

Schedule kind	Description
static	Iterations are divided into chunks of a size of chunk_size. The chunks are statically assigned to threads in a round-robin fashion. When no chunk_size is specified, the iteration space is divided into chunks that are approximately equal in size.
dynamic	Iterations are divided into chunks of a size of chunk_size. Each chunk is assigned to a thread that is waiting for an assignment. The thread executes the chunk of iterations and then waits for its next assignment, until no chunks remain to be assigned.

```
1  #pragma omp parallel for schedule(kind, chunk_size)
2  for (int i = 0; i < N; i += step) {
3    work(...);
4  }
```

Figure 14: OpenMP parallel for with schedule clause.

2.7.1 Scheduling Policies in OpenMP

We recap the scheduling policies of OpenMP for which we evaluate in Chapter VI. Although four kinds of scheduling are defined in OpenMP's `schedule` clause (`static`, `dynamic`, `runtime`, and `guided`), we focus on `static` and `dynamic` with different chunk size. Figure 14 is a typical OpenMP's parallel-for code snippet with an explicit `schedule` clause that takes `kind` and an optional `chunk_size`. Table 3 summarizes the description of the OpenMP clause and parameters [104]. In this thesis, we evaluate three kinds of scheduling: (1) `schedule(static,1)`, (2) `schedule(static)`, and (3) `schedule(dynamic,1)`. We may omit `schedule` for simplicity.

2.7.2 Recursive and Nested Parallelism in OpenMP and Cilk Plus

One of the motivation of the synthesizer is to support a complex nested parallelism pattern as shown in Figure 12(b). We demonstrate why this pattern is important in parallel programming with a simple QuickSort program.

```
1  #include <cilk/cilk.h>
2  #include <algorithm>
3  #include <functional>
4
5  void qsort_cilk(int* begin, int* end)
6  {
7    if (begin != end) {
8      // The last item is the pivot.
9      --end;
10     // Do partitioning.
11     int* middle = std::partition(begin, end,
12       std::bind2nd(std::less<int>(), *end));
13     // Pivot is in the middle.
14     std::swap(*end, *middle);
15
16     // Recursively sort two sub arrays divided by the pivot.
17     cilk_spawn qsort(begin, middle);
18     qsort(++middle, ++end);
19
20     cilk_sync;
21   }
22 }
23
24 // Assumes that the array A and N are given.
25 qsort_cilk(A, A + N);
```

(a) A parallelized QuickSort by OpenMP's nested parallelism

```
1  void qsort_omp_recursive(int* begin, int* end)
2  {
3    if (begin != end) {
4      ...
5      if (get_total_thread_num() > THRESHOLD) {
6        // If the number of threads is greater than a threshold
7        // (e.g., physical core number), stop the parallel execution.
8        qsort_omp_recursive(begin, middle);
9        qsort_omp_recursive(++middle, ++end);
10     } else {
11 #pragma omp parallel sections /*nowait*/
12       {
13 #pragma omp section
14         qsort_omp_recursive(begin, middle);
15 #pragma omp section
16         qsort_omp_recursive(++middle, ++end);
17       }
18     }
19   }
20 }
```

(b) A parallelized QuickSort by Cilk Plus

Figure 15: Parallelized QuickSort by Cilk Plus and OpenMP: (a) `cilk_spawn` and `cilk_sync` are the Cilk Plus' keywords. If these keywords are deleted, the code is a valid serial QuickSort program. Using OpenMP 3.0's `task` is similar to this version. (b) Notice a throttling code to control the number of threads. `omp_set_nested(1)` must be called to enable nested parallelism.

Figure 15 contains two parallel implementations of QuickSort using Cilk Plus [49] and OpenMP [138]. Note that the serial code of QuickSort can be obtained by removing Cilk Plus keywords (`cilk_spawn` and `cilk_sync`), which is also known as *serial elision* [31]. The Cilk Plus implementation simply inserts `cilk_spawn` at the point of the parallelism. This keyword overrides a function call statement to tell the runtime system that the function may run in parallel with the caller [49]. The main thread continues to call line 18 at Figure 15(a), but a logical parallel task performs line 17. The runtime decides whether both calls at line 17 ad line 18 can run in parallel. When `qsort_cilk` is recursively called, the number of logical task can be exponentially increased. However, the runtime, which implements a work-stealing dynamic scheduler [11], efficiently maps the spawned tasks to cores.

An implementation using traditional OpenMP may not be as efficient as Cilk Plus. Figure 15(b) uses OpenMP's nested parallelism. Most OpenMP implementations disable nested parallelism by default. One must enable by either calling `omp_set_nested(1)` or setting `OMP_NESTED` environmental variable. However, OpenMP's `parallel` construct physically creates multiple threads specified by `omp_num_threads()` and `OMP_NUM_THREADS`. This behavior could easily bloat the number of threads, resulting in slowdown. The code at Figure 15(b) contains a throttling code to limit the number of total physical threads. However, OpenMP 3.0 introduces a new `task` construct, which is similar to `cilk_spawn`. Both OpenMP's new `task` and `cilk_spawn` are ideal to parallelize irregular problems including recursive algorithms and unbounded loops.

Figure 16 shows an experimental result of four different implementations: (1) a Cilk Plus version as shown in Figure 15(a), (2) an OpenMP's nested parallel version as shown in Figure 15(b), (3) an OpenMP 3.0's `task` version, and (4) a parallelized version of an iterative QuickSort algorithm using traditional OpenMP.

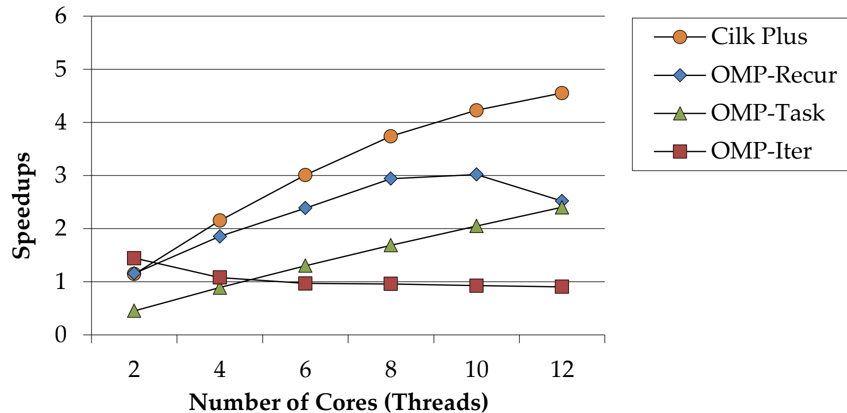


Figure 16: Speedups of various parallelization versions of QuickSort: A simple quick sort program is parallelized by (1) Cilk Plus, (2) OpenMP with recursive parallelism, (3) OpenMP 3.0 with `task` parallelism, and (4) OpenMP with an iterative algorithm.

The difference of the speedups are quite significant: while "Cilk Plus" and "OMP-Task" show scalability, the performance of the other implementations decrease as the core number increases.

2.8 Motivations for Post-Analyzer of Dependence Profiler

A profiling result of SD^3, Prospector's loop and dependence profiler, is a list of discovered data-dependence pairs in either a loop or a function. This *raw* result does not directly give hints on parallelism and code transformation. Programmers manually have to parse a raw result, which is sometimes impossible.

A result of an embarrassingly parallelizable loop may be straightforward: no data-dependence pair would be reported. However, even for this case, the raw result could have data dependences from (1) loop induction variables (although they could be easily filtered at the instrumentation or static analysis phases) and (2) static and global variables that need appropriate privatization. Programmers need to parse such dependences to write a parallel program.

Figure 17 shows an example of a raw result of the MPEG2 decoder in

```
=================================================================================
Loop name: slice:168@1
Location: MyMediabench\mpeg2\src\mpeg2dec/getpic.cpp(1minjang68)
Parent loop: <invalid>, Parent func: slice
Insts  =      7865588  AvgInst =       327732  Coverage = 93.71%
Invocs =           24  TotTrip =          192  (Min: 8, Max: 8)
# of conflicted pairs: 1044
+-------+--------------------------------+--------------------------------+------+
|  Type |             Source             |              Sink              | Freq |
+-------+--------------------------------+--------------------------------+------+
| liRAW | (R,  ld,  142, 2, Flush_Buffer) | (W,  ld,  194, 3,        slice) | 8309 |
| lcRAW | (R,  ld,  142, 2, Flush_Buffer) | (W,  ld,  142, 2, Flush_Buffer) |  168 |
|  ...  |               ...              |               ...              |  ..  |
| lcWAW | (W, tab,  464, 4, Decode_MPE..) | (W, tab,  464, 4, Decode_MPE..) |   26 |
|  ...  |               ...              |               ...              |  ..  |
| lcWAW | (W, PMV, 1163, 3, decode_mac..) | (W, PMV, 1174, 3, decode_mac..) |    5 |
|  ...  |               ...              |               ...              |  ..  |
+-------+--------------------------------+--------------------------------+------+
```

Figure 17: An example of a raw result of SD[3]: The MPEG2 decoder in MediaBench [79]: This raw result also includes the result of loop profiling. All discovered pairs of data dependences are reported. Legend: (1) li{RAW|WAW|WAR}: loop-independent dependences, (2) lc{RAW|WAW|WAR}: loop-carried dependences, (3) Source/Sink: (RW, variable name, line, column, function).

MediaBench [79]. The results include details of found dependences (data-dependence distances are not shown) and loop profiling. This MPEG2 is known to be parallelizable by pipelining [139, 140, 124]. However, the number of the reported dependence pairs are more than a thousand.[7] It is almost impossible for programmers to extract the hidden pipeline parallelism by hand. Prospector should provide a post-analysis algorithm to extract parallelism from the raw results, which in turn gives hints on code transformation. Specifically speaking, this post-analysis algorithm provides a way to avoid the discovered dependences for parallelization.

In particular, we investigate the following important techniques to avoid a data-dependence pattern: (1) privatization, (2) reduction, (3) a simple mutex, (4) barriers, and (5) pipelining. For example, if discovered dependences are only output or anti-dependences from non-auto variables, privatization would be enough to avoid these dependences. The other patterns require more studies to

[7] The current instrumentation logic conservatively instruments loads and stores. The result may have unimportant dependences that do not prevent parallelization.

understand the correct patterns and behaviors. When such hidden parallelism is exposed, the next step is to provide hints on code transformation, which can be highly informative for both programmers and potential compiler optimizations.

This post-analysis finally enables the integration of Parallel Prophet and SD^3. Parallel Prophet requires annotations that are manually inserted by programmers. However, the post-analyzer algorithm can provide an assisted annotation process. The post-analyzer completes Prospector so that holistic advice for the parallelization steps will significantly reduce the burdens of the parallelization.

2.9 *Overview of Prospector*

Figure 18 sketches an overview of Prospector: (1) an efficient loop profiler, (2) data-dependence profiler, (3) speedup predictor, and (4) post-analyzer.

Prospector takes a serial program to be parallelized, in either binary or source code and performs instrumentations with some static analysis. For executables,

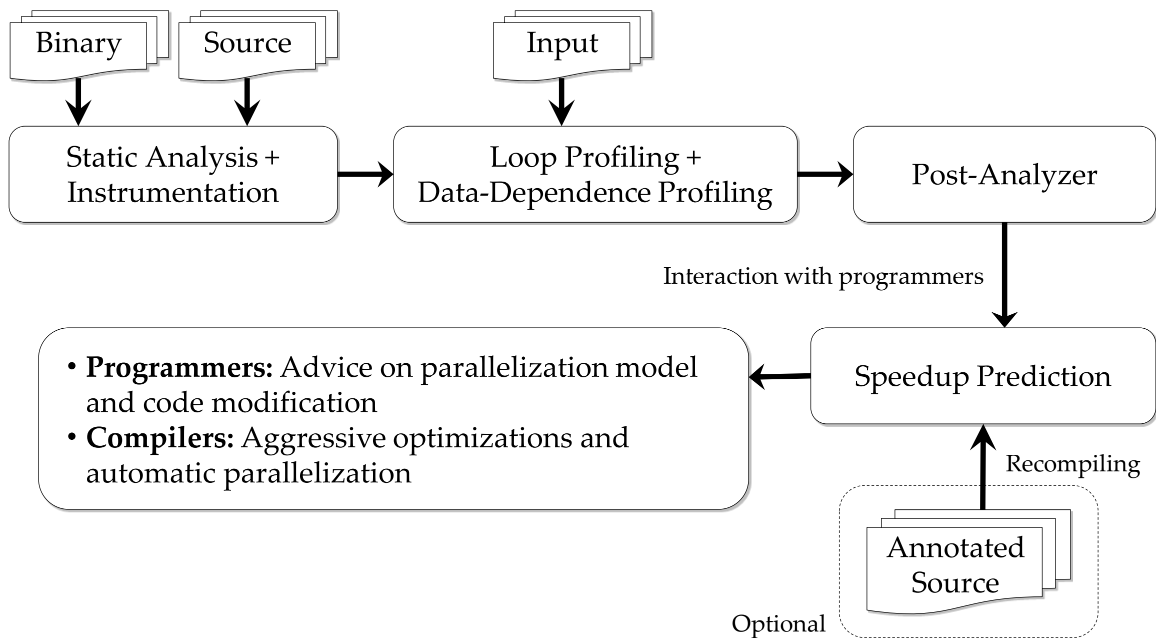


Figure 18: Overview of Prospector.

Prospector uses Pin [85] for the instrumentation. LLVM [77] is used for source-level instrumentation and static analysis. Except for the overhead experiments of SD^3 in Chapter V, we use LLVM-based Prospector for the other studies.

The SD^3 component efficiently performs loop profiling and data-dependence profiling together. Recall that the dependence profiler is built upon the loop profiler. Efficient loop profiling is achieved by static loop-structure discovery and minimal instrumentation. In order to minimize the overhead of data-dependence profiling, SD^3 performs compression and parallelization.

The post-analyzer processes raw results of SD^3. The focus of this step is to find feasible ways to avoid discovered data dependences that prevent parallelization. Because there is no feasible solution to avoid an arbitrary data-dependence pair, we attack this problem by its patterns such as reductions and pipelining.

Parallel Prophet, the speedup predictor of Prospector, profiles an annotated serial program in very low overhead and emulates the expected parallel behavior for a given parallel programming paradigm, the number of threads, and a particular scheduling policy. Parallel Prophet also employs a memory performance model to predict saturated speedup for a certain case.

The final assorted results will help both programmers and potential compiler optimizations. In this thesis, we focus on the former: supporting programmers in writing parallel code. Our suggested parallelization steps, introduced in Section 1.1, will be much easier than an approach without tools and advanced analyses.

CHAPTER III

RELATED WORK

This chapter summarizes the related researches to the components of Prospector: loop profiling, data-dependence profiling, dynamic speedup prediction, and post-analysis algorithm. Finally, an example of state-of-the-art software support is presented and compared to Prospector.

3.1 Related Research on Loop Profiling

Intel Parallel Advisor [51] provides some loop-level profiling, but a profiling result is a simple extension of traditional function-level profiling. A loop is simply considered as a function. Execution time in a loop is the only reported information. As argued in Section 2.1, detailed loop profiling is informative, such as average trip count, invocation count, and statistics of iteration lengths. Moseley et al. [94] proposed LoopProf for this need. LoopProf, which is implemented on Pin [85], takes an x86 binary. First, LoopProf instruments the beginning of basic blocks to capture the execution of basic blocks. During the runtime, executed basic blocks are stored in a stack. If a duplicated basic block is detected in the stack, LoopProf concludes that a loop has been detected and started. Although this approach is simple to implement and robust to many exceptional cases in x86 binary, its time overhead is fairly high on average 24 times slowdown. Because the loop profiler becomes the platform of the data-dependence profiler of Prospector, the overhead should be minimized.

3.2 *Related Research on Data-Dependence Profiling*

One of the early works that used dynamic data-dependence analysis is Petersen and Padua's work [108]. They used dynamic dependence analysis to evaluate the effectiveness of static dependence tests. Blume et al. used a dynamic test called PD test to detect dependence dynamically in a compiler [10]. Similarly, Rauchwerger and Padua used a runtime data-dependence analysis in their LRPD test. These three works, however, did not provide detailed algorithms and overhead data. Larus proposed a parallelism analyzer pp [75] with detailed descriptions. pp dynamically detects loop-carried dependences in a program. Larus analyzed six programs and showed that two different groups, numeric and symbolic programs, had different dependence behaviors. The profiling algorithm dynamically computes loop-carried dependences in loop nest, which is very close to our pairwise method. pp had severe memory and time overhead, but reducing the overhead was not in the scope of the work. To avoid excessive overhead, pp only records a single read of a location. As a result, pp cannot detect all kinds of anti-dependences.

Tournavitis et al. [141] proposed a dependence profiling mechanism to overcome the limitations of automatic parallelization. Similarly to pp, their dependence profiling algorithm is similar to our pairwise method, and they did not discuss the overhead problem. Zhang et al. [156] proposed a data-dependence-distance profiler called Alchemist. Their tool is specifically designed for the *future* language constructor that represents the result of an asynchronous computation [32]. Although they claimed there was no memory limitation in their algorithm, they evaluated very small benchmarks. The number of executed instructions of a typical SPEC 2006 reference run is on the order of 10^{12}, while that of the evaluated programs in Alchemist is less than 10^{8}. von Praun et al. [148] proposed a notion of dependence density: the probability of existing memory-level dependencies

among any two randomly chosen tasks from the same program phase. They also implemented a dependence profiler by using Pin. The author informed us that the runtime overhead was similar to our pairwise method.

The above works used the data-dependence profiling, but no work explicitly handles the overhead of the data-dependence profiling. To the best of our knowledge, we have presented the first efficient algorithm in the data-dependence profiling domain. Ramaseshan briefly sketched this overhead problem in his master's thesis [115], but did not provide a concrete solution.

After our SD^3 work, Yu and Li recently have proposed an algorithm, *multi-slicing* [153], which attacks the overhead problem. One of the key ideas of multi-slicing shares the same insight with our DAS-ID optimization presented in Section 5.3.5. DAS-ID optimization is to reduce dependence checking space by comparing only memory references from the *same* dynamic-allocation site. Similarly, multi-slicing partitions a program into many *independent* slices. The partitioning can be done in various granularity via compiler-time analysis and runtime techniques. For example, they divide a program into disjointed *alias classes* by a compiler's interprocedural pointer analysis. Because each slice accesses a disjointed memory space, different slices can be profiled in parallel. The idea is novel, and we believe SD^3 can also employ the algorithm of multi-slicing. The performance improvement is also impressive, but their baseline implementation is too slow. From our experience in building the pairwise method, our first implementation sometimes showed a 5,000+ times slowdown. However, after applying important implementation techniques, notably PC-set optimization (Section 5.2.6) and other programming techniques (Section 5.5.2), the overhead was significantly reduced as low as a couple of hundreds slowdown.

3.2.1 Data-Dependence Profiling for Speculative Parallelization

The concept of dependence profiling already has been used for speculative hardware based optimizations. TLS (thread-level speculation) compilers speculatively parallelize code sections that do not have many data dependences. Several methods have been proposed [136, 16, 25, 82, 151], and many of them employ sampling or allow aliasing to reduce overhead. All of these approaches do not have to give correct results like SD^3 since speculative hardware would solve violations in memory accesses.

Among these works, the dependence profiler by Chen et al. [16] is close to our baseline pairwise method. To save space and time, they use a shadow table. On a memory access, they map this reference to the corresponding entry of the shadow table via a simple hash function. Although the overhead is lower than our profiling, it suffers from false positives. The mechanism also employs an advanced sampling technique other than a random sampling, but this sampling costs the accuracy of the profiling.

3.2.2 Reducing Time Overhead of Dynamic Analysis

Other than sampling techniques, a number of previous works exploit parallelism in dynamic analyses to reduce time overhead. Shadow Profiling [95], SuperPin [149], PiPA [157], Speck [101], and Ha et al. [39] employed parallelization to reduce the time overhead of instrumentation-based dynamic analyses. Since all of them focus on a generalized framework, they exploit only task-level parallelism by separating instrumentation and dynamic analysis. The parallelization of SD^3 is a specialized approach in that it should save the time overhead at the same time. However, there are many design and implementation challenges for the effective parallelization. First, our memory-efficient profiling needed to be revised for parallelization as presented in Section 5.4.4. Second, data-level and pipeline

parallelism are exploited, but we should address additional design issues to achieve higher speedups, which is discussed in Section 5.4.5.

3.2.3 Limitations of Previous Compression Techniques

A number of techniques to decrease dynamic profiling memory overhead have been proposed. The root cause of the memory overhead in dynamic profiling is the amount of dynamic execution history, or dynamic trace. Figure 55 demonstrates an example of extremely large dynamic trace size at an order of the terabyte.

Many previous works such as Whole Program Paths [76], Whole Execution Traces [155], METRIC [86], and ParaMeter [113], addressed the space overhead problem. Their common approach is using specialized data structures and compression algorithms for instructions and memory accesses of specific patterns. Larus used SEQUITUR [99] algorithm to compress path trace in whole program path profiling [76]. METRIC [86], a tool to find cache bottlenecks, introduces PRSD (power regular section descriptors) compression technique to efficiently capture regular memory streams, which is comparable to our stride-based compression.

However, these compression techniques aim on only compressing dynamic execution traces itself. With such techniques, we have to uncompress traces to calculate data dependences. For example, METRIC decompresses traces before sending the offline internal analyzer. If we were use this approach, most of profiling time would have been spent on compressing and decompressing traces. SD^3 is fundamentally different from the previous compression techniques; SD^3 computes data dependence directly with the compressed stream.

3.3 Related Research on Dynamic Speedup Prediction

We compare compares Parallel Prophet to three recent dynamic speedup prediction tools: Kismet, Suitability, and Cilkview, in Table 4.

Table 4: Comparison of the recent dynamic speedup prediction tools. ◯: Predicts well for the experimentation in this chapter; △: Predictions are limited; ×: Not explicitly modeled; Legend for patterns: (1) P1: Simple loops/locks; (2) P2: Workload imbalance; (3) P3: Inner-loop parallelism; (4) P4: Recursive parallelism; and (5) P5: Memory limited behavior.

Name	Input to the Profiler	Parallel Program Patterns					Overhead
		P1	P2	P3	P4	P5	
Cilkview [42]	*Parallelized* code	◯	◯	◯	◯	×	Moderate
Kismet [62]	Unmodified *serial* code	◯	△	△	△	△ (Only superlinear)	Large
Suitability [51]	Annotated *serial* code	◯	△	△	△	×	Small
Parallel Prophet	Annotated *serial* code	◯	◯	◯	◯	△ (Only BW contention)	Small

Kismet [62] is a profiler that provides estimated speedups for a given unmodified serial program. Kismet is different from Parallel Prophet in that no annotation is needed. Kismet performs an extended version of hierarchical critical path analysis [33] that calculates self-parallelism for each dynamic region and then finally reports estimated overall speedups by summarizing all region profiles. However, the overhead of Kismet typically shows 100+ slowdowns because memory instructions are instrumented to perform a heavy analysis.

The Suitability analysis in Intel Parallel Advisor [51] estimates speedups of a program for a couple of threading paradigms and CPU numbers. An input program must be serial code, but requires annotation to specify parallel and protected sections. Suitability then collects timing information in the runtime to build a model of the program's dynamic parallel-region tree. Speedup estimation is done by emulating expected parallel behaviors. The emulation of the model is done by an interpreter that uses a priority queue to fast forward a pseudo-clock to the next event. The emulation includes some details specific to a threading paradigm - for example, how long it takes to acquire a lock including contention - but does not consider memory interactions or exact choices of task to run.

Cilkview [42] is different from Kismet, Suitability, and Parallel Prophet, because an input program should be already parallelized by Cilk Plus [49]. The purpose of

Cilkview is not to predict speedups from a serial code. Rather, Cilkview is a tool that visualizes the scalability of a parallelized program by Cilk Plus.

Suitability is the closest approach to ours and shares the basic methodology. However, our experimentation reveals that Suitability is currently limited to support various parallelism models and predict memory behavior.

3.4 Related Research on Post Analysis of Dependence Profiling

3.4.1 Parallelism Discovery Using Dynamic Analyses

Thies et al. proposed a method that verifies the correctness of a given pipeline configuration [139]. Programmers manually insert annotations to specify the potential pipeline stages. The annotated serial code is then dynamically profiled to verify the data dependences between the pipeline stages. Although the annotation step is a programmers' responsibility, this work motivates the adaption of the data-dependence profiler to facilitate such parallelism discovery process. Several researchers extended this work to assist the manual annotation step using data-dependence profiling: Tournavitis et al. [140], Paralax [144], and Rul et al. [124]. These work have a lot of similarity in the approaches. First, a data-dependence graph or program dependence graph is built, then a graph algorithm finds potential pipeline stages. Finally, parallelized code is generated.

Alchemist [156] finds concurrency in a serial C program by profiling types and distances of data dependences. Alchemist is, however, a more specialized tool to discover program sections that can be called by an asynchronous construct such as `future` in Java [32].

Kremlin [33] measures the optimistic potential parallelism in serial code using an extended critical path analysis (CPA), called hierarchical CPA. The hierarchical CAP provides parallelism quantity for a specific program region such as a loop nest. Based on this profiling result, Kremlin creates a parallelization plan to select

best regions for a target parallel machine considering constraints. An example of OpenMP using DOALL and DOACROSS are presented. The approach of Kremlin is complement to our data-dependence-based approach because it can provide more optimistic and upper bound of the inherent parallelism. However, in order to provide more sophisticated code transformation information such as privatization and pipelining, we believe that dependence information should be needed.

ALTER [143] proposed a system to relax some dependences while mostly preserving the original semantics of a program. Based on programmer annotations, ALTER is based on an observation that some dependences accidental artifacts of the implementation that will not prevent real parallelization. In particular, the system allows reordering of iterations or stale reads to expose hidden parallelism.

Knobe and Sarkar proposed a SSA (static single assignment) form for arrays to expose more parallelism opportunity [72]. Although this is a static compiler analysis, the idea can be applicable to our dynamic analysis. For example, dependences from an array can be further analyzed in element granularity. Some sections of an array then can be renamed to expose more parallelism.

3.4.2 Code Transformation to Avoid Dependences

The essential part of the post-analyzer is providing valuable information to modify code to avoid and remove dependences for parallelization. This problem can be seen as a *bug-fixing* problem if a data dependence is considered as a problem. Our data-dependence profiling is then a *bug-finding* problem. In general, devising a solution for an automated bug fix is much harder than finding bugs.

To the best of our knowledge, we have not found a research work or a commercial product that provide generalized solutions on writing or patching code to avoid data dependences for the parallelization purpose, especially for true dependences. All of the previous works are specialized for a particular

type of dependences. For example, many researchers addressed the problems of privatization and reduction for parallelization including Tu and Padua's work [142] and the LRPD test [117]. Recently, Privateer [65] is proposed for more efficient automatic privatization with speculative memory separation. Pipelining is also practically an important domain. Tournavitis et al. [140] and Rul et al. [124] implemented algorithms to generate pipelined parallel code automatically in IR (intermediate representation) level. Their works focused on coarse-grained pipeline parallelism in popular streaming applications such as bzip and mpeg. Decoupled Software Pipelining (DSWP) is a novel technique that exploits fine-grained pipeline parallelism in general-purpose applications [116]. An automatic algorithm that extracts DSWP is also introduced [106]. The idea is tested on a cycle-accurate simulator that implemented the synchronization array, a set of low-latency queues in hardware. All mentioned efforts are related to our post-analyzer in that their approaches eventually avoid certain types of data dependences.

Behavior Oriented Parallelization (BOP) is a software-based speculative parallelization technique [23]. A parallelism analyzer (based on instrumentation) profiles based on value-based checking (rather than data dependence) the program execution and identifies potentially parallel regions (PPR). These PPRs are executed in parallel with a virtual-memory-protection based software speculation. This approach allows an incremental parallelizing by removing dependences one by one during the runtime profiling. This thesis, however, does not assume such speculation techniques.

3.4.3 Bug-Fixing Algorithms in Concurrent Programming

We present two recent bug-fixing algorithms in concurrent programming. Concurrency bugs (e.g., data races and atomicity violation) share a similar aspect to data dependences in that both of them are from interactions of two (or more) tasks.

Hence, ideas in bug-fixing algorithms for concurrent bugs could be valuable in finding solutions to avoid data dependences in our post-analyzer.

AFix [63] is an interesting work that suggests a fixing solution for atomicity violations [83]. Atomicity violations are one of the concurrency bugs along with deadlock, data races, and ordering violation. Suppose two or more tasks should be executed as if they are atomic. However, for example, if a lock is omitted or the scope of the lock is wrong, the intended atomicity by a programmer will be broken. This work proposes an automated solution to fix such atomicity violations. Because their work is for already parallelized program, their work is not directly applicable to the domain of assisting parallelization. However, a set of flow data dependences whose computation order does not matter can be avoided by a mutex or a reduction. The idea of AFix could be extended to suggest a parallelization advice on a particular pattern of data dependences.

Volos et al. [147] showed that software transactional memory (STM) was able to fix 43 of the 60 concurrency bugs they experimented. This thesis does not consider TM as a fixing solution, remaining as a future work. The work of Volos et al. implies that Prospector could provide hints on code transformation using TM to avoid data dependences toward parallelization.

3.4.4 Parallelism Visualization

Visualizing data dependences is an effective tool that gives a summarized view of a given serial program. ParaMeter [113] visualized data dependences from a serial program to show potential data dependences. To visualize, ParaMeter uses DINxRDY [110], which plots parallel structures and exposes potential thread-level parallelism (TLP). DIN and RDY stand for Dynamic Instruction Number and Ready-time, respectively. A particular pattern in a plot may suggest a different type of parallelism. For example, diagonal lines that have overlapping

x-extents imply code sections that may have TLP. This plot visualizes the dynamic information; thus the size of the plot could be extremely large that requires an additional compression technique.

3.5 Tools for Assisting Parallelization

We first present two to assist parallelization from academia. We then survey the latest commercial tools to assist parallelization: Intel Parallel Studio [54], Vector Fabrics' Pareon [145], and Rogue Waves ThreadSpotter [121].

MAPS [14] is an integrated framework for application parallelization targeting multiprocessor system-on-chip (MPSoC) and a parallel model called tightly-coupled thread (TCT) model. MAPS takes a sequential C code and a specification of MPSoc as inputs, and then performs both static and dynamic analyses that produce weighted statement control data flow graphs (WSCDFGs). Instead of using basic block, MAPS exploits WSCDFGs to easily capture coarse-grained parallelism. The MAPS partitioning tools then extract task parallelism and generate parallelized C code for the TCT platform.

HPCTOOLKIT [1] is a set of tools that perform profiling, analysis, and presentation of performance of serial and parallel applications to provide information such as scalability bottleneck and resource contention. HPCTOOLKIT takes optimized binaries for language independence and uses statistical profiling and hardware performance counters for low overhead. This suite also provides interactive tools such as 'hpcviewer' and 'hpctraceviewer' to visualize program performance and pinpoint problematic spots. This suite focuses on the parallel performance analysis that can even diagnose the scalability problem for a 1024-core Cray system. This tool is, however, more close to a tool for post-parallelization steps.

3.5.1 Intel Parallel Advisor

Intel released Parallel Studio Suite [54] in 2010 for C/C++ developers. Parallel Studio is composed of several components such as Parallel Amplifier, Parallel Inspector, and Parallel Compose. Among these components, we focus on *Parallel Advisor*, a tool to guide manual parallel programming. We summarize its approach with the example in Figure 19.

Suppose a programmer wants to parallelize a 'sudoku' program that generates a hundred sets of Sudoku puzzle and its solution. First, the programmer finds that the for-loop at line 128 is the hottest spot by the profiler. Second, the programmer analyzes data dependences of the loop. Parallel Advisor requires annotations - like the suitability analysis - by programmers. A programmer manually inserts macros (`ANNOTATE_SITE_*`) to specify where Parallel Advisor will perform data dependence analysis. `SITE` defines a container where parallel tasks are running in parallel and an implicit barrier at the end. Another set of macros (`ANNOTE_TASK_*`) are used to indicate the boundary of a parallel task in the parent `SITE`. For example, an

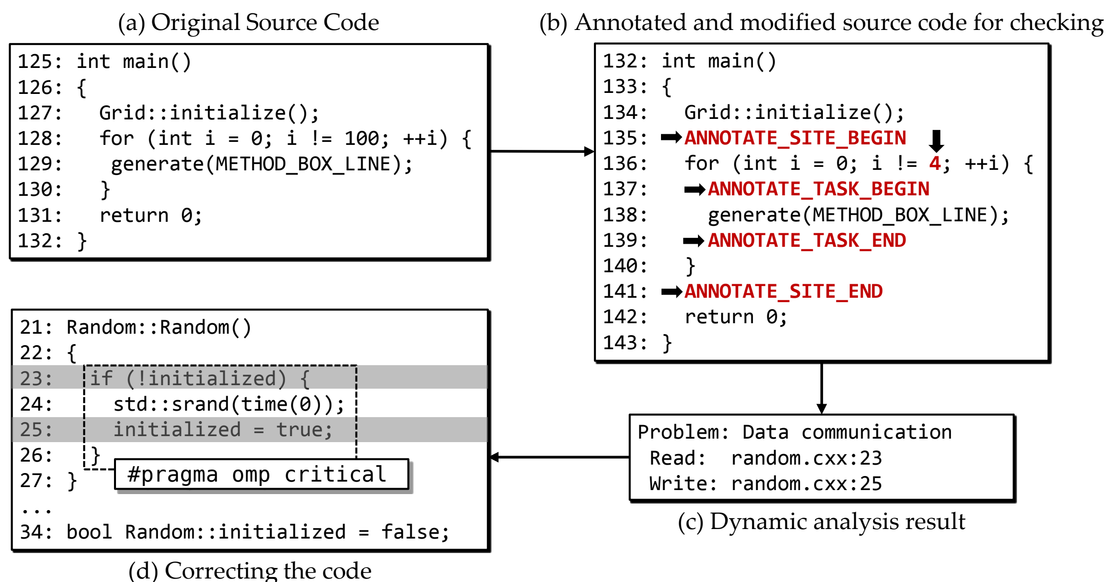


Figure 19: Overview of Intel Parallel Advisor's approach: This figure shows how a simple program is instrumented, analyzed, and finally corrected.

iteration of a loop will be a task, and a site encloses the loop, as shown in Figure 19(b). The third step is a dynamic analysis based on the annotations. We obtain results as shown in Figure 19(c). The programmer sees a data communication (i.e., read-after-write and write-after-read dependences) on lines 23 and 25 at `Random` function. To avoid this dependence, for example, the programmer now uses a critical section of OpenMP.[1] Finally, as a separate step, Suitability predicts an expected speedup. Parallel Advisor does not give advice on how to avoid found dependences. Programmers may repeat the second and third steps to correct dependences.

The approach of Parallel Advisor shares many aspect of Prospector. However, Prospector enhances key algorithms and introduces new analysis techniques. Loop profiling in Parallel Advisor only shows the total execution time, not detailed loop execution. Data-dependence profiling is limited for only small programs and inputs. Dependence distances and loop-carried information cannot be obtained as illustrated in Figure 31. Suitability currently cannot predict well on patterns shown in Figure 12 and Figure 13. No post-analysis is currently available in Parallel Advisor that Prospector provides.

3.5.2 Vector Fabrics' Pareon

Pareon [145] is a tool that assist parallelization steps from serial code. The basic approaches are also similar to Prospector and Parallel Advisor. However, several distinctions can be found in Pareon.

Figure 21 is a screen capture of Pareon that shows an interactive graphical user interface to visualize profiling information. For example, loop-carried dependences are visualized to make it more readable to programmers. Parallel Advisor is based on Pin, working directly on an x86 or x86-64 binaries. Pareon's

[1] In practice, the data dependence in `random` can be avoided by using a thread-safe random number generator.

Figure 20: Vector Fabrics Pareon (formerly vfThreaded x86): A graphical user interface to show loop-carried dependences.

methodology is intriguing: a programmer uploads to a server where Pareon is deployed. Pareon then compiles and instruments the program and performs several analyses including speedup prediction. For dependence profiling, their approach is similar to our pairwise method (Section 5.2). An engineer from Vector Fabrics provided us some profiling overhead as shown in Figure 6(c). The tool is not on a client computer; thus investigating this tool in depth is quite limited. We currently cannot speculate the mechanism of the speedup prediction. The contrasts over Parallel Advisor is that they consider both ARM and x86 platforms and provides a limited form of post analysis such as streaming pattern and compute dependency.

3.5.3 Rogue Wave's ThreadSpotter

Rogue Wave's ThreadSpotter [121] focuses on memory and cache performance for scalable parallel performance. This profiler provides memory bandwidth and latency, data locality, and thread communication and interaction (e.g., to diagnose false sharing). This tool also employs cache performance prediction, which is one of our future work for Parallel Prophet's memory performance model. However, this is not a speedup prediction tool like Parallel Prophet.

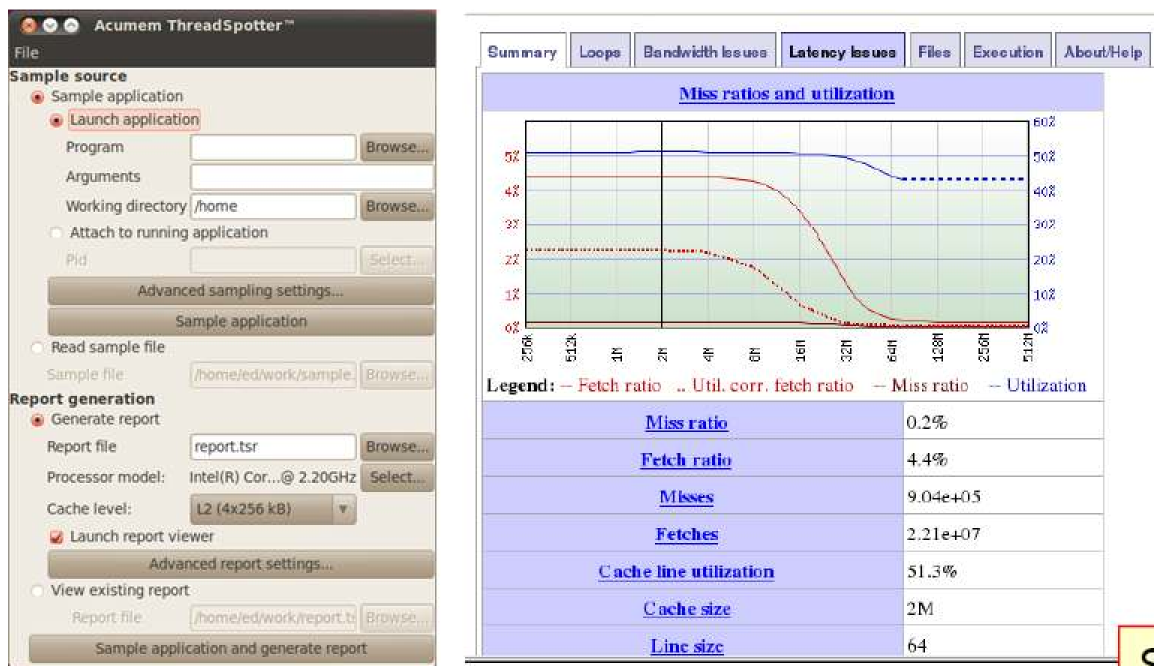


Figure 21: A screen capture of Rogue Wave's ThreadSpotter (formerly Acumem): The left figure shows an interface to launch the tool, and the right figure is an example of the memory latency issues.

CHAPTER IV

AN EFFICIENT LOOP PROFILER

4.1 Introduction

This chapter introduces an efficient loop profiler that will be the platform of the data-dependence profiler of Prospector. We implement both binary-level and source-level loop profilers and discuss the challenges and our experiences.

An example of our binary-level loop profiling result is shown in Figure 22. Loop profiling collects details of the loop execution including (1) the number of invocation; (2) the number of total trip counts with some statistics such as minimum, maximum, and average; (3) execution time and total instruction counts; and (4) loop hierarchy. Due to binary-level approach, some hex addresses are exposed at raw level. Debugging information enables mapping to the source code unless the code is aggressively optimized.

Our goal is to build a low-overhead and robust loop profiler while providing

```
------------------------------------------------------------------------------------------
main:331<1400014dd>, main<140001220>, spec.c, Parent: N/A
Insts  = 131,229,865,696  AvgInst = 131,229,865,696  Cover = 100%
Invocs =               1  TotTrip =               3  (Min: 3, Max: 3)
------------------------------------------------------------------------------------------
compressStream:462<140005e40>, compressStream<140005da0>, bzip2.c, Parent: N/A
Insts  = 104,280,265,821  AvgInst =  34,760,088,607  Cover = 79%
Invocs =               3  TotTrip =          50,334  (Min: 16778, Max: 16778)
------------------------------------------------------------------------------------------
mainSort:944<140003015>, mainSort<1400029b0>, blocksort.c, Parent: mainSort:934<140002f16>
Insts  =  51,570,828,455  AvgInst =          46,762  Cover = 39%
Invocs =       1,102,818  TotTrip =      19,517,066  (Min: 0, Max: 13562)
------------------------------------------------------------------------------------------
```

Figure 22: An example of binary-level loop profiling of 401.bzip in SPEC CPU2006: For example, “`main:311<1400014dd>`” reads “a loop at line 311 of function `main` (source file: spec.c), and the loop header is at `0x1400014dd`”. “`Insts`” is the number of executed instruction in this loop throughout the execution. “`Cover`” is the ratio of the total time spent in this loop (and their children loops, if any) to the whole execution time.

precise results. We discuss the algorithm and challenges. A significant performance improvement is achieved: from 24 times slowdown [94] for SPEC CPU2000 to 4 times slowdown for SPEC CPU2006.

4.2 *An Efficient Loop-Profiling Algorithm*

We mainly focus on the implementation of a *binary-level* loop profiler. Moseley et al. proposed LoopProf, a loop profiler for x86 binaries [94] based on Pin [85]. The key algorithm of LoopProf is the *dynamic* loop detection. First, all basic blocks are instrumented to track the beginning of a basic block. In the runtime, an encountered basic block is pushed on a stack. If a basic block is already in the stack, a loop has been detected; this basic block becomes the header of this loop.

The benefit of this algorithm is the simplicity and robustness because no static-time analysis such as loop structure discovery is needed in binary. It is well known that analyzing x86 and x86-64 binaries is a hard task because of many idiosyncrasies of x86 and binary itself. As we will discuss, recovering loop structures and instrumenting from an x86 has several implementation challenges.

However, the time overhead of LoopProf is significant because all basic blocks are instrumented, and a stack operation and a duplication check are needed. The overhead of LoopProf for SPEC CPU2000 is more than a 20 times slowdown on average. In our data-dependence profiler, SD^3, the overhead is ultimately limited by the loop profiler.[1] Achieving the optimal performance in loop profiling is critical. LoopProf provides a sampling technique to reduce the overhead, but we do not take a sampling technique as claimed in Section 2.4.

To solve this overhead issue, we *statically* discover the loop structures and *minimally* instrument instructions to capture loop beginning, iteration, and

[1]Strictly speaking, the ideal performance of dependence profiling is the overhead of loop profiling *and* the overhead of tracing memory instructions. Because many memory instructions are instrumented, nearly all basic blocks are instrumented and captured in the runtime.

termination in x86 binary. For binary-level approach, we first recover CFG and extract loop structures by using the well-known dominator-based loop-detection algorithm. Next, basic blocks that contain pre-headers, back edges, and exit edges are appropriately instrumented. This approach guarantees the optimal performance because the minimum instructions are instrumented. We solve a number of technical problems for this approach.

Interestingly, the motivation of LoopProf's dynamic loop discovery is to avoid such difficulties in handling x86 binaries. We demonstrate that a robust and fast binary-level loop profiler can be implemented.

4.2.1 Reconstructing CFGs and Loop Structures from Binary

Forming a control flow graph from a program is the very first step in any compiler-based analyses, and its algorithm can be easily found in a compiler textbook [3]. Detecting loop structures is also a classic problem, where a complete solution exists. We use the concept of dominators: A node X *dominates* a node Y if every path from the start node to Y includes X. A natural loop is formed by finding a *loop header* and a *back edge*. A pre-header and an exit edge of a loop are also easily defined. An exit edge connects a basic block inside of a loop and a basic block outside of the loop. The theory is clear, but implementation in an x86 binary faces many challenges and corner cases.

Recovering a correct CFG from a sole x86 binary has many obstacles. Function boundaries are even unknown in static time unless the binary is executed without heuristics. At least, debugging information, such as DWARF for Linux/ELF [26] and PDB for Windows [128, 92], is needed to analyze an x86 binary. Unless a binary is highly optimized, we may find function boundaries and source code mapping from debugging information.

However, there are still many challenging corner cases. A notable case is

an indirect branch, where determining outgoing edges from an indirect branch node is difficult. For an arbitrary dynamic dispatch (e.g., callback), obtaining target addresses could be impossible in static time. In some cases, however, we can reconstruct target addresses of an indirect branch such as trampolines and jump tables. We discuss the later case. A naive translation of a switch-case statement would be a chain of if-else statements with direct jumps. However, many compilers optimize with an indirect jump whose targets are stored in an array called jump table. Each compiler has its own style to generate jump table code. Once understanding a pattern, we can recover the targets of a jump table, and a precise CFG can be built in static time. A problem regarding this jump table is discussed in Case 2 of Section 4.3.

4.2.2 Instrumenting Loop-Behavior Instructions

The critical part of the optimal loop profiling is to minimize instrumented instructions. We instrument *only* the following basic blocks: (1) pre-headers, (2) *back-edge blocks*, and (3) *exit blocks* of a loop.[2] However, working in binary brings many challenges. The most critical reason is that we cannot modify basic blocks during dynamic binary-level instrumentation. In turn, we cannot simplify and normalize loop structures. In a compiler-based approach, for example, a single loop pre-header can be created by either introducing a new block or modifying other blocks. In contrast, multiple pre-headers may exist for a loop in a binary-level approach.

Figure 23 illustrates how we minimally instrument a loop to capture beginning, iteration, and termination. As the figure shows, we observe that even a simple loop is translated differently by major production compilers. Perhaps Figure 23(a) is the most straightforward style. We instrument the fall-through path of the pre-header

[2]Back-edge blocks and exit blocks are the basic blocks that contain the sources of back edges and exiting edges, respectively. These blocks have either a branch or fall-through to the destination.

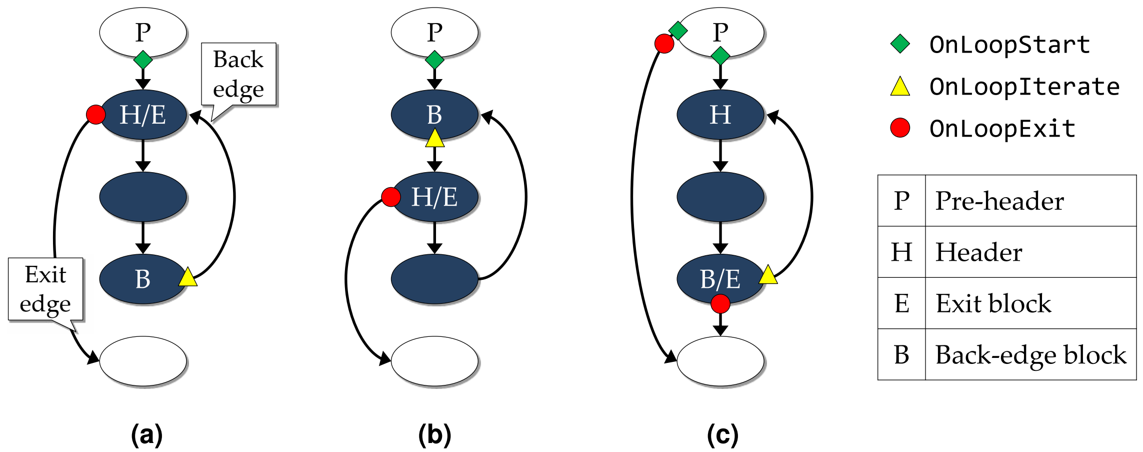


Figure 23: x86-64 loop-code-generation styles of three major compilers (gcc, Intel C/C++ compiler, and Microsoft C/C++ compiler) and instrumentation points. No optimization was applied.

(P) with `OnLoopStart`. The loop header (H) and the exit block (E) are co-located in a single basic block (denoted by H/E) with a conditional branch. On the taken-branch of this node, we insert `OnLoopExit`. Finally, the unconditional jump at the back-edge block (B) is instrumented with `OnLoopIterate`. Similarly, the case of Figure 23(b) is handled. However, Figure 23(c) is an exceptional case. If the trip count of the loop is zero, this code simply jumps to the outside of the loop. If the same instrumentation strategy of the cases of (a) and (b) were applied, we cannot detect an invocation in case of the zero iteration, resulting in a different profiling result by compilers. To make a consistent result regardless of compilers, we instrument the taken branch of a pre-header with `OnLoopStart` immediately followed by `OnLoopExit`. Our implementation considers all these cases.

Another subtle issue is the instrumentation position. For example, `OnLoopStart` for Figure 23(a) can be inserted either after the last instruction of the P node, or before the first instruction of the H/E node. We chose the first approach. `OnLoopIterate` and `OnLoopExit` are instrumented in the same way.

4.3 *Challenges: Case Studies and Solutions*

The important issues in our binary-level loop profiler are discussed so far. This section elaborates more complex cases that should be resolved to make a robust binary-level loop profiler.

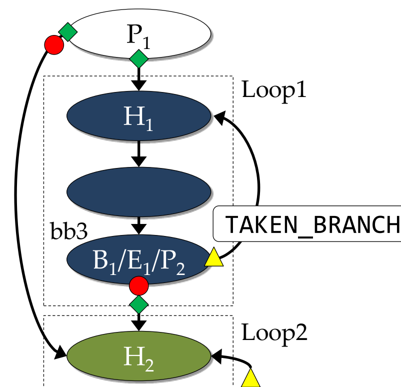


Figure 24: A complex case of binary-level loop instrumentation: A single basic block can be a back-edge block (B), an exit block (E) and a pre-header (P) at the same time. The basic block, bb3, is a back-edge block and an exit block of the Loop1. Simultaneously, bb3 is a pre-header of Loop2.

Case 1 Beside the cases of Figure 23, we further discovered a more complex case illustrated in Figure 24, where Loop2 immediately follows Loop1. For the basic block 'bb3', not only a back-edge block and an exit block but also a pre-header of Loop2 are located in this single basic block. We must insert `OnLoopStart` for Loop2 *after* inserting `OnLoopExit` for Loop1. This difficulty is because we cannot create and edit basic blocks in an easy way with Pin. In a compiler-based approach, the pre-header of Loop2 will be located in a separate basic block.

Case 2 The problem of indirect branch with switch-case is demonstrated in Figure 25, which is found a benchmark of SPEC CPU2006. This switch has three cases, but each case statement contains a loop, although this is unusual. The switch is optimized to an indirect branch with a jump table. The node "`jmp eax`" is the

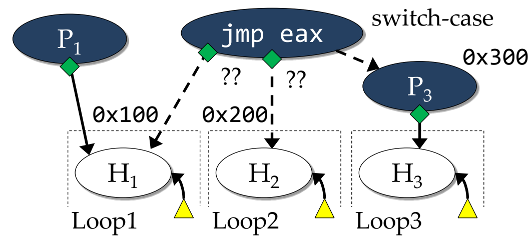


Figure 25: A case of an indirect-branch with a jump table: The "`jmp eax`" node, which is a switch-case, shares several pre-headers of loops. Unless identifying targets of the indirect branch (dotted lines), `OnLoopStart` cannot be instrumented.

node where branch occurs by the loaded address (one of `0x100`, `0x200`, `0x300`) in register `eax`. The problem is that `OnLoopStart` for Loop1 and Loop2 cannot be instrumented unless the CFG has edges between the "`jmp eax`" node and the headers of these loops. If so, the profiler may not able to detect the beginning of Loop1 and Loop2, but the back-edge blocks are properly instrumented. Our heuristic to recover jump table, described in Section 4.2.1, solves this problem.

Case 3 An interesting case regarding loop detection is sketched in Figure 26. The code does not seem to have any loop, but our loop detection algorithm identifies this combination as a natural loop. Programmers may be confused with such a false loop. This case can be, however, safely profiled if the switch-case is applied, as shown in Figure 25. A compiler-based instrumentation does not consider this combination as a loop. We do not write an explicit routine to handle this case.

```
1  RETRY:
2  switch (condition) {
3    case A:
4      if (foo() == false)
5        goto RETRY;
6      else
7        break;
8  ...
```

Figure 26: A combination of switch-case and goto (line 5 to line 1) can be identified as a loop in x86 binary.

Case 4 One of the most suffering cases is *loop-stack overflow*. When a loop is started, we push an *instance* of this loop into *loop stack*. These data structures are needed to detect recursive loops via recursive function calls. On a loop exit, we pop the stack. Section 5.2.5 presents the details of the data structures.

The problem is that sometimes we cannot properly detect the beginning or termination of a loop during the runtime, mostly due to binary-level loop reconstruction. Figure 25 demonstrated a potential case where a loop start could be missed. Similarly, a loop exit may not be detected. This unfinished loop stays on the loop stack, and this loop is called again and again. Then, the loop stack will be overflow. We observed such cases in several functions of SPEC CPU2006, especially for functions that parse an input file. The best solution is to instrument correctly, but we also implement a couple of safety net. We applied the following heuristics to make the loop profiler more robust:

- We dynamically ignore ill-behaved loops. When beginning, iteration, and termination of a loop do not match, we first try to correct the behavior. For example, when a loop is detected as iterated while this loop has not been in the loop stack, which means the loop start was not detected, we enforce to perform `OnLoopStart` for this loop. However, if such mismatch continues (reaching a threshold value), we stop profiling this loop.

- On a loop-stack overflow, we examine the stack and ignore the most frequently executed loops. These loops will not be profiled anymore (both loop and dependence profiling). The stack is flushed all, and then the profiler resumes the profiling. Note that the size of loop stack is fixed to 256. Section 4.5 discusses the related results with Table 5.

- We provide a black list of loops that should not be profiled. This approach is a kind of annotation.

Using a compiler-based approach would an ultimate solution to all the issues we discussed with the sacrifice of the benefits of a binary-level approach. We also implement an LLVM-based loop profiler [77]. Clang, a front-end for LLVM, automatically provides a rich and clean IR (intermediate representation) that has various data structures (e.g., `Function`, `LoopInfo`, and `BasicBlock` in LLVM). Pinpointing instructions to be instrumented is also trivial. For example, the loop start is simply instrumenting the last instruction of the *unique* pre-header of a given loop. Some careful considerations are needed to handle early exits in a loop nest and a complex control flow, but the implementation is much easier. The four cases of this section are correctly handled by the LLVM-based profiler without any particular effort. This is because we instrument on an IR that has not been translated to hard-to-decode indirect branches with jump tables.

4.4 Implementation

We implement both Pin-based and LLVM-based loop profilers, which will be the base platform for SD^3. The basic architecture is based on a producer and consumer structure where they communicate via inter-process communication. We use a shared memory object synchronized by semaphores. The producer is either a Pin-based or an LLVM-based tracer, which reconstructs CFGs and loop structures from an x86 binary and do instrumentation. In the runtime the tracer transfers events to the analyzer. An event is to inform (1) the beginning, (2) the iteration, and (3) the termination of a loop with a loop identifier. The analyzer waits for the shared queue to be filled by the tracer. Once the queue is full, the analyzer fetches events and performs loop profiling. Figure 27 summarizes the architecture.

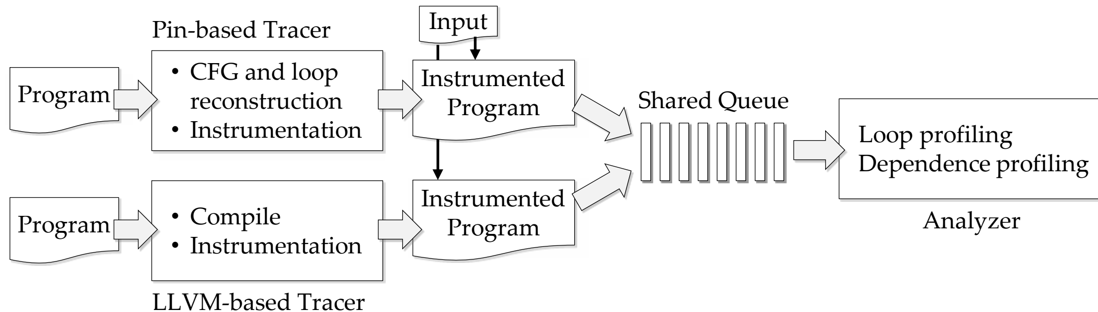


Figure 27: Overview of the loop profiler of Prospector: This framework is also shared by SD^3.

4.5 *Experimental Results*

We present the performance of our loop profiler and compare to LoopProf [94]. We implement the algorithm of LoopProf, the dynamic loop detection. Figure 28 contrasts our profiler and LoopProf when SPEC CPU2006 [134] is profiled with trained input. The geometric mean of the slowdowns is only 3.95 times while LoopProf suffers from a 11.3 times slowdown. Note that benchmarks are not optimized to have correct debugging information and easily interpretable results. This data clearly shows the effectiveness of our approach. The overhead of the LLVM-based profiler are also close to this result.

We analyze the robustness of our binary-level loop profiler to tolerate many corner cases in x86 binaries. We conduct this experiment with the highly optimized binaries by `-O3` of Intel compilers. Table 5 summarizes the results. The first column shows the number of identified loops from the binaries. Because of the aggressive optimization, we observed that more loops were identified due to inlining. The second column indicates the maximum depth of the loop stack. Except for 445.gobmk, the maximums are less than 30. We encountered a loop-stack overflow in 445.gobmk. However, our safety net dynamically fixes the stack and ignores suspicious loops. The third column shows the final depth of the loop stack, which should be zero unless a program terminates abnormally (e.g.,

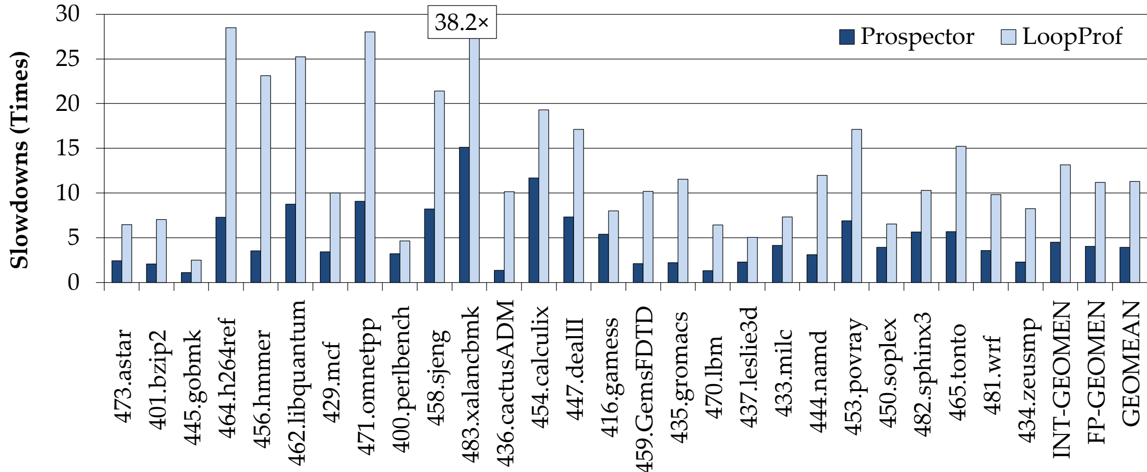


Figure 28: Time overhead comparison of Prospector's loop profiler and LoopProf with SPEC CPU2006 and the train inputs: Benchmarks from 473.astr to 483.xalancbmk are INT, and the others are FP. The benchmarks are compiled without optimizations. Slowdowns are against the native execution.

segmentation fault or `exit()`). Five INT and two FP benchmarks were terminated with the non-empty stack. The final column has the numbers of dynamically ignored loops whose behavior are not correctly captured. Without this ignoring heuristic, the loop-stack overflow occurs more frequently.

4.6 *Summary of This Chapter*

This chapter presented the loop profiler of Prospector. We implemented a low-overhead and robust loop profiler using both Pin and LLVM. In particular, many x86-binary related challenges were discussed and resolved. The performance of the loop profiler is minimal: only a 3.95 times slowdown by average against the native execution in SPEC CPU2006.

Table 5: Robustness of our binary-level loop profiler for SPEC CPU2006 with the train inputs: Loop stack overflow occurred once on 445.gobmk. See the maximum depth of the loop stack was 187. Some programs were not terminated with an empty loop stack. 403.gcc was not able to be instrumented successfully. This experimentation was done with optimized binaries.

Name	# of loops	Max depth	Final depth	# of ignored
473.astar	139	6	0	0
401.bzip2	205	10	0	0
445.gobmk	1295	187*	12	33
464.h264ref	1921	15	0	0
456.hmmer	885	5	0	0
462.libquantum	118	5	0	0
429.mcf	73	4	0	1
471.omnetpp	549	7	2	6
400.perlbench	1073	15	8	12
458.sjeng	277	8	1	0
483.xalancbmk	3654	20	13	15
436.cactusADM	1304	11	0	3
454.calculix	4350	11	2	4
447.dealII	6538	22	0	0
416.gamess	18897	14	0	0
459.GemsFDTD	1120	5	0	0
435.gromacs	2197	9	0	0
470.lbm	48	3	0	0
437.leslie3d	372	6	0	0
433.milc	462	12	1	0
444.namd	620	4	0	0
453.povray	1395	29	0	15
450.soplex	783	16	0	0
482.sphinx3	625	8	0	0
465.tonto	11465	13	0	0
481.wrf	8071	9	0	0
434.zeusmp	616	6	0	0

CHAPTER V

AN EFFICIENT DYNAMIC DATA-DEPENDENCE PROFILER

5.1 Introduction

The data-dependence profiler in Prospector, *SD*^{3} takes a program and profiles with a representative input. A raw result from the profiler is list of discovered data-dependence pairs, as shown in Table 6. The raw results will be further analyzed by the post-analyzer. All or some of the following information is provided by SD^3:

- **Sources** and **sinks** of data dependences in source code lines, columns, variable names if possible, otherwise in program counters[1];
- **Types** of data dependences: Flow (Read-After-Write, RAW), Anti (WAR), and Output (WAW) dependences;
- **Frequencies** and **distances** of data dependences;
- Whether a dependence is **loop-carried** or **loop-independent**, and data dependences carried by a particular loop in **nested loops**; and
- **Data-dependence graphs** in functions and loops.

However, Section 2.8 that showed the scalability problem of the current data-dependence profiling algorithms motivates an efficient mechanism. This chapter addresses the memory and time overhead problems by proposing an efficient data-dependence profiling algorithm called SD^3. Our algorithm has two components. First, we propose a new data-dependence profiling technique using a compressed data format to reduce the memory overhead. Second, we propose the use of

[1]Data dependences from registers can be analyzed at compile time.

parallelization to accelerate the data-dependence profiling process. More precisely, this work makes the following contributions to the topic of data-dependence profiling:

1. *Reducing memory overhead by stride detection and compression along with new data-dependence calculation algorithm:* We demonstrate that SD^3 significantly reduces the memory consumption of data-dependence profiling. SD^3 is not a simple compression technique; we should address several issues to achieve the profiling to be memory efficient. The failed benchmarks in Figure 6(a) are successfully profiled by SD^3 on a 12 GB memory machine.

2. *Reducing runtime overhead with parallelization:* We show that our memory-efficient data-dependence profiling itself can be effectively parallelized. We observe an average speedup of $4.1\times$ on profiling SPEC 2006 using eight cores. For certain applications, the speedup can be as high as $16\times$ with 32 cores.

Recall that SD^3 is built on our loop profiler. In addition, SD^3 is an extension of the baseline profiling algorithm, *the pairwise method*. Although some previous profiling algorithms [75, 81] are similar to the pairwise method, neither of them precisely and fully developed algorithms for our purpose. Therefore, we first present the details of the pairwise method and then propose SD^3.

5.2 The Baseline Algorithm: The Pairwise Method

This section describes our baseline algorithm, *the pairwise method*, which is still the state-of-the-art algorithm for existing tools. SD^3 is implemented on top of the pairwise method. At the end of this section, we summarize the problems of the pairwise method. We begin our descriptions of the algorithm by focusing on data dependences within *loop nests* because loops are major parallelization targets.

Table 6: Memory traces and discovered dependences of Figure 29.

	j = 1	j = 2
i = 1	A[1][1] = A[1][0] + 1; B[1][1] = B[2][1] + 1;	A[1][2] = A[1][1] + 1; B[1][2] = B[2][2] + 1;
i = 2	A[2][1] = A[2][0] + 1; B[2][1] = B[3][1] + 1;	A[2][2] = A[2][1] + 1; B[2][2] = B[3][2] + 1;

(a) Memory traces (Boldface means conflicting accesses)

Loop	Var	Source/Sink (R/W, Line#, Col#)	Dependence (Type, Freq)
For-i	B[]	(R,5,15)→(W,5, 5)	Loop-carried WAR, 2
For-j	A[]	(W,4, 5)→(R,4,15)	Loop-carried RAW, 2

(b) Discovered dependences (Inductions i and j are ignored.)

However, our algorithm is easily extended to find data dependences in arbitrary program structures, for example, dependences among function calls and loops across function boundary and recursion. These issues are presented in Section 5.2.3.

5.2.1 Checking Data Dependences in a Loop Nest

In the big picture, in order to calculate data dependences in a loop, we find conflicts between the memory references of the current loop iteration and the previous iterations that have been executed so far.

Our pairwise method temporarily buffers all memory references during the *current* iteration of a loop. We call these references *pending references*. When an iteration ends, we compute data dependences by checking pending references

```
1  // A, B are dynamically allocated integer arrays
2  for (int i = 1; i <= 2; ++i) {     // For-i
3    for (int j = 1; j <= 2; ++j) {   // For-j
4      A[i][j] = A[ i ][j-1] + 1;
5      B[i][j] = B[i+1][ j ] + 1;
6    }
7  }
```

Figure 29: A simple example of data-dependence profiling.

❶ i = 1, j = 1, PC@6

For-j: History	

For-j: Pending	
A[1][0]	R
A[1][1]	W
B[2][1]	R
B[1][1]	W

❷ i = 1, j = 2, PC@6

For-j: History	
A[1][0]	R
A[1][1]	W
B[2][1]	R
B[1][1]	W

For-j: Pending	
A[1][1]	R
A[1][2]	W
B[2][2]	R
B[1][2]	W

Loop-carried RAW on **A[][]** in **For-j**

❸ i = 1, PC@7

For-i: History	

For-i: Pending	
A[1][0]	R
A[1][1]	RW
B[1][2]	W
B[1][1]	W
B[1][2]	W
B[2][1]	R
B[2][2]	R

❹ i = 2, j = 1, PC@6

For-j: History	

For-j: Pending	
A[2][0]	R
A[2][1]	W
B[3][1]	R
B[2][1]	W

❺ i = 2, j = 2, PC@6

For-j: History	
A[2][0]	R
A[2][1]	W
B[3][1]	R
B[2][1]	W

For-j: Pending	
A[2][1]	R
A[2][2]	W
B[3][2]	R
B[2][2]	W

Loop-carried RAW on **A[][]** in **For-j**

❻ i = 2, PC@7

For-i: History	
A[1][0]	R
A[1][1]	RW
B[1][2]	W
B[1][1]	W
B[1][2]	W
B[2][1]	R
B[2][2]	R

For-i: Pending	
A[2][0]	R
A[2][1]	RW
B[2][2]	W
B[2][1]	W
B[2][2]	W
B[3][1]	R
B[3][2]	R

Loop-carried WARs on **A[][]** in **For-i**

Figure 30: Snapshots of the pending and history tables of each iteration when the pairwise method profiles Figure 29. Each snapshot was taken at the end of an iteration. 'PC@6' (PC is at line 6) and 'PC@7' mean the end of an iteration of `For-j` and `For-i`, respectively. Notice that the tables actually have absolute addresses, not a symbolic format like `A[i][j]`, which is only for the illustration purpose. The table entry only shows R/W, but also remembers the occurrence count and the timestamp (trip count) of the memory address to calculate frequencies and distances of dependences, respectively.

against the *history references*, which are the memory references that appeared from the first iteration to the previous loop iteration. These two types of references are stored in the *pending table* and the *history table*, respectively. Each loop has its own pending and history tables instead of having the tables globally. This is needed to compute data dependences correctly and efficiently while considering (1) nested loops and (2) loop-carried/independent dependences.

We explain the pairwise algorithm with a simple loop nest example in Figure 29 and Figure 30. We intentionally use a simple example. This code may be easily analyzed by compilers, but the analysis could be difficult if (1) dynamically allocated arrays are passed through deep and complex procedure call chains, (2) the bounds of loops are unknown at compile time, or (3) control flow inside of a loop is complex. We detail how the pairwise algorithm works with Figure 30:

- ❶: During the first iteration of `For-j` (`i = 1, j = 1`), four pending memory references are stored in the pending table of `For-j`. After finishing the current iteration, we check the pending table against the history table. For the first iteration, the history table is empty. Before proceeding to the next iteration, the pending references are *propagated* to the history. This propagation is done by *merging* the pending table with the history table (i.e., an entry of a table is simply copied if not existed, or merged if existed). This merge operation in the pairwise method is further discussed in Section 5.2.6, and will be more complex in SD^3.

- ❷: After the second iteration (`i = 1, j = 2`), we now see a loop-carried RAW on `A[1][1]` in `For-j` by checking the two tables. Meanwhile, `For-j` terminates its first invocation.

- ❸: At the same time, observe that the first iteration of the outer loop, `For-i`, is also finished. In order to handle data dependences across loop, we treat an

inner loop as if it were completely *unrolled* to its upper loop. This is done by propagating the history table of For-j to the pending table of For-i. Hence, the entire history of For-j is now at the pending table of For-i. Meanwhile, any dependence between the history table of For-j and the pending table of For-i are checked now. (These dependences will be loop-independent dependences. See more details in the following section.)

- ❹ and ❺: For-j executes its second invocation.
- ❻: Similar to ❸, the history of the second invocation of For-j is again propagated to For-i. Data dependences are checked, and we discover two loop-carried WARs on B[][] with respect to For-i.

In this example code, programmers can parallelize the outer loop after removing the WARs by duplicating B[]. The inner loop is not lucrative for parallelization due to the short-distant loop-carried RAWs on A[].

5.2.2 Handling Loop-independent Dependences

When reporting data dependences inside a loop, we must distinguish whether a dependence is *loop-independent* (i.e., dependences within the same iteration) or *loop-carried* because its implication is very different on judging parallelizability of a loop. While loop-independent dependences do not prevent parallelizing a loop by DOALL, loop-carried flow dependences generally prohibit parallelization except for DOACROSS or pipelining.

Consider the code in SPEC 179.art, shown Figure 31. The loop in scan_recognize at line 4 calls match(). The code communicates a result via a global variable, pass_flag. This variable is always initialized on every iteration at line 6 before any uses, and may be updated at line 18 and finally consumed at line 8. Therefore, a loop-*independent* flow dependence exists on pass_flag. Because the dependence

```
 1  void scan_recognize(...)
 2  {
 3    for (j = starty; j < endy; j += stride) {
 4      for (i = startx; i < endx; i += stride) {
 5      ...
 6      pass_flag = 0;
 7      match();
 8      if (pass_flag == 1)
 9        do_something();
10      ...
11      }
12    }
13  }
14
15  void match() {
16    ...
17    if (condition)
18      pass_flag = 1;
19    ...
20  }
```

Figure 31: Loop-independent flow dependences on `pass_flag` in 179.art.

is loop-independent, this dependence does not prevent parallelization of the loop.[2] If we do not differentiate loop-carried and loop-independent dependences, programmers might think the reported dependences could stop parallelizing the loop.

To handle such loop-independent flow dependences, we introduce a *killed* address, which is very similar to the *kill* set in data-flow analysis [3]. We mark an address as killed once the memory address is written in an iteration. Then, subsequent accesses within the same iteration to the killed address are no longer stored in the pending table and reported as loop-independent flow dependences. Killed information is cleared on every iteration.

In this example, once `pass_flag` is written at line 5, its address is marked as killed. Any following accesses on `pass_flag` are reported as loop-independent flow dependences and not stored in the pending table. However, the write operation at

[2]A global variable `pass_flag` may make loop-carried output dependences, but this dependence can be removed easily by privatizing the variable.

```
1  for (int i = 0; i < N; ++i) {
2    var1 = ...;
3    ...  = var2;
4    for (int j = 0; j < M; ++j) {
5      ...  = var1;
6      var1 = ...;
7      var2 = ...;
8    }
9    ...
10 }
```

Figure 32: Three kinds of loop-independent dependences in a loop nest.

line 5 is the first access within an iteration. Therefore, pass_flag does not make loop-carried flow dependences.

Figure 32 illustrates a case of loop-independent dependences involved in a loop nest. Suppose a program is at the line 9: we have the history table of For-j and the pending table of For-i is alive. As described previously, the history table becomes a part of the pending table of the parent. At this moment, we need to check potential loop-independent dependences because the accesses on var1 and var2 in lines 5 to 7 are now exposed to the For-i. First, var1 has been killed at line 2, so the access on 5 is reported as loop-independent flow dependence. The access on var1 at line 6 may be reported as loop-independent output dependence. However, this dependence can be disregarded because the closest dependence of the access at line 6 is the loop-independent anti-dependence with the access at line 5. Regarding var2, notice that there is an entry for var2 in the pending table of For-i from line 2. Hence, when merging the tables, we see that the access at line 7 makes loop-independent anti-dependence. After merging the entry of var2, we make this address as killed because the latest access on this address is a write.

5.2.3 Dependences in Functions and Handling Function Calls

We focused on finding data dependences in loops so far, but may want to see data dependences between function or within a function, not necessarily bounded to

```
1  T foo(...)
2  {
3    T ret;
4    do {
5      ret = function_body(...);
6    } while(0);
7    return ret;
8  }
```

Figure 33: Finding dependences among functions and loops: We treat as if an imaginary loop (do-while loop), which encompasses the whole function body with the zero trip count, has been started.

any loop. The pairwise algorithm as well as SD^3 easily captures data dependences between any functions and loops by a simple extension as depicted in Figure 33.

The profiler consider that an imaginary loop, which encloses the entire function body, is inserted to the function to be profiled. Dependences between functions are the same with the loop-independent dependences of imaginary loops.

The pairwise method and SD^3 handle loop nests with function calls and recursions without any difficulty with the data structures: `LoopInstance` and `LoopStack`, which are presented in Section 5.2.5.

5.2.4 Computing Data-Dependence Distance

Data-dependence distance [114] is useful in advanced compiler optimizations. Dependence distance in a loop nest is defined as the difference of two *iteration vectors* of the source and sink of a dependence. However, this work does not compute such distance vectors although we can compute them easily. This is because we do not just keep an iteration vector for a loop; we simply maintain a trip count of a loop. However, a loop can easily build an iteration vector by fetching the ancestor loops' iteration information from the loop stack. Hence, distance vectors can be computed without any technical challenges.

In this work, we report only the scalar dependence distance between the source and the sink of a dependence only within a loop. This information is easily

obtained by storing *the last access iteration number* to a memory reference data structure such as POINT (Section 5.2.5) and STRIDE (Section 5.3.4). Zero of the distance indicates a loop-independent dependence, and the sign of the dependence distance indicates the direction of the dependences.

5.2.5 Summary of the Pairwise Method

The pairwise algorithm is summarized in Algorithm 1 with the following data structures, also sketched in Figure 34:

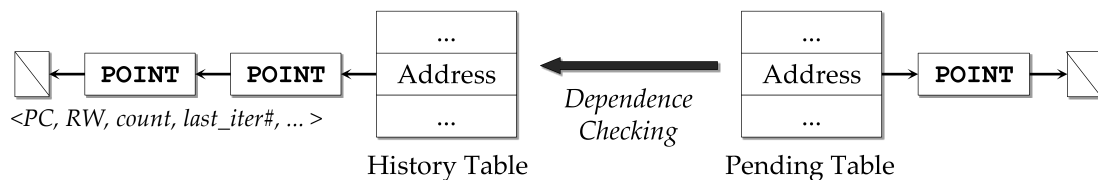


Figure 34: Data structures in the Pairwise method: History and pending tables are hash tables that associate an address with a list of POINT. All accessing PCs for a particular memory address are stored to report PC-wise dependence results.

- POINT: This represents a memory access on a particular single memory address from a PC. This structure has the following fields: (1) the PC address (or the location identifier of the memory instruction[3]); (2) the number of total accesses from this PC; (3) the pointer to the next POINT (if any), which is needed a memory address can be touched by multiple PCs); and (4) miscellaneous information including the read/write mode and the last accessed iteration number to calculate dependence distance.

- Loop: This represents and keeps the information of a *static* loop such as loop name, loop location, parent loop, and children loops (if any). This also maintains the statistics throughout the program execution including the

[3]In a Pin-based (binary-level) implementation, using program counter addresses is natural. In contrast, an LLVM-based implementation cannot directly see PC addresses because instrumentation is performed on LLVM IR (intermediate representation). Instead we use an identifer of a memory instruction represented by an integer (location ID). In this thesis, PC denotes both cases.

total invocation count, the average/minimum/maximum trip count per each invocation, and the average execution time per each invocation.

- `LoopInstance`: This represents a *dynamic* execution status of a loop by storing all runtime information including the number of invocation, current trip count, observed memory reference information, and discovered dependences. We separate `LoopInstance` from `Loop` to handle recursive invocation of a loop. Each loop instance has important tables for the data-dependence profiling, named the *pending* and the *history* tables.

- `PendingTable`: This table stores memory accesses in the *current* iteration of a loop. The table is implemented as a hash table keyed by memory address for fast conflict detection.[4]

- `HistoryTable`: The history table remembers memory accesses in *all* executed iterations of a loop so far. The structure is mostly identical to the pending table except for the killed bits.

- `ConflictTable`: All data dependences found throughout the program execution are stored in this table.

- `LoopStack`: This stack keeps the history of loop execution like the callstack for function calls. A `LoopInstance` is pushed or popped as the corresponding loop is executed and terminated. We need this loop stack to calculate data dependences across loop nests.

[4]A hash table is unordered and slow for enumeration. To address these shortcomings, we have an auxiliary vector of pointers to `POINT`, which does not necessarily increase memory consumption too much. When doing dependence checking and merging tables, we iterate this auxiliary array so that dependences are reported as they appear, and the enumeration is cheaper.

Algorithm 1 THE PAIRWISE ALGORITHM

1: When a loop, L, starts, `LoopInstance` of L is pushed on `LoopStack`.
2: On a memory access, R, of L's i-th iteration, check the killed bit of R. If killed, report a loop-independent dependence, and halt the following steps. Otherwise, store R in `PendingTable`. Finally, if R is a write, set its killed bit.
3: At the end of the iteration, check data dependences between the pending table and the history table. Report any found data dependences.
4: After Step 3, the pending table is *propagated* to the history table. Propagation is done by merging the two tables. The pending table is flushed.
5: When L terminates, flush the history table, and pop `LoopStack`.
6: To handle data dependences in a loop nest, we propagate the history table of L to the pending table of the parent of L (if exist). Data dependences of the parent loop will be recursively checked by Step 3.
7: When performing propagation (i.e., element-wise merging), loop-independent dependences (in the parent loop of L) are also checked and reported if any. In case of flow and output dependences, the conflicting entry of the history table of L is not propagated. If an access in the history table is a write, mark as killed.

5.2.6 Optimizing the Merge Operation: PC-set Optimization

We discuss an important issue on the merge operation in Algorithm 1. The merge operation is merging a history table and a pending table, shown in Figure 34. Because both history and pending tables are hash tables, merging two hash tables seems to be straightforward to implement: (1) enumerating each item from the smaller hash table; (2) checking whether the current key exists in the larger hash table; and (3) performing content-specific merge operation. At a glance, the merge operation seems to take linear time, proportional to the size of the smaller table. However, the content of the hash table is a *vector* of `POINT`. This list structure is needed because (1) a memory address may be accessed by multiple PC locations, and (2) we report PC-wise sources and sinks of discovered dependences. Computing an union of two vectors in element-wise requires $O(n \cdot m)$, where n and m are the lengths of the two vectors, respectively. Maintaining such list structure not only increases the memory space, but also consumes more processing time.

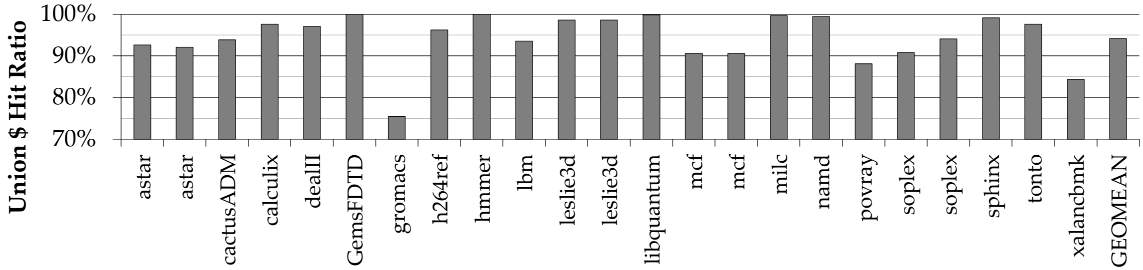


Figure 35: Hit ratio of the union-computation of SPEC 2006 (train) during the merge operation. Some benchmarks are duplicated because of multiple inputs.

In our data from SPEC 2006, the geometry average of the *maximum* length of `POINT` vectors is approximately 20. However, we observed as high as 217 in 483.xalancbmk . To reduce both time and space overhead in the pairwise method, we introduce *PC-set* optimization. This is important because SD^3 still uses the pairwise method when memory accesses cannot be compressed.

This optimization is based on the facts that the number of distinct `POINT`-list instances for SPEC 2006 was not large: the geometric mean is only 6,242 distinct instances. Note that we considered only PC and RW mode to distinguish a distinct `POINT`-list instance. We also learned that most of the union computation during the merge operation was repeated. Figure 35 shows that the average hit ratio of the union-computation cache is approximately 95%.

We employ two structures: (1) a global *PC-set table* that remembers all observed distinct lists and (2) a *cache* for the union computation. Figure 36 depicts this optimization. The point table has only a single PC-set ID instead of a list structure. The union computation is a simple table lookup of the union-computation cache. Only on a cache miss, we compute a union of the two `POINT` lists. The dependence checking code is simplified because there is no need to have list-list merge code.

This PC-set optimization, however, is a *lossy* compression. Introducing a PC set loses the precise occurrence count per each PC; the total occurrence count for a single PC set is only saved. Hence, when reporting a data dependence

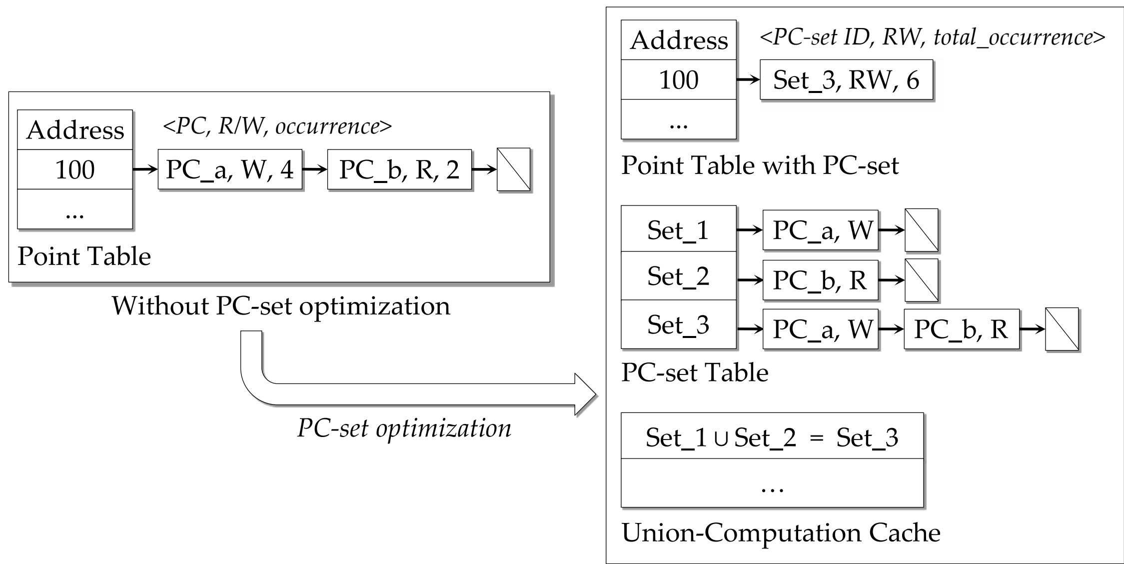


Figure 36: PC-set optimization for the fast `POINT`-list merging. A `POINT` list can be represented as a single PC-set ID, resulting in saving the memory. The union computation of `POINT` lists is accelerated by the union-computation cache. This optimization, however, is a lossy compression. A precise occurrence count per PC is lost. There could be an error in reporting data-dependence frequency.

frequency, the pairwise method may have an error. However, we still make no error on judging the existence of dependences. From our definitions of the correctness in data-dependence profiling introduced in Section 2.4, the pairwise method implements a correct profiler although a strictly correct profiler will not be implemented. This compression should not hurt the usefulness of dynamic dependence analysis.

5.2.7 Problems of the Pairwise Method

The pairwise method stores all distinct memory references within a loop invocation. The memory requirement per loop is increased as the memory footprint of a loop is increased. The memory requirement could be even worse because of nested loops. As explained in Section 5.2.1, history references of inner loops propagate to their upper loops. Only when the topmost loop finishes can all the history references within the loop nest be flushed. Many programs have fairly

deep loop nest. For example, the geometric mean of the maximum loop depth in SPEC 2006 FP is 12. Furthermore, most of the execution time is spent in loops. Hence, the whole distinct memory references often need to be stored (along with PC addresses and other information) throughout the program execution. In the following section, we solve this problem by *compression* and several algorithms.

Profiling time overhead is also critical since extremely many memory loads and stores could be traced. We attack this overhead by *parallelizing* the data-dependence profiling itself. We present our solution in Section 5.4.

5.3 A Memory-Efficient Algorithm in SD[3]

5.3.1 Overview of the Algorithm

The basic idea of solving the memory overhead problem is to store memory references as a *compressed* format. Since many memory references show stride patterns[5], our profiler can also compress memory references with a *stride* format: `A[a*i + b]`, where `i` is an induction variable of a loop, `a` and `b` are constant. However, a simple compression technique is not enough to build an efficient data-dependence profiler. We tried to compress memory streams and decompress them on dependence checking by using gzip. This naive way simply does not work because compression and decompression time just dominate the execution time. Therefore, we must directly check data dependences without explicit decompression. Then, the following problems should be addressed:

- How to detect stride patterns dynamically (Section 5.3.2),
- How to perform data-dependence checking with the compressed format *without* decompression (Section 5.3.3),

[5]This is also the motivation of hardware stride prefetchers.

- How to check dependences efficiently with both stride and non-stride patterns (Section 5.3.5), and

- How to handle loop nests and loop-independent dependence with the compressed format (Sections 5.3.6 and 5.3.7).

The above problems are now discussed in following sections.

5.3.2 Dynamic Detection of Strides

We define an address stream as a *stride* as long as the stream can be expressed as $base + stride_distance \cdot n$. SD^3 dynamically discovers strides and directly checks data dependences with strides and non-stride references. In order to detect strides, when observing a memory access, the profiler trains a *stride detector* for each PC and decides whether or not the access is part of a stride. Because the sources and sinks of dependences should be reported, we have a stride detector per PC. An address that cannot be represented as part of a stride is called a *point* in this thesis.

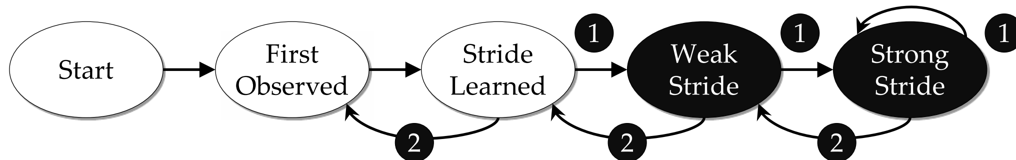


Figure 37: A finite state machine for stride detection: The current state is updated on every memory access with the following additional conditions: ❶ The address can be represented with the learned stride (stride); ❷ The address cannot be represented with the current stride (point).

Figure 37 illustrates that the state transitions in our stride detector. After watching two memory addresses for a given PC, a stride distance is learned. When a newly observed memory address can be expressed by the learned stride, FSM advances the state until it reaches the *StrongStride* state. The StrongStride state can tolerate a small number of stride-breaking behavior. For memory accesses like `A[i][j]`, when the program traverses in the same row, we will see a stride.

When a row changes, however, there could be an irregular jump in the memory address, breaking the learned stride. Having the WeakStride and StrongStride states tolerates a few non-stride accesses so that exceptional point references are minimized.

We separately handle fixed-location memory accesses (zero stride distance) for stride profiling and future uses. If a newly observed memory access cannot be represented with the learned stride, it goes back to the FirstObserved state with the hope of seeing another stride behavior. We do not further attempt to combine multiple strides from a PC into a single stride even if these strides are generated by two- (or more) dimensional accesses and compressible.

```
1  // Stride-compression-unfriendly allocation: nD style.
2  int** data2 = new int*[N];
3  for (int i = 0; i < N; ++i)
4    data2[i] = new int[N];
5
6  // Stride-compression-friendly allocation: 1D style.
7  int** data1 = new int*[N];
8  int*  raw1  = new int[N * N];
9  for (int i = 0; i < N; ++i)
10   data1[i] = raw1 + i * N;
11
12 // Using the arrays
13 for (int i = 0; i < N; ++i) {
14   for (int j = 0; j < N; ++j) {
15     ... = data1[i][j];
16     ... = data2[i][j];
17   }
18 }
```

Figure 38: Two different dynamic allocation styles for 2D array: The two styles are semantically identical. The former enables perfect stride compression.

We observed that the stride compression behavior on a two- (or higher) dimensional array is dependent on the allocation style. Consider the two allocation styles for a 2D array shown in Figure 38. The first style, which is more popular among programmers, allocates each row separately. This style virtually has no stride compression opportunity over the rows (`data[k][0]`, $0 \leq \texttt{k} < \texttt{N}$). As a result, the reading on `data1` at line 15 will create `N` separate strides.

Now notice that the second style allocates an 2D array in a contiguous 1D array while providing the exactly same access semantics with the first style. In this case, the accesses on `data2` at line 16 will be completely compressed into a single stride. The example of the matrix addition in Figure 6(c) shows the case of the perfect compression. Changing the allocation style from the first to the second style must not affect any program semantics and correctness. We strongly recommend the compression-friendly allocation style.[6] Nonetheless, even if such perfect compression is not possible, we still gain much better memory efficiency over the pairwise method.

Finally, our stride detector does not always require strictly increasing or decreasing patterns. For example, a stream [10, 14, 18, 14, 18, 22, 18, 22, 26] is considered as a stride $10 + 4 \cdot n$ ($0 \leq n \leq 4$). Note that such non-strict strides may cause slight errors when calculating the occurrence count of data dependences. We discuss this issue in Section 5.3.8.

5.3.3 Stride-Based Dependence Checking Algorithm

Checking dependences is trivial in the pairwise method. We exploit a hash table keyed by memory addresses, which enables fast searching whether or not a given memory address is dependent. Unfortunately, the stride-based algorithm cannot use such simple dependence checking because a stride represents an *interval*. This section first introduces algorithms to find conflicts within two strides (or a stride and a point, which is superseded to the case of two strides). We will then discuss how to find efficiently dependence among both strides and points in Section 5.3.5.

The key point in the new stride-based dependence checking algorithm is to find conflicts of two strides. We attack the problem through two steps: (1) finding overlapped strides and points, and (2) performing a new data-dependence test,

[6]Statically allocated arrays on stack or data segment (i.e., global variables) are contiguous. They are perfectly compressible to a single stride.

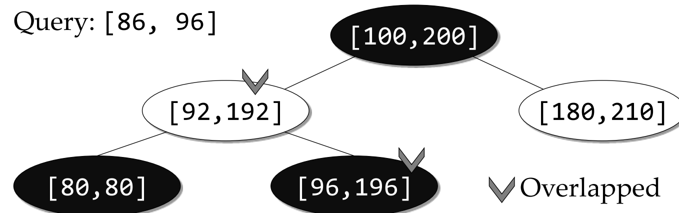


Figure 39: Interval tree (based on a Red-Black Tree) for fast overlapping point/stride searching. Numbers are memory addresses. Black and white nodes represent Red-Black properties.

DYNAMIC-GCD, to calculate the exact conflicts.

For the first step, the overlapping test, we employ an *interval tree*, which is based on the Red-Black Tree [18]. The test finds *all* overlapping strides and points in a tree for a given input. Figure 39 shows an example of an interval tree. Each node represents either a stride or point. Through a query, a stride of [86, 96] overlaps with [92, 192] and [96, 196].

This interval-tree based searching imposes a challenge. Finding *an* intersecting interval for a given interval is performed in $O(\log n)$. On the other hand, this problem needs to find *all* interesting intervals. In the worst case, where all the nodes of an interval tree intersect an input, linear time is required. Nonetheless, we found that using an interval tree is still better. On a unit test with randomly generated intervals, we observed that an amortized search time of interval tree is 30-40% faster than a simple linear search. To avoid excessive interval-based searching, we further employ an optimization that is presented in Section 5.3.5.

The next step is an actual data-dependence test between two overlapping strides. We extend the well-known GCD (Greatest Common Divisor) test to DYNAMIC-GCD test in two directions: (1) we dynamically construct affined descriptors from address streams to use the GCD test, and (2) we count the exact number of dependence occurrences (many static dependence test algorithms give a *may* answer along with *dependent* and *independent*).

To illustrate the algorithm, consider the contrived program in Figure 40. We

```
1  for (int n = 0; n <= 6; ++n) {
2    A[2*n + 10] = ...; // Stride 1 (Write)
3    ... = A[3*n + 11]; // Stride 2 (Read)
4  }
```

Figure 40: A simple example for DYNAMIC-GCD.

consider only the read at line 2 and the write at line 3, but ignore the memory accesses omitted in "...". We assume that array `A` is a type of `char[]` and begins at address 10. Two strides will be created:

(1) Stride 1 from line 2: [20, 32] with the distance of 2; and

(2) Stride 2 from line 3: [21, 39] with the distance of 3.

DYNAMIC-GCD returns the exact number of conflicting addresses in the two strides. The problem is reduced to solving a *Diophantine* equation:

$$2x + 20 = 3y + 21 \ \ (0 \leq x,\ y \leq 6). \tag{2}$$

A Diophantine equation is an indeterminate polynomial equation in which only integer solutions are allowed. In our problem, we solve a linear Diophantine equation such as $ax + by = 1$. DYNAMIC-GCD, described in Algorithm 2, solves this equation. We detail the computation steps using Figure 41:

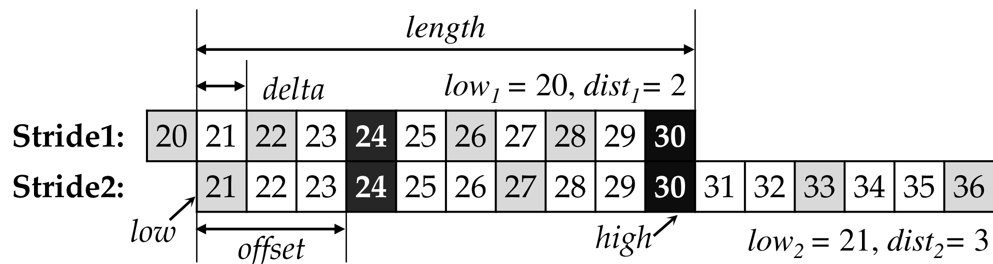


Figure 41: Two strides in Figure 40. Lightly shaded boxes indicate accessed locations, and black boxes are conflicting locations. The terms (length, delta, low, high, and offset) are explained in the following paragraphs.

Algorithm 2 DYNAMIC-GCD

Inputs: Two stride: $(low_1, high_1, dist_1)$, $(low_2, high_2, dist_2)$
Output: Return the number of dependences of the two strides.
Require: $low_1 \leq low_2$; otherwise swap the strides.

1: Calculate low, $high$, and $length$ as shown in Figure 41.
2: $delta \leftarrow (dist_1 - ((low - low_1) \mod dist_1)) \mod dist_1$
3: $gcd \leftarrow$ The greatest common divisor of $dist_1$ and $dist_2$
4: **if** $(delta \mod gcd) \neq 0$ **then**
5:     **return** 0
6: **end if**
7: $x, y \leftarrow$ EXTENDED-EUCLID$(-dist_1, dist_2)$
8: $lcm \leftarrow$ The least common multiple of $dist_1$ and $dist_2$
9: $offset \leftarrow ((dist_2 \cdot y \cdot delta/gcd) + lcm) \mod lcm$
10: $result \leftarrow (length - (offset + 1) + lcm)/lcm$
11: **return** $\max(0, result)$

1. Sort and obtain the overlapped bounds and lengths: Let $low_1 \leq low_2$; otherwise swap the strides. In Figure 41, the bounds are $low = 21$, $high = 30$, and $length = 10$.

2. Check the existence of the dependence by the GCD test without considering the bounds: We only have the runtime stride information. In order to use the GCD test, we transform the strides as if we have the common array base such as A[$dist_1 \cdot x + delta$] and A[$dist_2 \cdot y$], where $delta$ is the distance between low and the immediately following accessed address in Stride1. Then, we can use the GCD test. In Figure 41, $delta$ is 1; the GCD of 2 and 3 is 1, which divides $delta$. Therefore, the strides *may* be dependent.

3. Count the exact dependences within the bound: To do so, we first compute the smallest conflicting point in the bound (24 in Figure 41) by using EXTENDED-EUCLID [18], which returns x and y in $ax + by = \gcd(a, b)$, where a is $-dist_1$, and b is $dist_2$. $offset$ is then defined as the distance between low and this smallest conflicting point, which is 3 in Figure 41. Observe that the difference between two adjacent conflicting points is the least common

multiple of $dist_1$ and $dist_2$ (6 in Figure 41). Then, we can count the exact number of dependences: two addresses (24 and 30) are conflicting. The strides are dependent.

Clarification of Equation (2) **and Figure 40** We discussed DYNAMIC-GCD with the code of Figure 40. However, this code does *not* actually create the two strides as shown in Figure 41 and Equation (2). Because the loop is single-level, the code will only check dependences between two pending points (the write at line 2 and the read at line 3) against the history strides as the loop iterates. On `i`-th iteration (assuming zero-based index), the two points, $2i+20$ (the write at line 2) and $3i+21$ (the read at line 3), are being checked with the two history strides, $2k+20$ and $3k+21$, $(0 \leq k < i)$.[7] Therefore, there is no moment when Equation (2) is performed. We intentionally used an incorrect code to explain DYNAMIC-GCD easier.

```
1  for (int i = 0; i <= 7; ++i) {
2    A[2*i + 10] = ...;    // Stride 1 (Write)
3    for (int j = 0; j <= 6; ++j) {
4      ... = A[3*j + 11]; // Stride 2 (Read)
5    }
6  }
```

Figure 42: A correct program that will shows Figure 41 and Equation (2).

Figure 42 contains the correct code that will exactly show the case of Figure 41 and Equation (2). When `i` is 7 and the inner loop just finishes, the history stride table of `for-j` is now propagated (i.e., merged) to the pending stride table of `for-i`. After the propagation, when the current iteration (`i` = 7) finishes, we will finally see the dependence checking depicted in Figure 41 and Equation (2).

[7]In our implementation, the very first few points until learning a stride still exist in the point table. Hence, strictly speaking, there are a few history points. However, we can safely assume that no such transient point exists to explain the DYNAMIC-GCD.

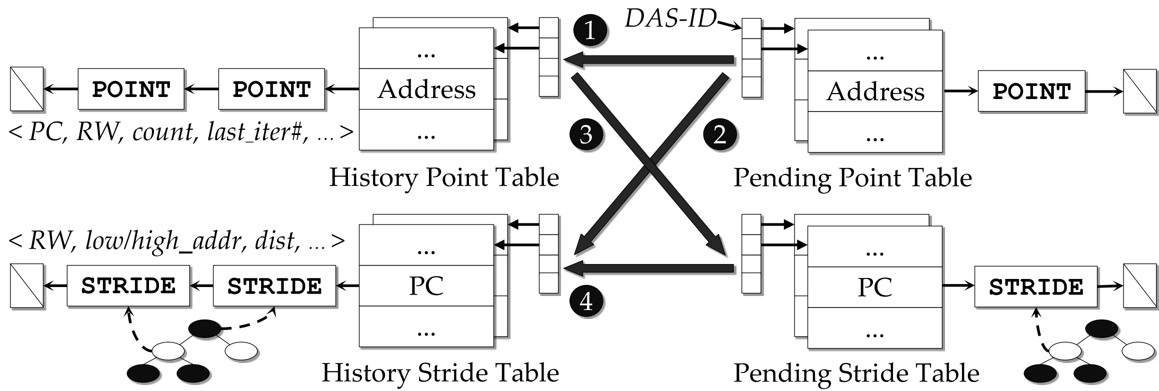


Figure 43: Structures of the point and stride tables and four cases of the dependence checking: This figure shows the moment when the current iteration finishes. A stride table has an associated interval tree. The tables support DAS-ID (dynamic allocation-site ID) to minimize the checking space. Our implementation uses an additional hash table, where the key is DAS-ID and the value is either point or stride sub-table that only contains memory references from the same DAS-ID.

5.3.4 Overview of the Memory-Efficient SD^3 Algorithm

The first part of SD^3, a memory-efficient algorithm, is presented in Algorithm 3. The algorithm augments the pairwise algorithm and will be parallelized to decrease time overhead. Section 5.2.5 described key data structures for the pairwise method, and now we update and extend the data structures as follows:

- `POINT`: This structure now only represents a non-stride, or point memory access, from a single PC. The member fields remain the same.
- `STRIDE`: This represents a compressed stream of memory addresses from a PC. This structure has (1) the lowest and highest addresses; (2) the stride distance; (3) the number of total accesses in this stride; and (4) the pointer to the next `STRIDE` because a PC can create multiple strides that cannot be combined. Notice that the miscellaneous fields such as RW mode and the last accessed iteration number are stored along with the PC because these fields are common per PC.
- `PendingPointTable` and `PendingStrideTable`: These tables replace the `PendingTable`

in the pairwise. `PendingPointTable` is a hash table that associates `POINT` with memory address to enable fast conflict checking. `PendingStrideTable` associates `STRIDE` with the originating PC address. The stride table has an auxiliary interval tree. Both pending tables store *killed bits* to handle loop-independent dependences. The tables also support *DAS-ID* (dynamic allocation-site ID) to accelerate the dependence checking by minimizing the search space. Figure 43 and Section 5.3.5 further discuss this optimization and the necessary change in the hash tables.

- `HistoryPointTable` and `HistoryStrideTable`: This holds memory accesses in *all* executed iterations of a loop so far. The structure equals the pending tables except for killed bits.

Although the big picture of the algorithm is described, the stride-based algorithm needs to address several challenges. The following four subsections elaborate these issues.

5.3.5 Optimizing Stride-Based Dependence Checking

Figure 43 summarizes the table structures and stride-based dependence-checking algorithm. When the current iteration finishes, the two pending tables are checked against the two history tables. Because it has both point and stride tables, the dependence checking now requires four sub-steps for every pair of the tables, shown as four large arrows in the figure.

Arrow ❶ is the case of the pairwise method. Arrows ❷ and ❸ show the case of checking a stride table and a point table. We take *every* address in the point table and check the conflict against the stride table using the associated interval tree. Arrow ❹ is the most complex step. *Every* stride in the pending stride table needs to be enumerated and checked against the history stride table. This enumerating and checking could take a long time, especially for a deep nested loop. Therefore,

Algorithm 3 THE MEMORY-EFFICIENT ALGORITHM

Note: New steps added on top of the pairwise method are underlined.

1: When a loop, L, starts, `LoopInstance` of L is pushed on `LoopStack`.

2: On a memory reference, R, of L's i-th iteration, check the killed bit of R. If killed, report a loop-independent dependence, and halt the following steps. Otherwise, store R in either `PendingPointTable` or `PendingStrideTable` based on the result of the stride detection (Section 5.3.2). If R is a write, set its killed bit.

3: At the end of the iteration, do the stride-based dependence checking (Sections 5.3.3 and 5.3.5). Report any found dependences.

4: After Step 3, merge the pending and history point tables. Also, merge the stride tables (Section 5.3.6). Finally, the pending tables, including killed bits, are flushed.

5: When L terminates, flush the history tables, and pop the `LoopStack`. To handle loop nests, we propagate the history tables of L to the parent of L, if they exist. Propagation is done by merging the history tables of L with the pending tables of the parent of L.
Meanwhile, to find loop-independent dependences, do the stride-based dependence checking. Consider a special case of a stride kill (Section 5.3.7).

in order to reduce this enumeration and potentially vast search space, we introduce *dynamic allocation-site* optimization.

This optimization is based on the fact that a memory access on a variable or a structure must have its associated *allocation site*. Memory accesses from different allocation sites will *never* collide in a correct program. Once allocation sites are known, we need to check only dependences among memory accesses within the same allocation site, which can reduce search space significantly.

In particular, we focus on heap accesses, other than local and global accesses, because they are the main target of the analysis. To obtain allocation site IDs on heap accesses effectively, we *dynamically* track allocated heap regions and issue an ID on each heap region, which we call *dynamic allocation-site ID*, or DAS-ID.

We illustrate how DAS-ID works with Figure 44. Two `Node` are created at line 6 and 7 by `CreateNode`. It is obvious the accesses at line 8 and 9 never make dependences each other because their allocation sites are different. Here is the

```
1  Node* CreateNode(...) {
2    return new Node(...);
3  }
4
5  void foo() {
6    Node* a = CreateNode(...);
7    Node* b = CreateNode(...);
8    ... = a->data;
9    b->counter = ...;
10 }
```

Figure 44: A simple example of DAS-ID: The memory accesses at line 8 and 9 have different DAS-IDs. No need to check dependences between them.

optimization opportunity: we do not need to check dependences between memory accesses at line 8 and line 9.

We now discuss how we implement this optimization using Figure 44. One might think of using static allocation site information. This approach is not useful because both `a` and `b` have the same static allocation site at line 2. There will be no reduction in searching and checking space. Instead we obtain *dynamic* allocation site information that differentiate different call path (i.e., context sensitive). Our solution is exploiting dynamically allocated heap memory regions as a key.

First, heap allocation and deallocation functions and operators, such as `new` operator at line 2, are instrumented. In the runtime, when `CreateNode` is called at line 6, we capture the allocated memory region. Say the allocated range is $[100, 200]$. We then assign an ID, say 1, to this memory range. This is the *dynamic allocation-site ID*, or *DAS-ID*. We keep the pair of $<[100, 200], 1>$ into a global table, which is implemented as an interval tree. Similarly, suppose that a memory range of $[208, 308]$ is allocated at line 7, and 2 is then assigned as DAS-ID. $<[208, 308], 2>$ is also stored in the global table.

On memory access at line 8, we retrieve the corresponding DAS-ID of the access. Say that the address of `a->data` is 104. Because the allocated ranges are stored in an interval tree, retrieving the DAS-ID is extremely fast. The query result

is either a unique DAS-ID or no DAS-ID. The latter implies the given access is either from a local or a global variable. We obtain 1 as the DAS-ID of the access on `a->data`. This memory access is now stored in a table that is reserved for memory accesses only from DAS-ID of 1. Likewise, the write on `b->counter` whose DAS-ID is 2 is stored in a separate table. Hence, no tables from different DAS-ID are checked. We summarize how DAS-ID is implemented:

1. Instrument heap allocation and deallocation functions and operators (e.g., `malloc`, `new`, `free`, and `delete`).

2. On heap allocation, retrieve the allocated memory range, and issue a DAS-ID by an simply increasing counter. Store this pair of the allocated range and the DAS-ID into a global table. The table is an interval tree that allows a fast query of the associated DAS-ID for a given memory access.

3. On heap deallocation, delete the corresponding node.

4. On load and store, fetch the effective address, and query the tree to obtain the corresponding DAS-ID of the access. Because of an interval tree, finding an interval that includes the given point is done $O(\log n)$. If no matching range is found, the access is either a local or a global access because we currently do no track allocation sites of local and global variables. Store the memory reference to either a point or stride table reserved for this DAS-ID only.

We also need to revise the point and stride table structure. The change is briefly sketched in Figure 45 using C++ STL syntax. The original tables are renamed as *sub*

```
typedef hash_map<uint64_t /*Address*/,        POINT*>   SubPointTable;
typedef hash_map<uint64_t /*PC address*/,     STRIDE*>  SubStrideTable;
typedef hash_map<uint64_t /*DAS_ID*/,  SubPointTable*>  PointTable;
typedef hash_map<uint64_t /*DAS_ID*/, SubStrideTable*>  StrideTable;
```

Figure 45: Pseudo code for the table structure supporting DAS-ID optimization.

tables. We introduce an additional hash table layer, where the key type is DAS-ID and the value is either a point or a stride table. The valued sub-point and sub-stride tables now contain memory references from the same DAS-ID only.

5.3.6 Merging Stride Tables for Loop Nests

In the pairwise method, we propagate the histories of inner loops to its upper loops to compute dependences in loop nests. Introducing strides makes this propagation difficult. Steps 4 and 5 in Algorithm 3 require a *merge* operation of a history table and a pending table. Without strides (i.e., only points), this step is relatively straightforward to implement: we simply compute the union set of the two point hash tables.

However, merging two stride tables is not trivial. A naive solution is just concatenating two stride lists. If this is done, the number of strides could be bloated, resulting in potentially significant memory consumption. Hence, we perform *stride-level* merging rather than a simple stride-list concatenation. The example is demonstrated in Figure 46.

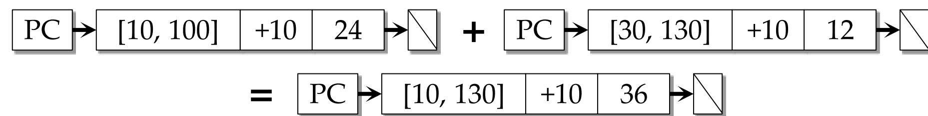


Figure 46: Two stride lists from the same PC are about to be merged. The stride ([10, 100], +10, 24) means a stride of (10, 20, ..., 100) and total of 24 accesses in the stride. These two strides have the same stride distance. Thus, they can be merged, and the number of accesses is summed.

Naive stride-level merging requires quadratic time complexity. Here, we again exploit the interval tree for fast overlapping testing. Nonetheless, we observed that tree-based searching still could take a long time if there is no possibility of stride-level merging. To minimize such waste, the profiler caches the result of the merging test in history counters per PC. If a PC shows very little chance of having stride merges, SD^3 skips the merging test and simply concatenates the lists.

5.3.7 Handling Killed Addresses in Strides

We showed that maintaining *killed* addresses is important to distinguish loop-carried and independent dependences. As discussed in Section 5.2.2, the pairwise method prevented killed addresses from being propagated to further steps. However, this step becomes complicated with strides because strides could be killed by the parent loop's strides or points.

```
1  for (int i = 0; i < N; ++i) {    // Loop_1
2    A[rand() % N] = 10;            // Random kill on A[]
3    for (int j = i; j >= 0; --j)   // Loop_3
4      A[j] = i;                    // A write-stride
5    for (int k = 0; k < N; ++k)    // Loop_5
6      sum += A[k];                 // A read-stride
7  }
```

Figure 47: The stride from line 6 can be killed by either a point at line 2 or the stride at line 4.

Figure 47 illustrates this case. A stride is generated from the instruction at line 6 when `Loop_5` is being profiled. After finishing `Loop_5`, its `HistoryStrideTable` is merged into `Loop_1`'s `PendingStrideTable`. At this point, `Loop_1` knows the killed addresses from lines 2 and 4. Thus, the stride at line 6 can be killed by either (1) a random point write at line 2 or (2) a write stride at line 4. We detect such killed cases when the history strides are propagated to the outer loop. Detecting killed addresses is essentially identical to finding conflicts between strides and points. We use the same dependence-checking algorithm.

Interestingly, after processing killed addresses, a stride could be one of three cases: (1) a shrunk stride (the range of stride addresses is reduced), (2) two separate strides, or (3) complete elimination. For instance, a stride [4, 8, 12, 16] can be shortened by killed address 16. If a killed address is 8, the stride is divided.

5.3.8 Lossy Compression in Strides

Our stride-based algorithm essentially uses compression, which can be either *lossy* or *lossless*. If we only consider a strictly increasing or decreasing stride, SD^3 guarantees the perfect correctness of data-dependence profiling, which means SD^3 results are identical to the pairwise method results.

However, as discussed in Section 5.3.2, a stride like [10, 14, 18, 14, 18, 22, 18, 22, 26] is also considered a stride in our implementation. In this case, our stride format cannot perfectly record the original characteristic of the stream. We only remember two facts: (1) a stride of $10+4 \cdot n, (0 \leq n \leq 4)$ and (2) the total number of memory accesses in this stride is 9. The stride format cannot precisely remember the occurrence count of each memory address. Such lossy compression may cause slight errors when DYNAMIC-GCD calculates.

Suppose that this stride has a conflict at address 26. Address 26 is accessed only one time, but this information has been lost. For the compensation, we add a correction on the result of DYNAMIC-GCD by taking the average occurrence count of each reference: $\lceil 9/5 \rceil = 2$, the total accesses in the stride divided by the number of distinct addresses in the stride.

Nonetheless, such error does not noticeably affect the usefulness of our approach because we still guarantee the correctness of the existence and distance of data dependences. The error can only be observed in occurrence counts. From our definitions of the correctness in data-dependence profiling introduced in Section 2.4, our memory efficient algorithm implements a correct profiler although a strictly correct profiler will not be possible.

5.4 *Reducing Time Overhead by Parallelization*

5.4.1 Overview of the Algorithm

Figure 6 demonstrates that the runtime overhead of data-dependence profiling is extremely high. This is not surprising result as majority of the entire memory accesses are instrumented and complex dependence checking is performed on loop execution events. A typical method to reduce this time overhead would be to use sampling and approximation techniques [81]. However, we argued that such techniques are not suitable to assist parallelization based on a non-speculative parallel programming model. We need a *correct* data-dependence profiler.

We found that the memory efficient SD^3 could be effectively parallelized. We minimize the time overhead by *parallelizing* data-dependence profiling itself. In particular, we need to solve the following problems:

- Which parallelization model is most efficient?
- How do the stride algorithms work with parallelization?

5.4.2 A Hybrid Parallelization Model of SD^3

We first survey potential parallelization models of the profiler that implements Algorithm 3. Before the discussion, we need to explain the structure of our profiler although the details can be found in Section 5.5. Our profiler has two components: a tracer that instruments a program to be profiled and an analyzer that performs SD^3. Before parallelization, our profiler is composed of the following three steps:

1. Fetching *events* from a tracer (i.e., an instrumented program): An *event* is a unit of information that is transferred from a tracer to an analyzer. Events include (1) *memory events*: memory reference information such as effective address and PC, and (2) *loop events*: beginning/iteration/termination of a loop, which is essential to implement Algorithm 3.

2. Loop execution profiling and stride detection: An SD^3 analyzer collects statistics of loop execution (e.g., trip count), and train the stride detector on every memory instruction.

3. Data-dependence profiling: An analyzer runs Algorithm 3.

Our goal is to design an efficient parallelization model for the above steps. Three parallelization strategies would be candidates: (1) task-parallel, (2) pipeline, and (3) data-parallel. We survey these strategies:

- With the task-parallel strategy, several approaches could be possible. For instance, the profiler may spawn concurrent tasks for each loop. During a profile run, before a loop is executed, the profiler forks a task that profiles the loop. This is similar to the shadow profiler [95]. This approach is not easily applicable to the data-dependence profiling algorithm because it requires severe synchronization between tasks due to nested loops. Therefore, we do not take this approach.

  Note that a recent work called multi-slicing [153] takes a task-parallel strategy to accelerate the dependence profiling. Each task performs only memory accesses from the same allocation site, which is very similar to DAS-ID optimization. Because no dependences exist among these tasks, tasks run in parallel with minimal synchronization.

- Pipelining enables each step to be executed on a different core in parallel. We have three steps, but the third step, the data-dependence profiling, is the most time-consuming step. Although the third step determines the overall speedup, we still can hide computation latencies of the first (event fetch) and the second (stride detection) steps from pipelining. We use pipelining.

- Regarding the data-parallel method, first notice that SD^3 itself is embarrassingly parallel. Checking data dependences for a particular address requires only information on this address; no information from the other addresses is needed. We take a SPMD (Single Program Multiple Data) style to exploit this data-level parallelism. A set of *task* perform Algorithm 3 concurrently, but each task only processes a *subset* of the entire input. This data-parallel method is the most scalable one and does not require any synchronization except for the trivial final result reduction step. We also use this model.

From this survey, our solution is a hybrid model: we basically exploit pipelining, but the dependence profiling step, which is the longest, is further parallelized by a SPMD style. Figure 48 summarizes the parallelization model of SD^3. To obtain even higher speedup, we may exploit *multiple* machines (See details in Section 5.5.2). However, there are several issues for an efficient parallelization, which is now discussed.

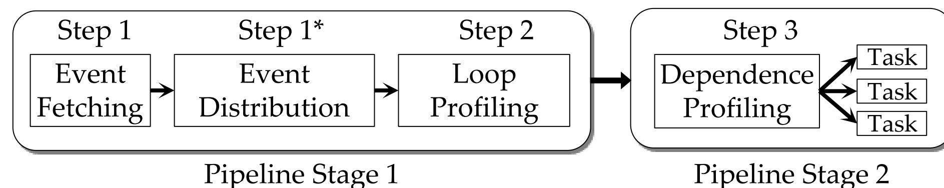


Figure 48: SD^3 exploits both pipelining (2-stage) and data-level parallelism. Step 1* is augmented for the data-level parallelization.

5.4.3 Event Distribution for Parallel Processing

Note that the *event distribution* step is introduced in the stage 1. Because of a SPMD-style parallelization at the pipeline stage 2, we need to prepare inputs for each task. Here, an input is a stream of events extracted from an instrumented program, which is transmitted by a tracer and will be consumed by an analyzer. However, the distribution is not simple partitioning of the entire events. Figure 49 illustrates

the process of the event distribution. First, memory events are distributed by some policies (we will discuss shortly later). By contrast, loop events are not distributed, instead must be *duplicated* for the *correctness* of the data-dependence profiling because the steps of Algorithm 3 are triggered on a loop event. Note that the correctness in this context means that results from a parallelized SD^3 must be the same with the results from the serial SD^3.

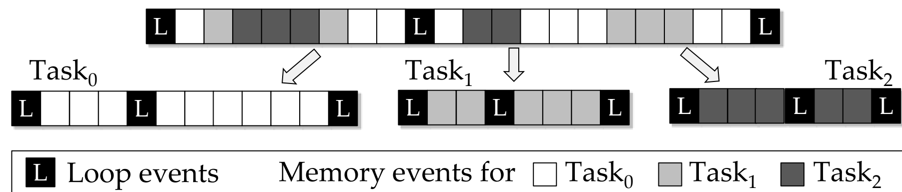


Figure 49: An example of the event distribution step with 3 tasks: Loop events are duplicated for all tasks while memory events are divided depending on the address-range size and the formula of Figure 51.

We discuss the issue of the distribution of the memory events. A simple and naive approach would be dividing the entire address space into N pieces, where N is the number of the parallel tasks, as shown in Figure 50.

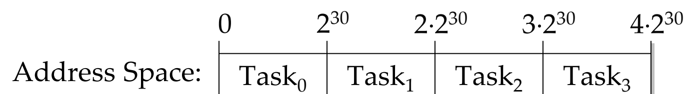


Figure 50: A naive address space division scheme in a 32-bit address space: Memory accesses are likely to be highly localized, which causes load imbalance and poor parallel speedup.

However, we soon discover this approach is not effective as the memory addresses are often localized, resulting in a poor parallel speedup. Instead, we divide the address space in an *interleaved* fashion for better speedup, as depicted in Figure 51. The entire address space is divided by every 2^k bytes, and each subset is mapped to M tasks in an interleaved way. Each task analyzes only the memory references from its own range. A thread scheduler then executes M tasks on N cores. Section 5.4.5 presents more details on choosing k and N.

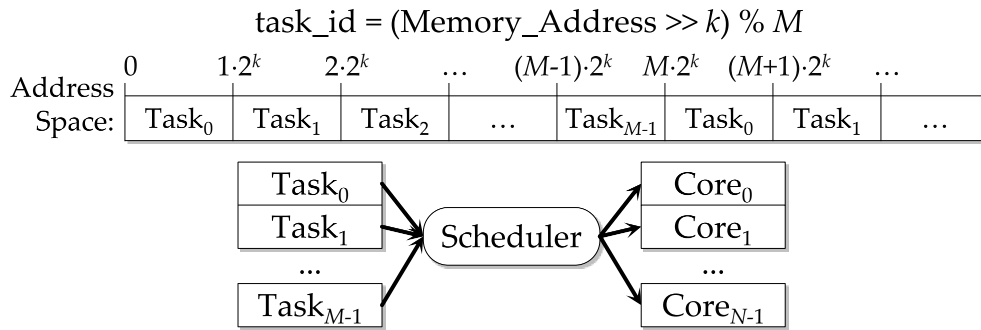


Figure 51: Data-parallel model of SD^3 with the address-range size of 2^k, M tasks, and N cores: Address space is divided in an interleaved way. The above formula is used to determine the corresponding task id for a memory address. In our experimentation, the address-range size is 128-byte ($k = 7$), and the number of tasks is the same as the number of cores ($M = N$).

5.4.4 Strides in Parallelized SD^3

Our stride-detection algorithm and Dynamic-GCD also need to be revised in parallelized SD^3 for the following reason. Stride patterns are detected by observing a stream of memory addresses. However, in the parallelized SD^3, each task can only observe memory addresses in its own address range. The problem is illustrated in Figure 52.

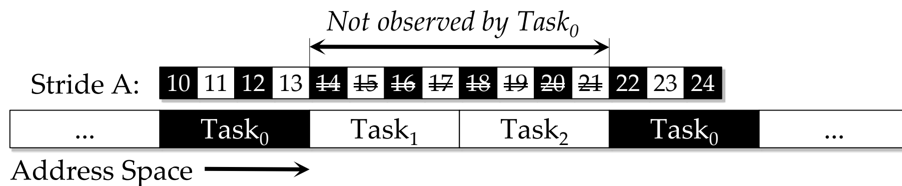


Figure 52: A single stride can be broken by interleaved address ranges. Stride A will be seen as two separate strides in $Task_0$ with the original stride-detection algorithm.

Here, the address space is divided for three tasks with a range size of 4 bytes. Suppose a stride A with the range of [10, 24] and the stride distance of 2. However, $Task_0$ can only see addresses in the ranges of [10, 14) and [22, 26). Therefore, $Task_0$ will conclude that there are two different strides at [10, 12] and [22, 24] instead of only one stride. These broken strides dramatically bloat the number of strides.

To solve this problem, the stride detector of a task assumes that any memory access pattern is possible in out-of-my-region so that broken strides can be combined into a single stride. In this example, the stride detector of $Task_0$ assumes that the following memory addresses are accessed: 14, 16, 18, and 20. Then, the detector will create a single stride. Even if the assumption is wrong, the correctness is not affected. To preserve the correctness, when performing Dynamic-GCD, SD^3 excludes the number of conflicts in out-of-my-region.

5.4.5 Details of the Data-Parallel Model

This chapter discusses two important design issues in our data-parallel model.

Choosing a good address-range size A key point in designing the data-parallel model is to obtain higher speedup via good load balancing. However, the division of the address space inherently makes a load unbalancing problem as memory accesses often show non-uniform locality. Obviously, having too small or too large address-rage size would worsen this problem. We use an interleaved division as discussed and then need to find a reasonably balanced address-range size.

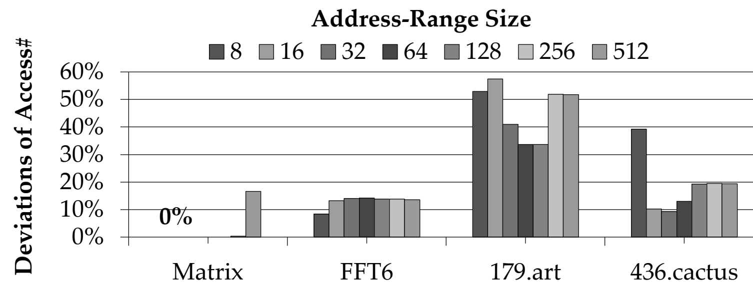


Figure 53: Deviations of total number of memory accesses of eight tasks by the address-rage sizes (in bytes). Lower is better.

According to our experiment, shown in Figure 53, as long as the range size is not too small or too large, address-range sizes from 64 to 256 bytes yield well-balanced workload distribution. In our implementation, we choose 128 bytes.

Choosing an optimal number of tasks Even when taking the interleaved approach, we cannot avoid the load unbalancing problem. To address this problem, we attempt to create sufficient tasks and employ the work-stealing scheduler [11], that is, exploiting fine-granularity task parallelism. At a glance, this approach would yield better speedup, but our data negated our hypothesis, as shown in Fig 54. We observed that no speedup was gained by this approach.

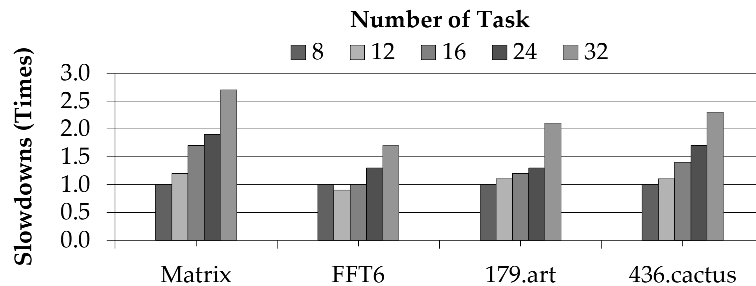


Figure 54: Having more tasks than the number of cores (on a eight-core machine) exhibits *slowdowns* in our hybrid parallelization model.

There are two reasons: (1) First, even if the quantity of the memory events is reduced, the number of stride may not be proportionally reduced. In Figure 52, despite the revised stride-detection algorithm, the total number of stride for all tasks is three; on a serial version of SD^3, the number of stride would have been one. Hence, having more tasks may increase the overhead of storing and handling strides, eventually resulting in poor speedup. (2) Second, the overhead of the event distribution would be significant as the number of tasks increase. Recall again that loop events are duplicated while memory events are distributed. This restriction makes the event distribution a complex and memory-intensive operation. On average, for SPEC 2006 with the train inputs, the ratio of the total size of loop events to the total size of memory events is 8%. Although the time overhead of processing a loop event is much lighter than that of a memory event, the overhead of transferring loop events could be serious as the number of tasks is increased.

Therefore, we let the number of tasks be identical to the number of cores.

Although the data-dependence profiling is embarrassingly parallel, the mentioned challenges, handling strides and distributing events, hinder an optimal workload distribution and an ideal speedup.

5.5 Implementation

Building a profiler that implements SD^3 has many implementation challenges. We discuss important issues in this section. We believe the discussion would be informative to the implementation of other program analysis tools that exploit instrumentation. We implement SD^3 on both Pin [85], a dynamic binary-level instrumentation toolkit, and LLVM [77], an open-source compiler framework. The SD^3 algorithm itself is orthogonal to the choice of instrumentation mechanisms. We first discuss the common issues regardless of Pin and LLVM. At the end of this section, we elaborate on issues specific to each instrumentation method.

5.5.1 Basic Architecture

Our profiler consists of a tracer and an analyzer, which is a typical producer and consumer architecture. We implement these two modules as two separate processes for several practical development conveniences. Section 4.4 presented the overview of this architecture. We briefly summarize again:

- Tracer: This instruments a program, captures runtime execution traces (i.e., memory events and loop events), and transfers the traces to the analyzer via shared memory. A tracer is built on either Pin or LLVM.
- Analyzer: This takes events from the tracer and performs the SD^3 algorithm, which is theoretically orthogonal to tracers.

Our profiler must be an *online tool*. Because the majority of loads and stores could be instrumented, generated traces could be extremely large, such as an order

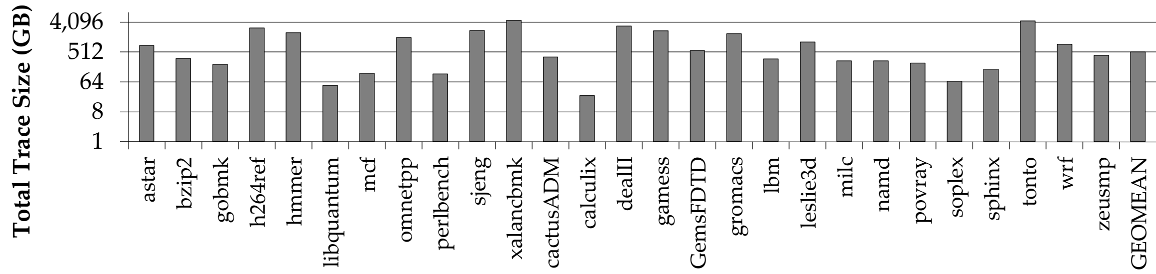


Figure 55: Total trace size for SPEC 2006 with the train inputs: The graph is in log scale. The geometric mean is 520 GB. xalancbmk and tonto show around 4.5 TB total trace size (train), which is far beyond a typical storage capacity.

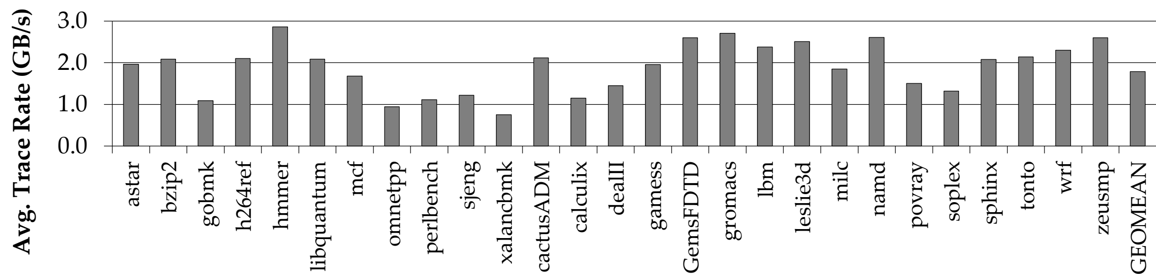


Figure 56: Average trace transfer rate between a tracer and an analyzer: The geometric mean for all benchmark is approximately 1.79 GB per second.

of 10 TB, especially when a reference input is profiled. Figure 55 demonstrates well the total size of the trace. Even with the train inputs, a tera bytes trace could be generated. We cannot simply use an offline approach. An example of such an offline approach would be storing and compressing events (e.g., using bzip) and then decompressing and analyzing the events. This approach is not effective at all because compressing/decompressing traces take a significant amount of time.

One concern of this online approach would be the overhead of inter-process communication. We observed that the average amount of event transfer rate between the two processes was up to 3 GB per second, as shown in Figure 56, which can be sufficiently handled by modern computers. The size of the execution event is 12 bytes, and events are transferred as uncompressed.

This separation of tracer and analyzer results in two significant benefits. First, the pipeline parallelism, explained in Section 5.4.2, is easily achieved. Second, we

can design the analyzer to be reused by different tracers. We separately implement tracers based on instrumentation mechanisms. We also define an abstracted communication layer between the single analyzer and multiple tracers, regardless of the choice of instrumentation tools. Finally, such separation eases debugging of the SD^3 algorithm.

5.5.2 Implementation of Analyzer

The analyzer first implements the data structures described in Section 5.3.4 and Algorithm 3, and we then parallelize using Intel Threading Building Block (TBB) [56]. To obtain even better parallelism, we extend our profiler to work on *multiple machines*, based on a MPI-like execution model [35]. The same tracer, analyzer, and application are running in parallel on multiple machines, but each machine has equally divided workload. This is a simple extension of our data-parallel model, but applies across different machines. Our profiler also profiles multithreaded applications and provides per-thread profiling results.

Importance of programming techniques Many programming techniques are extremely essential to improve the performance of the pairwise and SD^3 significantly, other than the key algorithms. Specialized data structures should be implemented rather than using general data structures in C++ STL. Customized memory allocation is also critical because the memory reference structures (`POINT` and `STRIDE`) are frequently allocated and removed.

False Positive and False Negative Issues We discuss the false positive (reported as having a dependence, but it was a false alarm) and false negative (no dependence reported, but it has a dependence) issues in data-dependence profiling.

First, false negatives can occur when not all code can be executed with a specific input. Section 5.6.4 discusses this problem. False positives can also occur if we take

a larger granularity in the memory instruction instrumentation, such as 8-byte or cache-line granularity rather than a byte granularity. Data-dependence profiling in the speculative multithreading domain can use a large granularity to minimize overhead, but this approach suffers from more false positive dependences [16]. In our implementation, the stride-based approach does not suffer from false positives. We correctly handle the size of the memory access (e.g., whether `char`, `int`, or `double`) in the stride-based data structures and DYNAMIC-GCD.

For the pairwise method in which hash tables are keyed by addresses, we always use 1-byte granularity. No false positives exist. However, it may have false negatives in a very unusual case, shown in Figure 57.

```
1  void*   raw   = malloc(1024);
2  double* data1 = (double*)raw;
3  char*   data2 = (char*  )raw;
4  data1[0] = 1.0;              // Writing 8 bytes
5  char t   = data2[2];         // Reading only part of data1[0]
```

Figure 57: A false negative case with 1-byte granularity: both `data1` and `data2` are the alias of `raw`, but their access types are different.

Even if there is an 8-byte write at line 4, the 1-byte granularity policy records only the first byte of the access. The read from line 5 results in a missing data dependence. We believe such a case is very unlikely to occur in well-written code. This problem can be resolved by our DAS-ID optimization. The heap accesses at line 4 and 5 both have the same dynamic-allocation site at line 1. We can easily detect aliased accesses by the different access types.

5.5.3 Implementation of Tracers

We first implemented SD^3 on Pin in our previous work [70]. We discuss challenges for Pin-based SD^3 and the motivations for LLVM-based SD^3.

Issues in a Pin-based Tracer A Pin-based tracer enables dependence profiling at the *dynamic* and *binary* levels. This approach broadens the applicability of the tool compared to a compiler and source-code-level approach. A dynamic instrumentation does not require a recompilation of a profile. This is a great benefit if the application does not have full source code that requires different and complex tool chains.

The downside of the Pin-based approach is that additional binary-level static analysis is needed to recover control flow graphs and loop structures, which is generally difficult to implement. For example, recovering indirect branches (e.g., jump tables for switch-case) and pinpointing the correct locations of loop entries and exits are challenges in binary-level analysis.

Regarding the instrumentation of loads and stores, an x86 binary executable typically has a lot of artifacts from push/pop in stacks and system function calls. Without eliminating such redundant loads and stores, results of a Pin-based profiler would have a lot of dependences that are not useful for the parallelization hints. Some loads and stores also do not need to be instrumented if their dependences can be identified at compile time, notably inductions and reductions. However, filtering such loads and stores selectively is also difficult with binary. These challenges motivate the use of an LLVM-based SD^3. Note that an alternative to direct binary-level static analysis would be using a x86 binary translator to LLVM IR and then exploiting the LLVM framework [74].

Issues in an LLVM-based Tracer Using compiler-based instrumentation such as LLVM may address the issues in the Pin-based tracer. LLVM also allows dynamic compilation and instrumentation, but we use LLVM as a static and source-level instrumentation toolkit. LLVM provides a very rich static-analysis infrastructure, including correct control flows and loop structures. Furthermore,

LLVM solves many challenges in binary-level instrumentation. For example, skipping inductions and reductions is relatively easy to implement since all the data-flow information is retained, unlike with binaries. Further static analysis may be performed before the dynamic profiling to decrease the profiling overhead [20]. For example, all memory loads and stores whose data dependences can be identified at compile time could be excluded as profiling candidates. Our implementation in this dissertation skips induction and reduction variables (both basic and derived ones) and some read-only accesses.

However, the greatest downside of using an LLVM-based tracer is that it requires recompilation. Recompiling an application with instrumentation code is not always easy. It sometimes requires modifications in compiler tool chains and compiler driver code. The analyzer needs some information from the instrumentation phase, such as a list of instrumented loops and memory instructions. As the instrumentation phase is separated from the runtime profiling, such information should be transferred via a persistent medium like a file.

5.6 *Experimentation Results*

This section presents experiment results that show the space and memory efficiency of the SD^3 profiler, followed by supplementary results regarding the input-sensitivity problem of the profiling and stride compression opportunities.

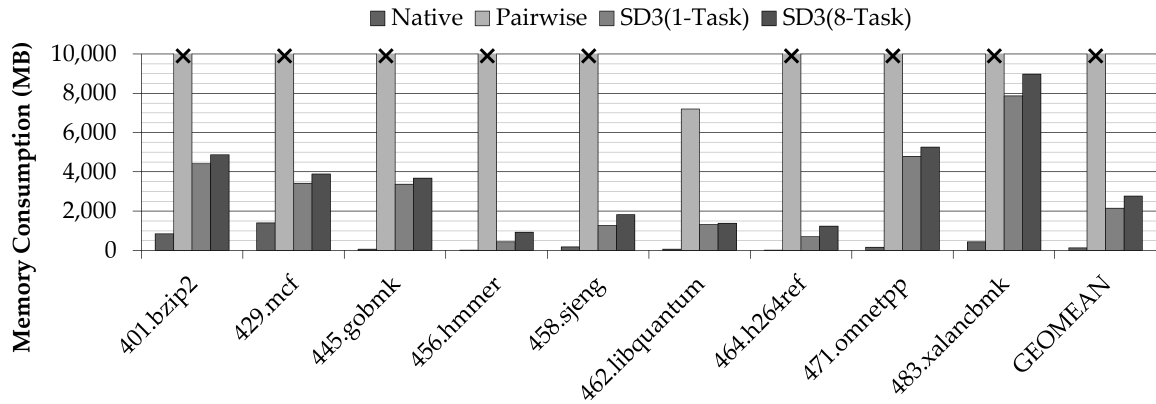
5.6.1 Experimentation Methodology

We use 22 (out of 29) SPEC CPU2006 benchmarks [134] to report runtime overhead by running the *entire* execution of benchmarks with the reference input. Seven SPEC benchmarks were not profiled successfully due to several implementation issues. Among the successfully profiled 22 benchmarks, we observed that a few loops from functions that parse input files caused runtime errors due to incorrect binary-level loop instrumentation. We excluded such erroneous loops.

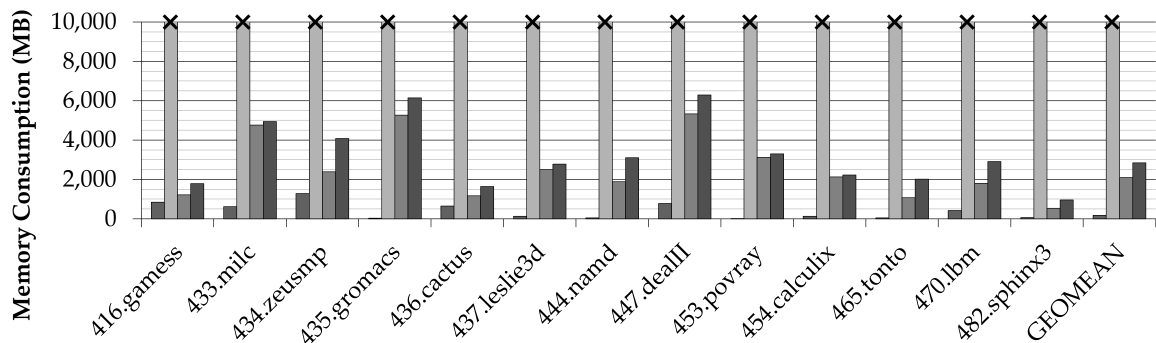
We should note that we intentionally used highly *optimized* binaries (`-O3` of Intel compilers) in the experimentation to reduce excessive profiling time overhead. It is true that a result from a highly optimized binary is virtually useless when we want to map the result to the source code by using debugging information. In practice, one should profile an unoptimized binary to obtain a human-readable result. However, the difference of native execution time between unoptimized and optimized binaries could be 10 times. The difference in memory overhead is not worse as the time overhead because memory accesses to local stacks are mostly optimized. The purpose of the experimentation is to measure the overhead, not to see actual dependence profiling results. Note that the profiling results in Chapter VII are from *unoptimized* binaries or source code, but with the reduced input sets such as train or test sets.

We instrument all memory loads and stores except for certain types of stack operations and corner cases. Our profiler collects details of data-dependence information as enumerated in Section 5.1. We profile the 20 hottest loops (based on the number of executed instructions) *and* their inner loops. For comparing the overhead, we use the pairwise method. We also use seven OmpSCR benchmarks [103] for the input-sensitivity problem.

Our experimental results were obtained on machines with 8-core (two sockets of quad-core processors) with Intel Hyper-Threading Technology (total 16 threads), and 16 GB main memory. The operating system is 64-bit Windows 7. Memory overhead is measured in terms of the peak physical memory footprint (similar to `rss`). For results of multiple machines, our profiler runs in parallel on multiple machines but only profiles distributed workloads. We then take the slowest time for calculating speedup. We experiment the pairwise method, a serial version of SD^3, parallelized SD^3 with 128-byte of the address-range size and (8-task on 8-core) and (32-task on 32-core).



(a) Profiling memory overhead of SPEC CPU2006 INT



(b) Profiling memory overhead of SPEC CPU2006 FP

Figure 58: Absolute memory overhead for SPEC 2006 with optimized (`-O3`) binaries and the reference inputs: 21 out of 22 benchmarks (× mark) need more than 12 GB in the pairwise method. The benchmarks natively consume 158 MB memory on average.

Currently, the LLVM implementation cannot instrument Fortran programs. The results in this thesis are from the Pin-based profiler, but the LLVM-based profiler shows a similar performance for programs written C/C++.

5.6.2 Memory Overhead of SD^3

Figure 58 shows the absolute memory overhead of SPEC 2006 with the reference inputs. The memory overhead includes everything: (1) native memory consumption of a benchmark, (2) instrumentation overhead, and (3) profiling overhead. Among the 22 benchmarks, 21 benchmarks cannot be profiled with the pairwise method on a 16 GB memory budget. Sixteen out of 22 benchmarks consumed more than

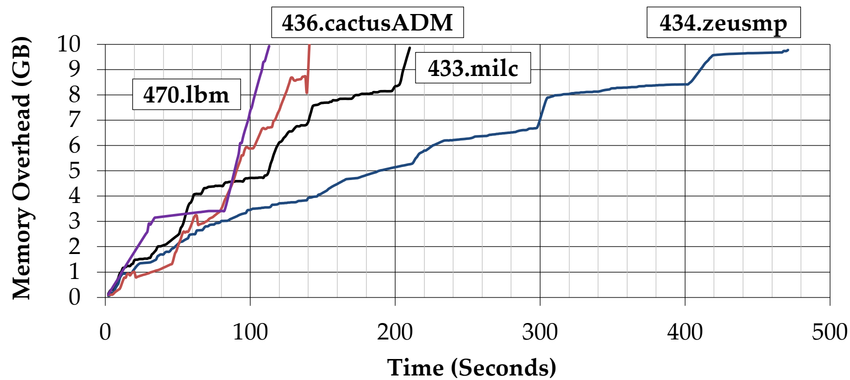


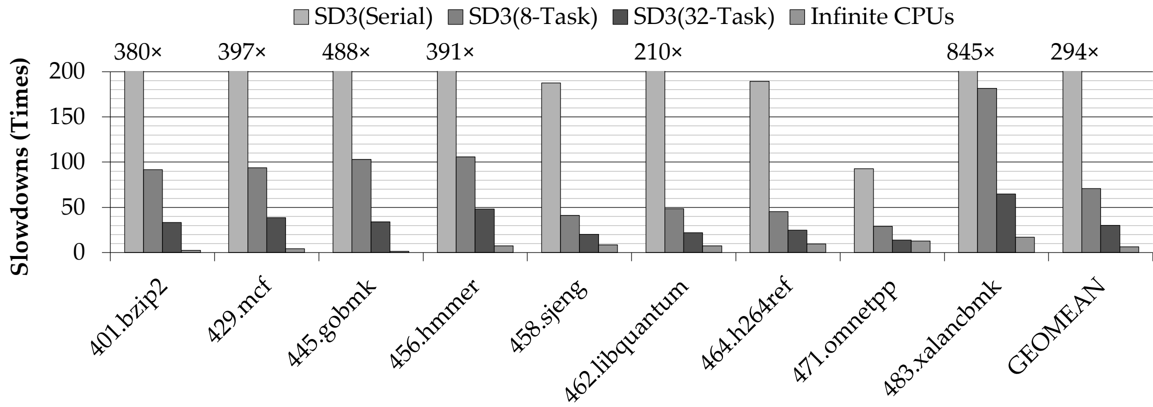
Figure 59: Memory overhead of the pairwise for four SPEC 2006 benchmarks.

12 GB even with the train inputs.

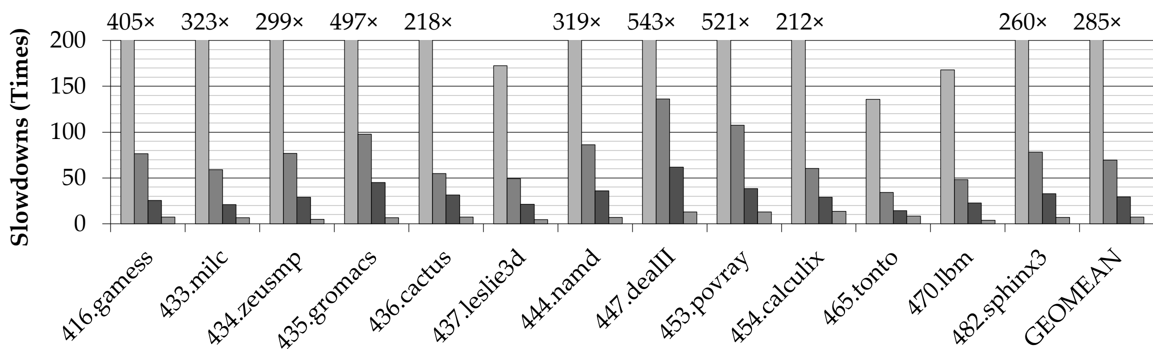
Figure 59 shows the memory consumption of the pairwise method in every second for 433.milc, 434.zeusmp, 435.lbm and 436.cactusADM. Within 500 seconds, these four benchmarks reached 10 GB memory consumption. We do not even know how much memory would be needed to complete the profiling with the pairwise method. We also tested 436.cactus and 470.lbm on a 24 GB machine, but still failed. Simply doubling memory size could not solve this problem.

However, SD^3 successfully profiled all the benchmarks on our 16 GB machine. For example, while both 416.gamess and 436.cactusADM demand 12+ GB in the pairwise method, SD^3 requires only 1.06 GB (just 1.26$\times$ of the native overhead) and 1.02 GB (1.58$\times$ overhead), respectively. The geometric mean of the memory consumption of SD^3 (1-task) is 2113 MB while the overhead of native programs is 158 MB. Although 483.xalancbmk needed more than 7 GB, we can conclude that the stride-based compression is very effective.

Parallelized SD^3 naturally consumes more memory than the serial version of SD^3, 2814 MB (8-task) compared to 2113 MB (1-task) on average. The main reason is that each task needs to maintain a copy of the information of the entire loops to remove synchronization. Furthermore, the number of total strides is generally increased compared to the serial version since each task maintains its own strides.



(a) Profiling overhead (slowdowns) of SPEC CPU2006 INT



(b) Profiling overhead (slowdowns) of SPEC CPU2006 FP

Figure 60: Slowdowns (against the native run) for SPEC 2006 with optimized (-O3) binaries and the reference inputs: From left to right, (1)SD^3 (one-task on one-core), (2) SD^3 (eight-task on eight-core), (3) SD^3 (32-task on 32-core), and (4) estimated slowdowns of infinite CPUs. The address-range size is 128 bytes. The geometric mean of native runtime is 488 seconds on Intel Core i7 3.0 GHz.

Despite of such negative effect of strides, SD^3 still reduces memory consumption significantly against the pairwise method.

5.6.3 Time Overhead of SD^3

The time overhead results of SD^3 are presented in Figure 60. For the speedup comparison, since only a few benchmarks can be profiled by the pairwise method successfully, we report speedups of parallelized SD^3 over the serial version of SD^3. The time overhead includes both instrumentation-time analysis and runtime profiling overhead. The instrumentation-time overhead, such as recovering loops,

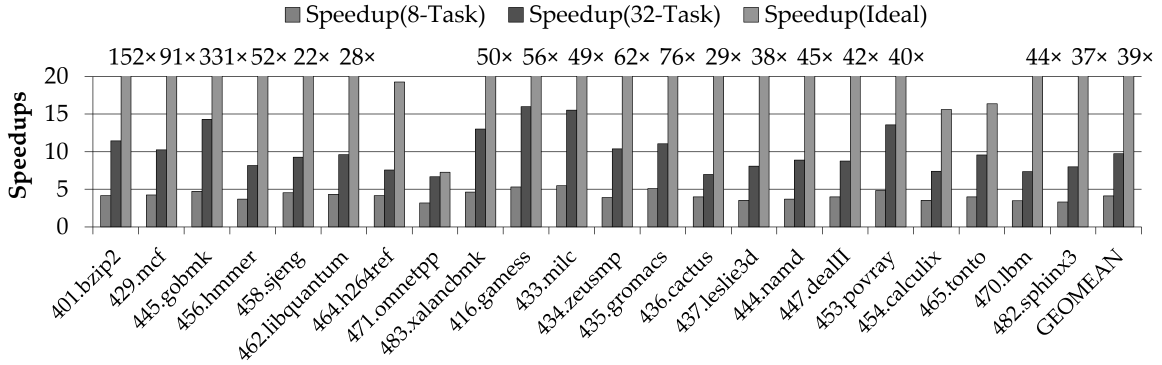


Figure 61: Speedups of SD^3 in SPEC 2006 benchmarks: We also calculate the ideal speedup when SD^3 is parallelized on infinite cores.

is quite small. For SPEC 2006, this overhead is only 1.3 seconds on average. The slowdowns are measured against the execution time of native programs. As discussed in Section 5.4.5, the number of tasks is the same as the number of cores in the experimentations.[8]

As shown in Figure 60, serial SD^3 shows a 289× slowdown on average, which is not surprising given the quantity of computations on every memory access and loop execution. We do an exhaustive profiling for the 20 hottest loops and their nested loops. The overhead could be improved by implementing better static analysis that allows us to skip instrumenting loads and stores that have proved not to make any data dependences.

When using eight tasks on eight cores, parallelized SD^3 shows a 70× slowdown on average, 29× and 181× in the best and worst cases, respectively. We also measure the speedup with four eight-core machines (total 32 cores). On 32 tasks with 32 cores, the average slowdown is 29×, and the best and worst cases are 13× and 64×, respectively, compared to the native execution time.

Calculating the speedups over the serial SD^3, we achieve 4.1× and 9.7×

[8]In fact, "running 8-task on 8-core" means that only the data-parallel part uses 8-task and 8-core. SD^3 requires one more thread for the event fetch, distribution, and stride profiling as shown in Figure 48. However, because most of time is consumed by the data-parallel stage, the overhead of having one additional thread than the number of physical core would not be problematic.

speedups on eight and 32 cores, respectively, shown in Figure 61. Although the data-dependence profiling stage is embarrassingly parallel, our speedup is lower than the ideal speedup. The first reason is that we have an inherent load unbalancing problem. The number of tasks is equal to the number of cores to minimize redundant loop handling and event distribution overhead. The address space is statically divided for each task, and there is no simple way to change this mapping dynamically. Second, with the stride-based approach, processing time for handling strides is not necessarily decreased in the parallelized SD^3.

We also estimate slowdowns with infinite CPUs. In such case, each CPU only observes conflicts from a *single* memory address, which is extremely light. Therefore, the ideal speedup is identical to the runtime overhead without the data-dependence profiling, which is the overhead of loop profiling plus tracing memory instructions. The ideal overhead is slightly greater than the overhead of loop profiling shown in Figure 28. Some benchmarks, like 483.xalancbmk and 454.calculix, show 17× and 14× slowdowns even without the data-dependence profiling. The large overhead of the loop profiling mainly comes from frequent loop invocation and deeply nested loops.

5.6.4 Input Sensitivity of Data-Dependence Profiling

One of the concerns of using a data-dependence profiler as a programming-assistance tool would be the input-sensitivity problem. In Section 2.2, we already have argued qualitatively the usefulness of dependence profiling even with this inherent weakness of dynamic analysis. To support this claim, we quantitatively measure the *similarity* of data-dependence profiling results from different inputs. A profiling result has a list of discovered dependence pairs (source and sink). We compare the discovered dependence pairs from a set of different inputs. We

compare only the top 20 hottest loops and ignore the frequency of the data-dependence pairs. We define similarity as follows, where R_i is the i-th result (a set of data-dependence pair):

$$\text{Similarity} = 1 - \sum_{i=1}^{N} \frac{\left|R_i - \bigcap_{k=1}^{N} R_k\right|}{|R_i|}$$

A similarity of 1 means all sets of results are exactly the same (no differences in the existence of discovered data-dependence pairs, but not frequencies). We first tested eight benchmarks in the OmpSCR [103] suite. All of them are small numerical programs, including FFT, LUReduction, and Mandelbrot. We tested them with three different input sets by changing the input data size or iteration count, but the input sets are sufficiently long enough to execute the majority of the source code. Our result shows that the data-dependence profiling results of OmpSCR were *not* changed by different input sets (i.e., Similarity is 1). The parallelizability prediction of OmpSCR by SD^3 has not been changed by the input sets we gave.

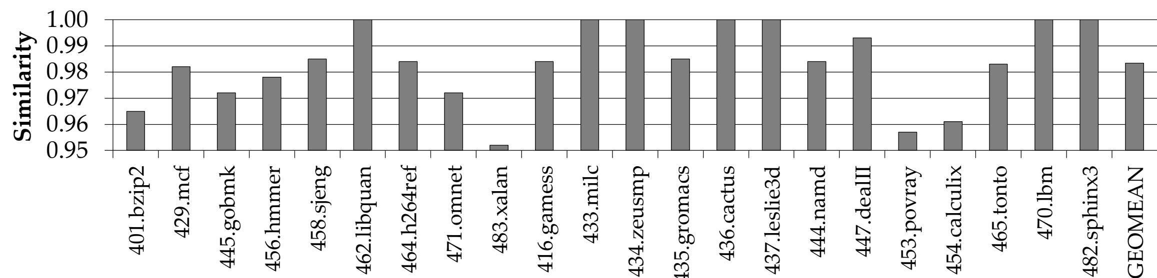


Figure 62: Similarity of the results from different inputs: 1.00 means all results were identical (not the frequencies of dependence pairs).

Figure 62 shows the similarity results of SPEC 2006, which are obtained from the reference and train input sets. Our data show that there are very high similarities (0.98 on average) in discovered dependence pairs. Recall that we compare the similarity for only frequently executed loops. Some benchmarks show a few differences (as low as 0.95), but we found that the differences were highly

correlated with the executed code coverage. In this comparison, we minimize x86-64-specific artifacts such as stack operations of prologue/epilogue of a function.

A related work also showed a similar result. Thies et al. used a dynamic analysis tool to find pipeline parallelism in streaming applications with annotated code [139]. Their results claimed that memory dependences between pipeline stages are highly stable and predictable over different inputs.

5.6.5 Opportunities for Stride Compression

We would like to see how many opportunities exist for stride compression. We easily expect that a regular and numerical program has a higher chance of stride compression than an integer and control-intensive program.

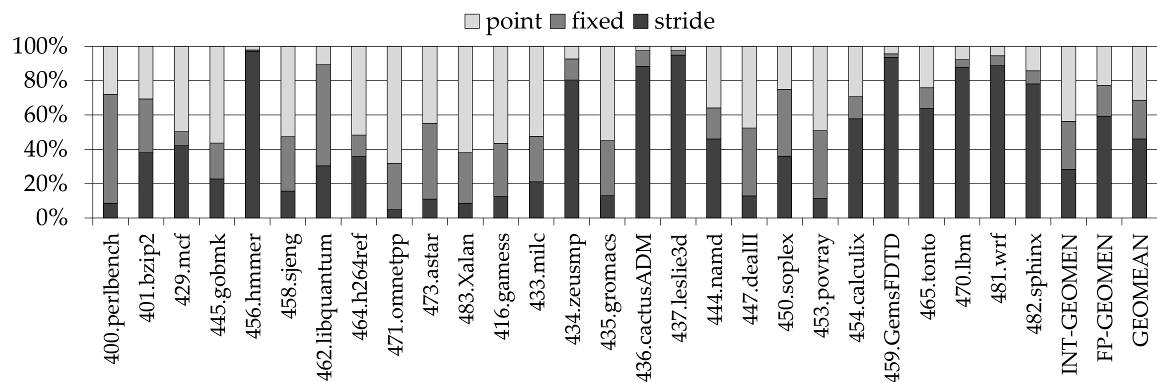


Figure 63: Classification of stride detection for SPEC 2006 benchmarks with the train inputs: (1) point: memory references with non-stride behavior, (2) fixed: memory references whose stride distance is zero, and (3) stride: memory references that can be expressed in an affined descriptor.

Figure 63 shows the distributions of the results from the stride detector when SD^3 profiles the entire memory accesses of SPEC 2006 with the train inputs. Whenever observing a memory access, our stride detector classifies the access as one of the three categories: (1) a part of a stride, (2) a fixed-location access, or (3) a point (i.e., non-stride). Clearly, SPEC FP benchmarks have a higher chance of stride compression than INT benchmarks: 59% versus 28%. Overall, 46% of the memory accesses are classified as strides. Some benchmarks have similar distributions of

stride and non-stride accesses such as 444.namd: 46% and 35%. Such fact supports why efficiently handling stride and point accesses simultaneously is important.

One should not translate the data in Figure 22 to the compression rates. We tried to obtain the actual compression rate by stride compaction, but the metric was not obvious and implementation was also challenging.

5.7 Summary of This Chapter

This chapter proposed a new efficient data-dependence profiling technique called SD^3. Although data-dependence profiling is an important technique for helping parallel programming, large memory and time overhead are the biggest challenge. SD^3 is the first solution that attacks both memory and time overhead at the same time. For the memory overhead, SD^3 not only reduces the overhead by compressing memory references that show stride behaviors, but also provides a new data-dependence checking algorithm with the stride format. SD^3 also presents several algorithms on handling the stride data structures. For the time overhead, SD^3 parallelizes the data-dependence profiling itself while keeping the effectiveness of the stride compression. SD^3 successfully profiles 22 SPEC CPU2006 benchmarks on a 16 GB machine.

CHAPTER VI

AN EFFECTIVE SPEEDUP PREDICTOR FROM SERIAL CODE

6.1 Introduction

In Sections 2.5 and 2.6, we thoroughly discussed why a dynamic-analysis-based speedup prediction tool is desirable to help the parallelization steps. This chapter presents *Parallel Prophet* , the speedup predictor of Prospector. Parallel Prophet predicts potential speedups over serial code while considering a specific parallel programming paradigm and memory bandwidth requirement in parallel code.

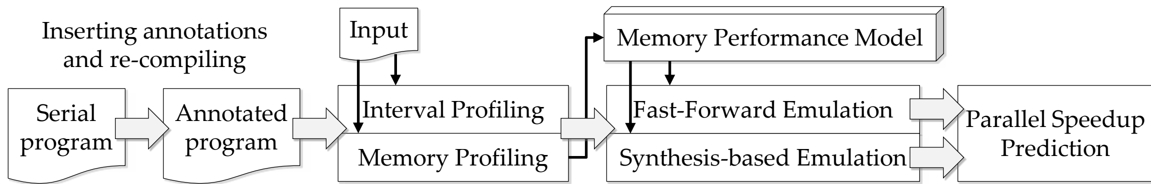


Figure 64: Work flow of Parallel Prophet.

An overview of Parallel Prophet is sketched in Figure 64. Parallel Prophet uses a *serial* program (or a serial portion of a multithreaded program) for the prediction. A programmer inserts *annotations* on the serial program. Annotations specify potentially parallel and protected regions. The annotation can be done by either a programmer's guess, or the post-analyzer of Prospector can assist. The annotated source code is recompiled and then profiled with an input. We perform two lightweight profiling: *interval profiling* and *memory profiling*. Interval profiling measures the elapsed time between a pair of annotations and then builds a *program tree*. Memory profiling uses a lightweight hardware performance counters to collect the cache miss ratio of the last-level cache and the number of executed instructions. Performance counter values are processed by the *memory performance*

model to compute *burden factors* that emulate performance degradation due to limited memory scalability. Burden factors are augmented on the program tree.

The program tree is consumed by the *emulators*. Parallel Prophet provides two emulation algorithms: (1) *fast-forwarding* emulation (the FF) and (2) *program synthesis-based* emulation (the synthesizer). Both emulation methods imitate the parallelized behavior of a given serial program from profiled data and annotations, thereby projecting potential speedups. Finally, speedups are reported for different parallelization knobs such as scheduling policies (e.g., dynamic and static), parallel programming paradigms (e.g., OpenMP and TBB), and core numbers.

Although Parallel Prophet shares the basic approach with Suitability[1], the synthesizer and the memory performance model are particularly new contributions comparing to the previous mechanisms. Section 2.6 presented several motivations why a new emulation algorithm and memory performance model are needed. The contributions of Parallel Prophet are summarized as follows:

1. We implement a baseline dynamic speedup predictor framework using annotation, interval profiling, and the fast-forward emulation algorithm. This framework not only enables the development of the synthesizer and memory performance model, but also serves a platform for future extensions such as supporting GPGPU-based programming models.

2. We propose a new dynamic emulation method, the program synthesis-based emulation. The synthesizer precisely and easily models broader parallel program patterns shown in Figures 11 and 12: (1) workload imbalance and arbitrary locking; (2) nested and recursive parallelism; (3) detailed semantics of a parallel programming paradigm (e.g., different scheduling policies in

[1]The basic approach of Parallel Prophet (annotation, interval profiling, and fast-forward) is loosely based on Suitability [51]. However, the implementation is totally separate and independent because Suitability is a proprietary product. We have built Parallel Prophet from the scratch. The details of Suitability have not been yet published.

Cilk Plus); and (4) effects due to intervention of operating systems in parallel execution, which is detailed in Section 6.3.2. The synthesizer uses real parallel constructs and measures an actual execution time on a real machine. Note that the FF supports only (1) and some limited patterns of (3). See the full comparisons of the prediction coverage in Table 4.

3. We introduce a memory performance model to predict the degradation of the parallel performance due to increased memory traffic based on analytical models with empirical coefficients from a specially crafted microbenchmark. The NPB-FT benchmark [64] in Figure 13 shows this case. Burden factors are introduced to connect our emulators and the memory performance model.

6.2 The Front-end of Parallel Prophet: Annotation and Profiling

This section presents the front-end of Parallel Prophet that consists of the annotation interface and the profiling phase. The outcome of the front-end is a program tree that records the execution history based on annotations and hardware performance counters.

6.2.1 Annotating Serial Code

Parallel Prophet takes a serial portion of a program. Because predicting speedups from serial code, Parallel Prophet does not profile multithreaded or parallelized code. In order to be profiled, an input serial program needs annotations.

Table 7 enumerates our annotations. This annotation interface allows programmers to specify potentially parallel and protected regions, which is similar to those of Suitability. A pair of PAR_TASK_* defines a *parallel task* that may be executed in parallel. A pair of PAR_SEC_* defines a *parallel section*, a container in which parallel tasks within the section are executed in parallel. A parallel section defines an

Table 7: Annotations in Parallel Prophet.

Interface and argument	Description
PAR_TASK_BEGIN(string task_name)	This task may be run in parallel.
PAR_TASK_END()	The end of a task.
PAR_SEC_BEGIN(string sec_name)	This parallel section begins.
PAR_SEC_END(bool nowait = true)	The end of the current section.
LOCK_BEGIN(int lock_id)	Try to acquire the given lock.
LOCK_END(int lock_id)	Release the owned lock.

implicit barrier at the end, but this barrier can be omitted via `nowait` parameter like OpenMP's `nowait`. `LOCK_*` annotations describe multiple critical sections. The parallel programming patterns in Figure 12 as well as loop-level parallelism can be represented by our annotation interface. Figure 65 is an example of annotated code. Suitability, as of 2012, does not support multiple locks and the `nowait` option.

6.2.2 Interval Profiling to Build a Program Tree

Parallel Prophet performs low-overhead *interval profiling* and *memory profiling* on an annotated program. The outcome of the profiling is a program tree that records program execution information directed by annotations. We explain how interval profiling builds a program tree with the example of Figure 65. This figure has a critical section and a nested parallel loop. The inner parallel loop may be executed on the condition of the parameter, `p3`. The inner loop has four iterations, and the length of each iteration is either 40 or 50 cycles.

The purpose of interval profiling is to collect *lengths* of all pairs of annotations. A length of an annotation pair is defined as either (1) the elapsed time or (2) the number of dynamically executed instructions, *between* a pair of annotations. In this implementation, we use the elapsed time (or cycles) as the unit of the interval profiling. The pros and cons of these two units are discussed in Section 6.5.2.

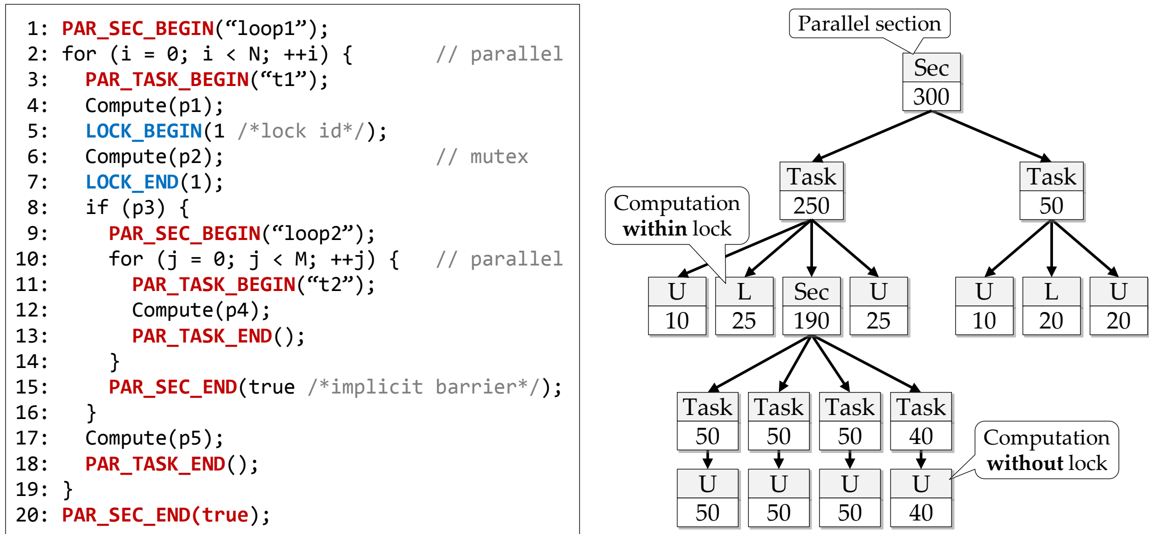


Figure 65: An example of an annotated program and the corresponding program tree: The numbers in the nodes are the elapsed cycles (time). U (unlocked) and L (locked) nodes represent computation *without* or *within* a lock, respectively. The root node is omitted. No memory profiling (burden factors) is shown in here.

While collecting lengths, Parallel Prophet concurrently builds a *program tree* that is a reflection of the intended parallel execution conveyed via annotations. The kinds of nodes in program trees are (1) `section`, (2) `task`, (3) `L`, (4) `U`, and (5) `root`. The first four nodes are corresponding to parallel section, parallel task, computation in a lock (locked), and computation without a lock (unlocked). These nodes are self-explanatory. The `root` node has a list of top-level parallel sections and `U` nodes representing serial sections.

The example shows that `PAR_TASK_*` annotations are placed at the loop iteration granularity. Each iteration of the parallel loops is recorded as a separate `Task` node. Therefore, the size of the tree could be extremely large when the trip count of a loop is large and deeply nested parallel loops exist. We solve this space overhead by compression, which is discussed in Section 6.5.3.

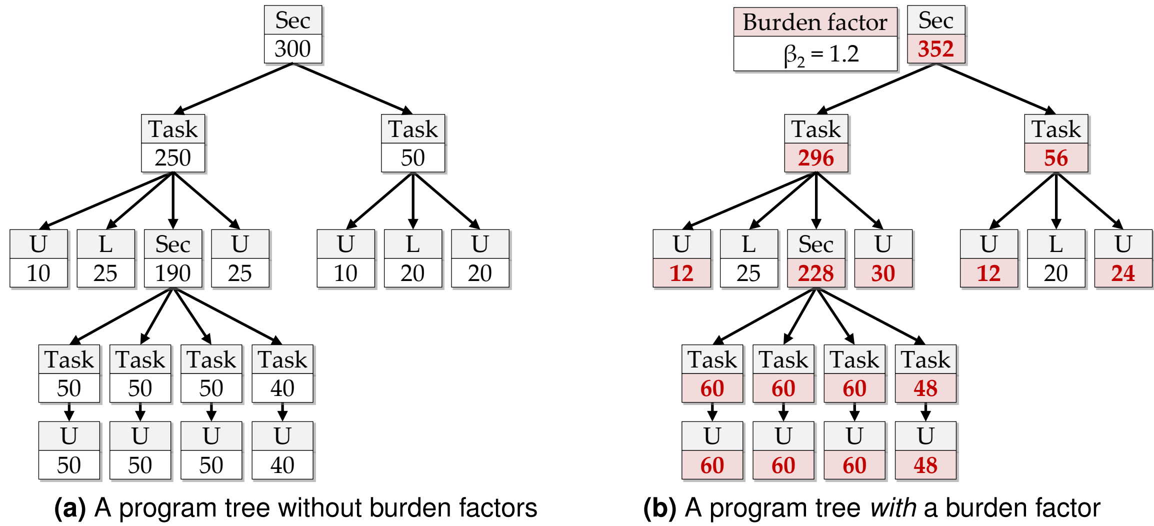
6.2.3 Memory Profiling: Burden Factors

In order to predict parallel performance degradation illustrated in Figure 13, we implement a *memory performance model*. The memory profiling collects information that will be consumed by the memory model. While interval profiling is being performed, we also do memory profiling.[2] Memory profiling uses hardware performance counter provided by processors because one of the important goals is to achieve low overhead such as a 2× slowdown in a typical case. In this thesis, we avoid heavy instruction-level instrumentation.

We collect the following three counters to predict expected memory bandwidth requirement when the program is parallelized: (1) the number of executed instructions, (2) the number of total accesses to the last-level cache (LLC), and (3) the number of LLC misses. Section 6.4 explains why these counter values are needed. Memory profiling may be performed for each loop iterations, or even each annotation pairs to achieve higher accuracy. However, such fine granularity does not necessarily bring higher accuracy because speedup predictions become more sensitive to small fluctuations. We collect profiling data per a dynamic instance of a top-level parallel section. If a parallel section is invoked multiple times, we profile per each invocation.

Once profiling data is collected, we immediately compute *burden factors*, β_t, from our memory performance model. We have a burden factor for a particular thread number (t). For example, β_2 is the factor for two cores. The computation of a single burden factor is trivially done in a constant time. Section 6.4 presents the definition of burden factor and the details of the memory performance model. This section we focus on how burden factors are used in a program tree.

[2]Due to implementation issues, we currently cannot perform interval and memory profiling simultaneously. Memory profiling is done in a separate phase. We believe simultaneous profiling is implementable with the probe mode of Pin [85].



(a) A program tree without burden factors **(b)** A program tree *with* a burden factor

Figure 66: A program tree without and with burden factors: A burden factor is multiplied to all U nodes. L nodes are excluded by our assumption. β_2 stands for a burden factor for two cores. 1.2 implies 20% penalty is predicted when this program is parallelized and executed on two cores.

An example of burden factors is visualized in Figure 66. This figure assumes that the top-level parallel section has a burden factor of 1.2 for two cores, $\beta_2 = 1.2$. This burden factor indicates that our memory performance model predicts that if the code would be parallelized on two cores, the durations of the nodes in this parallel section would be penalized by 20% due to increased memory traffic.

Burden factors provide an intuitive interface to connect the memory model and a program tree. To apply a burden factor to a program tree, we first multiply all `U` nodes in the section (including nested parallel loops, if any) with the factor. The parent `Task` and `Sec` nodes are then updated accordingly, as depicted in Figure 66 (b). We do not apply burden factors to `L` nodes based on the following assumption: when the computation of `L` node is performed, the other threads are either in blocked (due to the lock) or running without making significant memory traffic. Finally, when the emulators process a program tree, these burdened nodes will simulate the degraded parallel performance.

6.2.4 Summary of the Profiling

The procedure of the interval and memory profiling is as follows:

1. When an `*_BEGIN` annotation is observed, the current timestamp (or cycle stamp) and the annotation type are pushed on a stack.

   If the observed annotation is the beginning of a top-level section, we start to collect hardware performance counters.

2. On an `*_END` annotation, we check the kind of the current `END` (section, task, or lock) against the top of the stack. If they do not match, an error is reported. If they match, the elapsed time (or cycles) between the two annotations are calculated by subtracting the top of the stack from the current cycle stamp.

   If a top-level section finishes, the memory profiling is halted. Burden factors are computed by the memory performance model for a target machine.

   The stack is finally popped.

We must exclude the profiling overhead itself (other than real computation, such as annotations and instrumentation) when calculating the elapsed time in the step 2. This issue is further discussed in Section 6.5.2.

6.3 The Backend of Parallel Prophet: The Emulators

The final step of Parallel Prophet is the *emulation*, which literally emulates parallel execution to compute the projected speedup. Parallel Prophet provides two complementary emulators: (1) the fast-forward emulation (*the FF*) and (2) the program-synthesis-based emulation (*the synthesizer*). The basic idea of both emulations is to emulate parallel behavior by traversing a program tree. The FF emulates in an analytical form that does not require actual time measurement on a real multicore machine, except for some overhead coefficients. In contrast, the

synthesizer measures the actual speedup of an automatically generated parallel program on a real machine.

Once built the emulation logics, the overall speedup for a given number of threads, nt, is calculated as follows:

$$S_{nt} = \frac{\sum_i^N \textbf{Length}(sec_i) + \sum_i^M \textbf{Length}(U_i)}{\sum_i^N \textbf{EmulTopLevelParSec}(sec_i, nt) + \sum_i^M \textbf{Length}(U_i)} \,, \quad (3)$$

where sec_i is a top-level parallel section node, U_i is a top-level serial computation node, N is the total number of top-level parallel sections, and M is the total number of top-level serial nodes. **Length** returns the length of a node. **EmulTopLevelParSec** is an emulation logic that returns the estimated ticks (either time or cycles) for a given top-level parallel section and the thread number. Each emulation algorithm for a specific model implements this method.

We first discuss the FF and its challenges and then introduce the synthesizer.

6.3.1 Fast-Forwarding Emulation Algorithm

The FF traverses a program tree and calculates the expected elapsed time for each processor of an *ideal* parallel machine while considering several constraints and characteristics, such as the number of cores, scheduling policies, parallel programming paradigms, and parallel overhead. The FF predicts speedups for an arbitrary number of cores. Regarding scheduling policies, three static and dynamic schedulings of OpenMP are modeled. This thesis only considers OpenMP in the FF. Several overhead factors for parallel constructs are also modeled, such as `omp parallel for` and `omp critical`. Instead of writing a complex algorithm directly, we give two illustrations how the FF works in Figures 67 and 68. We explain important data structures as well.

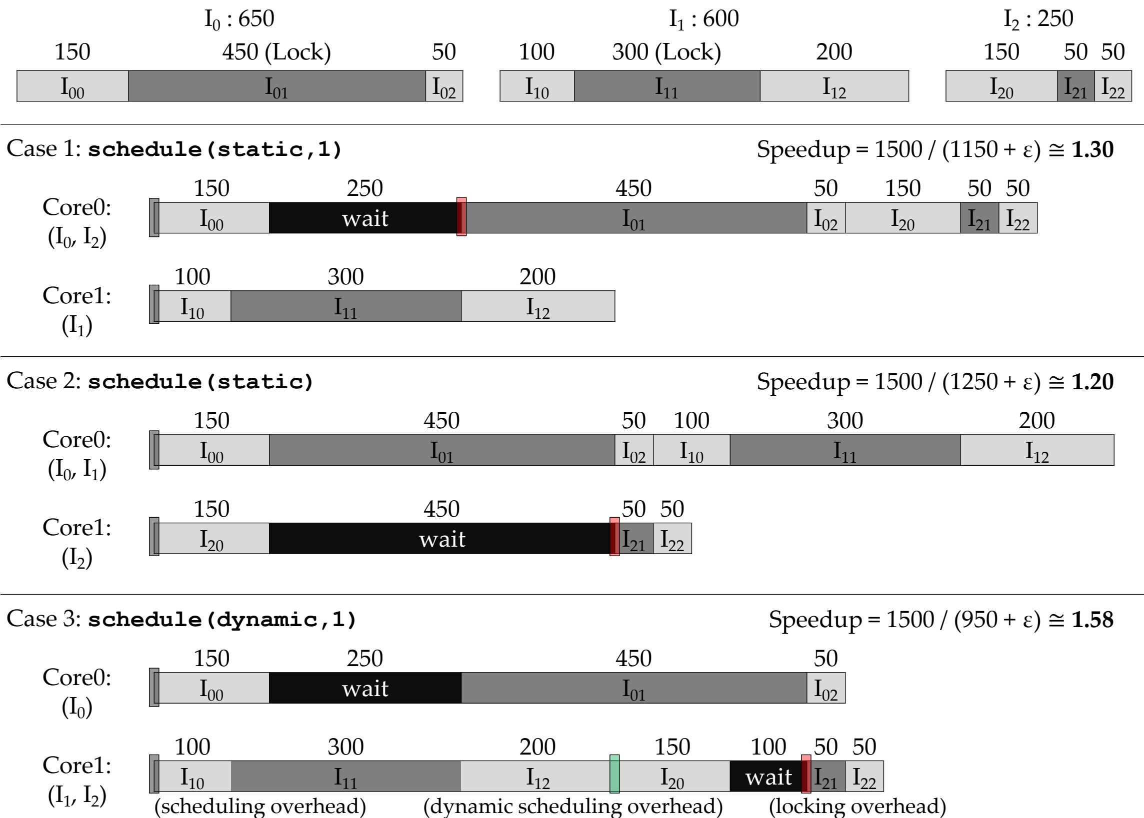


Figure 67: An example of the fast-forwarding method, where a loop with three unequal iterations and a lock is to be parallelized on a dual-core with OpenMP. The numbers above the boxes are the elapsed cycles. ϵ is the parallel overhead.

A simple parallel loop with a lock Figure 67 sketches how the FF computes the estimated parallel execution time on two cores while considering (1) workload imbalance; (2) a critical section; and (3) three scheduling policies of OpenMP: `(static,1)`, `(static)`, and `(dynamic,1)`. I_0, I_1, and I_2 are the first, second, and third iteration of the loop, respectively, and these are parallel tasks. C_0 and C_1 denote the first and second cores, respectively.

The FF allocates a ready-to-run *work* to an available *core* based on a scheduling policy and lock contention. A work represents either a `U` node, a `L` node, or a parallel-construct overhead. In Figure 67, I_{nk} denotes k-th work of n-th task. A core represents a logical machine state of a compute unit of an abstracted parallel

machine[3], including the state (idle, busy, waiting) and the elapsed time. Along with work and core, FF also has important data structures: a *scheduling context* to handle the semantics of a particular scheduling policy; a *lock* to maintain the status of a lock; and the *work queue* to manage running and waiting works.

We detail key steps when the FF emulates Figure 67:

(1) For the very first time, all works are ready and all cores are idle. The FF assigns the first work of the first task, I_{00}, to the first core, C_0. However, the first work for C_1 depends on the scheduling policy. Schedulings with the chunk size of one, `(static,1)` and `(dynamic,1)`, assign I_{10} to C_1. However, `(static)` statically maps {I_0, I_1} and {I_2} to C_0 and C_1, respectively. C_1 fetches I_{20} from the program tree. A scheduling context is responsible to decide the next task to run.

(2) The FF maintains the *work queue*, a global priority queue that holds running and waiting works. The queue is implemented as a minheap whose key is the completion time of a work. After the first step, two works, {<I_{00}, 0, 150, C_0>, <I_{10}, 0, 100, C_1>} or {<I_{00}, 0, 150, C_0>, <I_{20}, 0, 150, C_1>}, will be in the queue depending on scheduling policies. A work is a tuple of <*ID*, *start_time*, *completion_time*, *assigned_core*>.

(3) Now, all cores are busy. We then *fast-forward* to the next event. We fetch the work that will be finished in the earliest time from the work queue (i.e., deleting the minimum element in the minheap). We then set the fetched work as finished. Its completion time becomes the elapsed CPU time of the corresponding core. The fetched work can be deleted. When fetching the minimum element, two works may tie in the queue if the completion times are the same. We break the tie by the following priorities:

[3]The FF is based on simulation like an cycle-accurate architectural simulator.

- A running work has a precedence over a waiting (for a lock) work.
- If tied works are all busy, an older work (i.e., a work started earlier) has a precedence over the older one. As shown in the tuple format, we write the timestamp when a work is placed on the queue.

(4) Handling locks is also straightforward as the above steps. Consider the case of `(static,1)`. When C_1 completes I_{10}, the scheduling context returns I_{11} as the next work. I_{11} is inserted in the work queue. However, I_{11} is a `L` node. We update the corresponding *lock* data structure as *owned* by C_1. All cores are running, so do fast-forward. I_{00} is removed from the work queue and finished. Next, I_{01} is about to run. However, I_{01} needs a lock that is now owned by C_1. We simulate this block by inserting a work (a fake `U` node in Figure 69) whose completion time is the lock release time. When I_{11} finishes and releases the lock. We then continue the emulation as usual.

By this algorithm, the FF can precisely predict the differences in speedup due to scheduling policies: 1.20 with the default static scheduling, but 1.58 with `(dynamic,1)`. This example shows that precise modeling of scheduling policies is important for accurate prediction.

We also model the overhead of the OpenMP's parallel constructs. We use the benchmarks [13] and [22] to obtain the overhead factors. We then add the factors in the FF emulator when (1) a parallel loop is started and terminated, (2) an iteration is started, and (3) a critical section is acquired and released. The overhead is shown as thin shaded rectangles in Figure 67.

A case of nested loops Figure 68 is provided to illustrate a more complex case: handling nested parallel loops in the FF.

The parent loop (Loop1) has three iterations, and two iterations invoke a nested loop (LoopA and LoopB). All loops are to be parallelized. The order of the tree

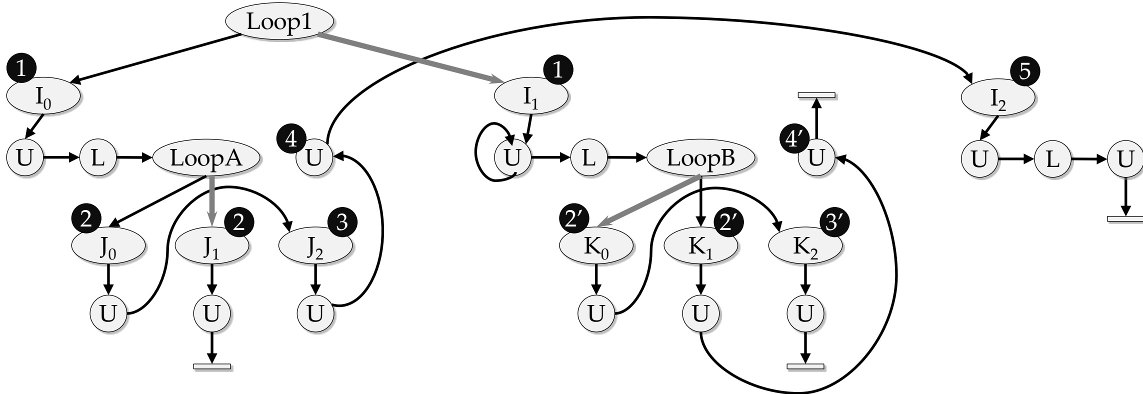


Figure 68: An example of nested parallel loops in the FF: Loop1, LoopA, and LoopB are parallel loops (parallel sections), while LoopA and LoopB are nested parallel loops. I_n are the parallel tasks of Loop1. J_n and $K_n$are the parallel tasks of the nested LoopA and LoopB, respectively. Black circles and arrows indicate an order of the tree traverse, assuming two cores and `(static,1)`. The same numbers in the black circles (see the gray arrows) mean parallel executions.

traverse is depicted as black circles in this figure. Although the exact total order of the traversing can be determined by the lengths of the all U and L nodes (works), we explain important steps:

(1) For the very first time, the scheduling context of `(static,1)` assigns I_0 and I_1 to C_0 and C_1, respectively. This means that the first and second iterations are executed in parallel (parallel execution denoted by the gray). The execution order of the `U` and `L` nodes of I_0 and I_1 are determined by their lengths as in Figure 67. We do not care the exact order in this example. After finishing I_0, `(static,1)` assigns I_2 (❹ $\rightarrow$ ❺) on C_0.

(2) For a nested loop, one would think calling the emulation algorithm recursively with a separate scheduling policy may work, while the work queue, locks, and cores should be shared. However, this guess does not work. For example, if the algorithm routine is called recursively to handle LoopA, there is no way to handle LoopB that can be run in parallel with LoopA. Although nested or recursive structure, we still need to process iteratively.

Function EmulateTopParLoop(section)

```
Initialize the machine state and create a scheduling context.
Assign initial parallel work of the top loop to CPUs.

while the work queue is not empty do
  Fetch the earliest work to be finished from the queue.
  Update the machine state: core tick and lock (if necessary).
  next_work ← Find next work from the scheduler of the work.
  if next_work is nil then
    if the finished work is from the top level loop then
      Set the core of this work as Idle.
    else
      Join to the spawning core.
    end if
  else
    if next_work is U node then
      Queue this work on the same core of the finished work.
    else if next_work is L node then
      if the lock is held then
        Queue a fake U node at the next lock releasing time.
      else
        Take the lock and queue this work.
      end if
    else if next_work is Loop node then
      Create a new scheduling context for this nested loop.
      Assign initial parallel work of the nested loop to cores.
    end if
  end if

return the longest core time.
```

Function Emulate the program

```
foreach core number and scheduling context
  estimated_par_tick ← 0
  foreach  node (top-level) in the program tree
    if node is U node then
      estimated_par_tick += node.interval
    else if node is Loop node then
      estimated_par_tick +=
        EmulateTopParLoop(node)
    end if
return estimated_par_tick;
```

Data Structures

Machine State
- Work queue: works on the fly.
- Elapsed ticks of cores.
- Next ready ticks of cores.
- States of cores: Busy, Waiting, and Idle.
- Owners of locks, and releasing ticks of locks.

Work
- Owner scheduling context.
- Assigned core Id.
- Started ticks and ticks to be finished.
- Time stamp when this work is queued.

Scheduling Context
- Nesting level.
- Upper level scheduling context.
- Arrays of next available task for each core.

Figure 69: Simplified emulation code and data structures of the FF.

(3) To solve the nested and recursive loops, we create a separate scheduling context for each parallel loop. This context remembers the parent context. Now, a work has the pointer to the scheduling context who dispatched. For example, when J_0 is inserted in the work queue, we associate the new scheduling context for LoopA. When J_0 finishes, we know that the current context. This context will return J_2. After J_2 finishes, there is no more work for LoopA. We switch to the parent context for Loop1.

We summarize the data structures and algorithm of the FF in Figure 69. The following section discusses the challenges and limitations.

6.3.2 Challenges and Limitations in Fast-Forwarding Method

The FF shows highly accurate prediction ability in many cases, but we have observed cases where the FF produces significant errors. First, if the parallel overhead is too high, such as frequent lock contention or frequent fork/join, the predictions could show low accuracy. Second, the FF does not model complex thread scheduling intervention and overhead by the operating system. This simplification may often lead to incorrect predictions for nested and recursive parallelism. Recall the examples of Figures 67 and 68. We did not model any preemptive scheduling and oversubscription (the number of worker thread is greater than the number of cores). We also did not model the concept of logical parallel tasks what many parallel programming paradigms have, which is an effective approach to handle nested and recursive parallelism as shown in Figures 15 and 16. We simply have a fixed 1:1 mapping between tasks and cores.

Figure 70 shows a counterexample of the FF. A simplified program tree of a 2-level nested parallel loop is shown. If the code is parallelized on a dual-core, the real speedup is 2 (=30/15). However, our initial FF implementation as well as Suitability give a speedup of 1.5 (=30/20). This is because both of them do not explicitly model the details of OS thread scheduling such as preemptive scheduling and oversubscription.

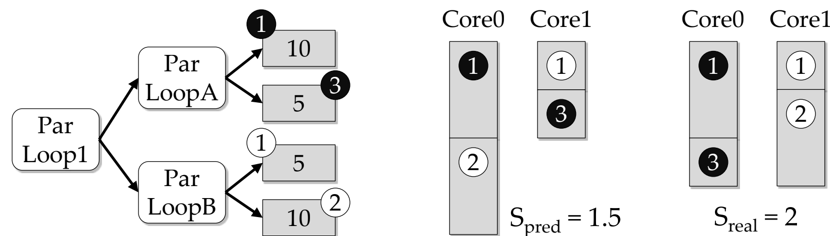


Figure 70: A two-level nested parallel loops where the FF and Suitability cannot predict precisely. The numbers in circles are the scheduling order. The numbers of the nodes are computation lengths. The predictions were 1.5, while the real speedup is 2.

We detail the reason of such inaccuracy in Figure 70. After finishing ①, the FF needs to assign ② (the second parallel task of ParLoopB) to a core. However, the FF simply maps ② to Core0 in a round-robin fashion. Furthermore, the work is assigned to a core in a non-preemptive way: A whole `U` (or `L`) node is assigned to a logical processor.

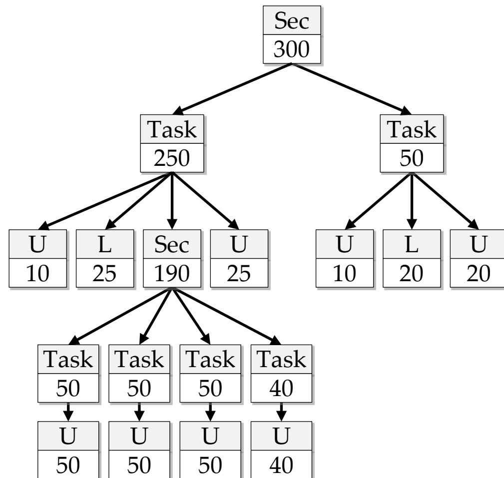
Of course, we can modify this behavior, but a simple ad-hoc modification cannot solve the problem entirely. We must model the thread scheduling of the operating system as well, which is very hard to implement and also takes a long time to emulate. Furthermore, both the FF and Suitability cannot predict the recursive or deeply nested parallelism, shown in Figure 12(b). To support these complex behaviors, we also need a sophisticated emulator that implements a task concept and work-stealing scheduler, as in Cilk Plus and TBB.

To address these challenges in the FF and Suitability, we provide an alternative and complementary way to emulate the parallel behavior, program synthesis-based emulation, or the synthesizer.

6.3.3 Program-Synthesis-Based Emulation Algorithm: Basic Idea

The synthesizer predicts speedups by measuring the *actual* speedups of *automatically generated parallel code* on a *real* machine. This synthesized parallel code does not contain any real computation what the original code has, but contains intended parallel behaviors, task lengths, and synchronizations. The original computation is replaced by a simple loop that spins for the duration of the computation. The biggest difference from the FF is that the synthesizer exploits real parallel constructs and real computers to estimate speedups.

Figure 71 illustrates the concept of the synthesizer. The program tree on the left is transformed to the right parallel code (using OpenMP) by the synthesizer. A real OpenMP parallel construct (`omp parallel for`) and a real lock are used. The




```
const int64_t p2[2]    = {25, 20};
const bool    p3[2]    = {true, false};
const int64_t p4[2][4] = {{50, 50, 50, 40},
                          { 0,  0,  0,  0}};
const int64_t p5[2]    = {25, 20};

#pragma omp parallel for
for (int i = 0; i < 2; ++i) {
  Delay(10);
  mutex.lock();
  Delay(p2[i]);
  mutex.unlock();
  if (p3[i]) {
    #pragma omp parallel for
    for (int j = 0; j < 4; ++j)
      Delay(p4[i][j]);
  }
  Delay(p5[i]);
}
```

Figure 71: Conceptually, the synthesizer generates a real parallel code from the program tree. However, such direct generation is difficult when a program has a lot of iterations and diverse task lengths. Hence, a general approach is needed.

lengths of works and branch predicates are represented as static const arrays. The original computation cannot be reproduced simply because we did not capture during the profiling phase, and running them in parallel does not guarantee any safe execution. Hence, we only emulate the computation durations by `Delay(tick)`. After synthesizing this artificial parallel code, we measure its actual running time on a target parallel machine.

This approach obviously solves the difficulties of the FF. Because we actually use a real parallel construct, unlike the FF, all the details of schedulings, complex parallel programming models, and parallel overhead are automatically and silently modeled. For example, we do not need to consider the order of traversing parallel tasks; the parallel library and operating system will automatically handle them. Hence, the counterexample in Figure 70 is perfectly emulated by the synthesizer. Note that the computation cannot be simply skipped like the FF. Doing so may repeat the same mistake of Figure 70.

Supporting other parallel programming paradigms can be done easily. For example, if one would like to emulate the behavior the dynamic scheduling of

Thread Building Block (TBB) [56], we just need to replace `omp parallel for` with TBB's `parallel_for` construct. In contrast, the FF must implement a precise emulation logic to model such scheduling behavior. Supporting beyond loop-level parallelism and mutex is also relatively easy. Pipelining, barrier, and condition variables could be simply modeled by the synthesizer.

The benefits of the synthesizer are clear, but we must address several challenges to make the synthesizer more practical.

6.3.4 Challenges in the Synthesizer

The example in Figure 71 is a small program where all the different work lengths are trivially declared as constant static arrays. However, when a program has large number of iterations, diverged branches, recursively parallel loops, and highly deviated work lengths, such simple code generation strategy will not work. Hence, we need a general synthesis algorithm that directly translates a program tree to a real parallel construct.

Figure 72 shows a synthesizer code for Cilk Plus. The code first sets the number of threads to be estimated by a Cilk Plus runtime function at line 22. Then, we start to measure the elapsed cycle by using `rdtsc()` (or other variants) for a given top-level parallel section, which is implemented in `EmulWorkerCilk`.

Recall the program tree in Figure 65: a parallel section (loop) has a list of children-parallel tasks (iteration). Each task then has a set of works. These children tasks are to be run concurrently in the parent section. To run the children tasks concurrently, we apply a parallel construct, `cilk_for`, on a section node (Line 3). For each concurrently running task, we now iterate all computation nodes (Line 4). The computations in `U` and `L` nodes are emulated by `Delay` in which a simple busy loop spins for a given time without affecting any caches and memory. The computation length is adjusted by the burden factor (Line 7), which is given by

```
1   void EmulWorkerCilk(NodeSec& sec, reducer& overhead)
2   {
3     cilk_for (int i = 0; i < sec.trip_count; ++i)     /*  actual  */
4       foreach (Node node in sec.par_tasks[i]) {       /* overhead */
5         overhead += OVERHEAD_ACCESS_NODE;             /* overhead */
6         if (node.kind == NODE_U)                      /* overhead */
7           Delay(node.length * sec.burden_factor);     /*  actual  */
8         else if (node.kind == NODE_L) {               /* overhead */
9           locks[node.lock_id]->lock();                /*  actual  */
10          Delay(node.length);                         /*  actual  */
11          locks[node.lock_id]->unlock();              /*  actual  */
12        } else if (node.kind == NODE_SEC) {           /* overhead */
13          overhead += OVERHEAD_RECURSIVE_CALL;        /* overhead */
14          EmulWorkerCilk(node, overhead);             /* overhead */
15        }
16      }
17    }
18  }
19
20  int64_t EmulTopLevelParSec(NodeParSec& sec, int nt)
21  {
22    __cilkrts_set_param("nworkers", nt);
23    reducer_opadd<int64_t> overhead;
24    int64_t gross_time = rdtsc();
25    EmulWorkerCilk(sec, overhead);
26    gross_time = rdtsc() - gross_time;
27    return gross_time - GetLongestLocalSum(overhead);
28  }
29
30  void Delay(int64_t delay_ticks)
31  {
32    int64_t end = rdtsc() + delay_ticks;
33    volatile int64_t t;
34    while (rdtsc() < end) ++t;
35  }
```

Figure 72: A general synthesizer for Cilk Plus: Pseudo code is shown. The comments `/* overhead */` indicate the tree-traverse overhead.

the memory performance model. For a `L` node, a real mutex is acquired and released, thereby emulating precise behavior and overhead of lock contention (Line 9 and 11). For a `Sec` node - nested and recursive parallelism that was extremely hard to emulate in the FF - is supported by a simple recursive call (Line 14). This recursive call is the exact same way that nested and recursive parallelism is actually executed.

Implementing a precise synthesizer has one another challenge: *tree-traversing overhead* must be subtracted. If the number of nodes is small, the traversing

overhead is negligible. However, a program tree could be very large (an order of GB), for example, if nested loops are parallelized and its nested loops are frequently invoked. In this case, the traversing overhead itself takes considerable time, resulting in significantly deviated estimations. To attack this problem, we measure two unit overhead of the tree traversing via a simple profiling on a given real machine: `OVERHEAD_ACCESS_NODE` and `OVERHEAD_RECURSIVE_CALL`, as shown in Figure 72. This approach is based on the observation that the unit overhead tends to be stable across various program trees. For example, we carefully measured the two unit overhead on our machine and obtained approximately 50 cycles for both unit overhead. We count the traversing overhead per each work thread (Line 5 and 13), and take the longest one as the final tree-traversing overhead (Line 27). Note that we used Cilk Plus' parallel reducer `reducer_opadd` because `overhead` has data races when counting the overhead.

However, we remain a future work for handling tree-traversing overhead. Our measurement of the unit overhead was only for single and two-level parallel loops. We noticed that the overhead was not stable when deeply nested (or recursive) parallel loops were profiled. One of the possible way is to use helper thread [84] to prefetch program tree's nodes in advance.

6.3.5 Summary of the Emulations

We do not claim that the synthesizer always outperforms the FF. There are several cases where the FF can be more useful. Table 8 compares these two approaches.

The synthesizer obviously requires a real machine, which may be a weak point. Programmers should run Parallel Prophet where they want to run the parallelized code in the future. The synthesizer does not predict speedups on arbitrary core numbers. The FF is a better choice if a programmer wants to see inherent scalability and diagnose the parallel performance. We believe that the FF mechanism can

Table 8: Comparison of the FF and the synthesizer.

Emulation Method	**The Fast-Forward Algorithm**	**Program-Synthesis Algorithm**
Characteristic	Analytical model	Experimental method
Time overhead	Mostly 1.1-3× slowdown In worst case, could be 30+× slowdown	Mostly 1.1-3× slowdown (Discussed more in Section 6.6.6)
Memory overhead	Moderate (by compression)	Moderate (by compression)
Accuracy	Accurate, except for some cases	Highly accurate
Target machine	An abstracted parallel machine	A real parallel machine
Supported models	Limited (hard to implement)	Easy to support a new model
Ideal for	To see inherent scalability and diagnose bottleneck; For a small program tree and 1-level parallel loops.	To see more realistic predictions; For a large program tree and nested and recursive parallel loops.

be easily transformed to a tool that identifies reasons of poor speedups. The biggest strength of the synthesizer is that it can easily extend to other parallel programming models and machines. One of the our future work is to investigate the synthesizer for GPGPU programming models, such as NVIDIA CUDA [100], OpenCL [67], Many Integrated Core (MIC) derived from Intel Larrabee [129] and Microsoft's C++ AMP [89].

Memory overhead for both emulations is the same as the same program tree is used. Regarding time overhead, there are a number of issues to be addressed. More discussion can be found in Section 6.6.6.

6.4 Lightweight Memory Performance Model

This chapter presents the memory performance model (MPM) of Parallel Prophet. Figure 13 showed the saturated speedups of the NPB-FT benchmark for which current Suitability and vanilla Parallel Prophet cannot predict. We design a MPM to predict such parallel performance saturation and degradation while achieving low overhead. From memory profiling results, MPM computes burden factors. Burden factors are then applied to a program node so that the emulators simulate slowdowns comparing to the serial computation time.

MPM is built on analytical models with coefficients that are measured on a

Table 9: Expected speedup classifications based on memory behavior. This thesis only considers the second row cases (✔) for lightweight predictions.

Variation of LLC miss/instruction	Observed memory traffic from serial code		
	Low	Moderate	Heavy
Par $\gg$ Serial	Likely scalable	Slowdown+	Slowdown++
✔ Par $\cong$ Serial	Scalable	Slowdown	Slowdown++
Par $\ll$ Serial	Scalable or super-linear	-	-

parallel machine where the speedup prediction will be made. As discussed in the memory profiling, we use hardware performance counters as Parallel Prophet particularly focuses on low overhead.

6.4.1 Assumptions

To build low-overhead MPM, we need a number of assumptions. We discuss and justify the assumptions.

• **Assumption 1**: We can separate the execution time of a program into two *disjoint* parts: (a) computation cost and (b) memory cost. Our model only predicts additional overhead on the memory cost due to parallelization. Assumption 2 discusses the computation cost.

• **Assumption 2**: We assume that the work is ideally divided among threads, similar to an SPMD (Single Program Multiple Data) style. Consequently, we assume that each thread has n-th of the total parallelizable work, where n is the number of threads. We also assume that no significant differences exist in branches, caches, and computations among threads. The tested subset of NPB benchmarks satisfies this assumption.

• **Assumption 3**: A simplified memory system is assumed: (a) We consider only the last-level cache(LLC) and the miss rate of LLC; (b) the latencies of memory read and write are the same; (c) SMT or hardware multi-threading is not considered; and (d) hardware prefetchers are disabled from the BIOS of our computers.

• **Assumption 4**: We consider only the case of when the LLC misses per instruction

(a kind of MPKI) does not significantly vary from serial to parallel. Table 9 shows the classifications of applications based on the trend of the LLC misses per instruction from serial to parallel: (1) increases, (2) does not vary significantly, and (3) decreases. However, in order to estimate LLC changes, an expensive memory profiling or cache simulation is necessary. The main goal of this thesis is to provide a lightweight profiling tool. The cases of the first and third rows in Table 9 will be investigated in our future work. Section 6.6.5 provides supporting profiling data.

• **Assumption 5**: The super-linear case is not considered. When LLC misses per instruction is less than 0.001 for a given top-level parallel section, the burden factor is 1, which is the minimum value in our model.

6.4.2 The Performance Model

Based on these assumptions, the execution time (in cycles) of an application may be represented as follows:

$$T = CPI \cdot N = CPI_{\$} \cdot N + \omega \cdot D, \tag{4}$$

where N is the number of all instructions, D is the number of memory (DRAM) accesses, $CPI_{\$}$ is the average cycles per instruction assuming *no* off-chip memory accesses (i.e., all data are fit into the CPU caches), and ω is the average CPU stall cycles for one memory access throughout the program execution. For example, suppose that throughout program execution, the total stalled cycles is 100 cycles, and 10 memory requests were observed. Then, ω is 10. $CPI_{\$}$ and ω represent

Table 10: Parameters in the performance model shown in Equation (4).

Name	Meaning	How to obtained
T	Total execution cycles	Memory profiling
N	Number of instructions	Memory profiling
D	Number of memory accesses	Memory profiling
ω	Average stall cycles for a memory access	Prediction
$CPI_{\$}$	Average cycles per instruction with perfect LLC	Equation (4)

the disjointed computation cost and memory cost, respectively, as in the first assumption. A similar usage of the $CPI_{\$}$ concept can be found in CPI_{exe} of [38].

6.4.3 The Definition of the Burden Factor

Although the application of burden factors was explained with a program tree, we did not yet provide the precise definition of the burden factor. The burden factor for a given thread number t, β_t, represents the performance degradation only due to the memory contention when a program is parallelized. We define β_t to be the ratio of T_t and T_t^i, where T_t is the execution time of a parallelized program on a real target machine with t cores[4], and T_t^i is the execution time on an *ideal* machine where the memory system is perfectly scalable:

$$\beta_t = \frac{T_t}{T_t^i} = \frac{CPI_t \cdot N_t}{CPI_t^i \cdot N_t^i} = \frac{CPI_{\$,t} \cdot N_t + \omega_t \cdot D_t}{CPI_{\$,t}^i \cdot N_t^i + \omega_t^i \cdot D_t^i}. \tag{5}$$

The subscript t to a number indicates that the number is obtained from one thread when a program is parallelized on t cores. We have assume that profiling values from different threads will not vary significantly. For simplicity, the subscript t can be omitted if t is one or the number is from a serial program.

Equation (5) can be further simplified by our assumptions:

- N_t, the number of instructions in parallel code for a thread, is obviously the same as N_t^i.
- In our SPMD parallelization assumption, we may safely consider that $N/N_t \simeq t$ and $D/D_t \simeq t$.
- Recall assumption 4: the LLC misses per instruction do not vary from serial to parallel. Hence, MPI (or D/N, LLC misses per instruction from a serial program) will be identical to MPI_t and MPI_t^i.

[4]We interchangeably use thread and core because we use 1:1 thread to core mapping.

Table 11: Parameters in the burden factor shown in Equations (5) and (6).

Name	Meaning	How to obtained
MPI	LLC misses per instruction (serial)	Memory Profiling (D/N)
ω_t	Average stall cycles for a memory access from a thread of the parallelized program	Prediction
β_t	Burden factor for t threads	Equation (6)

- We assumed that the computation cost, $CPI_{\$}$ will not change in parallel code. Hence, $CPI_{\$}$, $CPI_{\$,t}$, and $CPI^i_{\$,t}$ are all identical.
- Finally, the memory access penalty on a perfectly scalable parallel machine equals the penalty on a single core. Hence, ω^i_t and ω (= ω_1) is be identical.

As a result, the burden factor is finally expressed as:

$$\beta_t = \frac{CPI_{\$} + MPI_t \cdot \omega_t}{CPI_{\$} + MPI^i_t \cdot \omega} = \frac{CPI_{\$} + MPI \cdot \omega_t}{CPI_{\$} + MPI \cdot \omega}. \tag{6}$$

Our goal is to calculate β_t by analyzing only an annotated serial program. We obtain the T, N, D, and MPI profiling of a serial program with hardware performance counters. The remaining terms are now ω, ω_t, and $CPI_{\$}$. However, once we predict ω, Equation (4) computes $CPI_{\$}$. Therefore, the calculation of the burden factor is reduced to the estimation of ω and ω_t, which equals predicting ω_t.

6.4.4 Memory Access Overhead, w_t Prediction

Recall that ω_t is the additional cycles purely due to memory accesses, and we predict ω_t using only T, N, and D. Our assumption for the prediction is that ω_t would depend on memory access traffic. Hence, the prediction of ω_t can be done by two steps: (1) predict the total memory access traffic of t threads if a given program would be parallelized and would run on a target machine and (2) predict ω_t under this predicted memory traffic. This step is formulated as follows:

$$\delta_t = \Delta\,(\delta), \tag{7}$$

$$\omega_t = \Omega\,(\delta_t), \tag{8}$$

where δ is a *measured* memory traffic from a given serial program, δ_t is an *estimated* memory traffic of a *single* thread if the serial program would be parallelized and would run on t cores. The total expected memory traffic for a parallelized program is then $t \cdot \delta_t$. We obtain δ via the memory profiling:

$$\delta = \frac{1}{2^{20}} \cdot \frac{D \cdot S_{\$L}}{T \cdot f} \text{ (MB/s)}, \tag{9}$$

where $S_{\$L}$ is LLC line size (typically 64 bytes) and f is the CPU frequency (MHz). Based on the modelings at Equations (7) and (8), we find two empirical formulas, Δ and Ω from a specially designed microbenchmark.

Although the microbenchmark is detailed in the following section, we outline now. The microbenchmark can generate a spectrum of memory traffic by changing LLC hit ratio. This also controls the number of threads to measure the total memory traffic of multiple threads.

To determine Δ, we first measure the single-thread traffic from the microbenchmark. Then, we measure the total traffic when multiple threads are running together. We would observe both scalable and saturated cases depending on the traffic. We obtain Δ per each thread number on our machine. The machine specification is described in Section 6.6.1. We ignore small traffic less than 2 GB/s. For example, the obtained formula on our machine is ($\delta \geq 2000$ MB/s):

$$\begin{aligned}
\delta_2 &= (1.35 \cdot \delta + 1758) \;/\; 2, \\
\delta_4 &= (5756 \cdot \ln(\delta) - 38805) \;/\; 4, \\
\delta_8 &= (6143 \cdot \ln(\delta) - 39657) \;/\; 8, \\
\delta_{12} &= (6314 \cdot \ln(\delta) - 39621) \;/12.
\end{aligned} \tag{10}$$

Ω returns an expected memory access overhead for a given memory traffic. To find the relationship, we also use the same microbenchmark. We first measure total execution cycles by only changing LLC misses while L1 and L2 cache misses are fixed. We discuss the reason in the next section. However, we must subtract

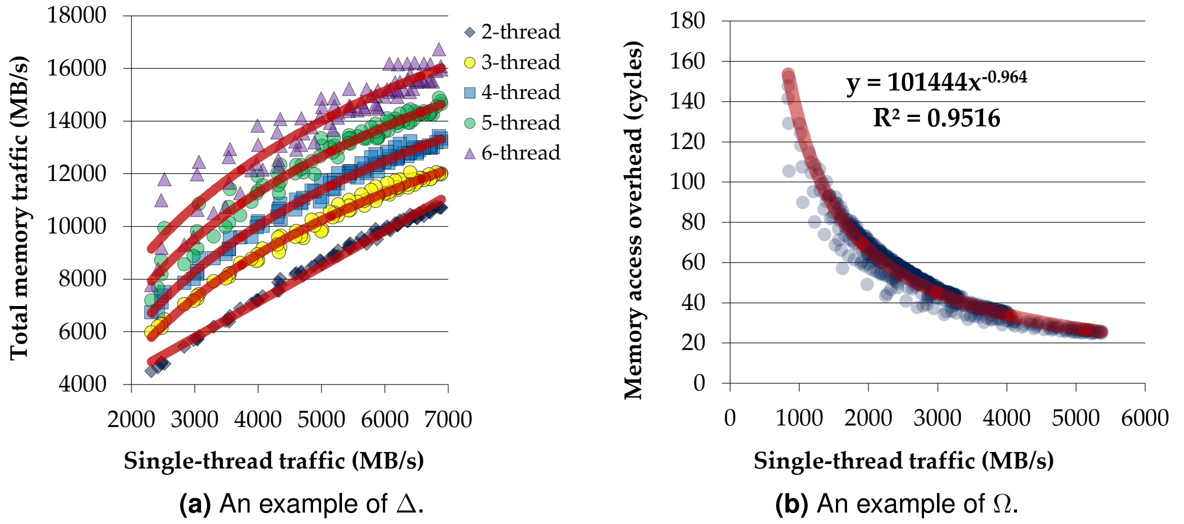


Figure 73: Examples of the coefficients of MPM: Equation (10) and Equation (11).

the computation cycles from the total execution cycles to expose the memory overhead, which is ω. This is done by measuring the total cycles when there are *no* LLC misses; however, again, L1 and L2 misses must be the same. We then compute the pure overhead due to memory accesses under an arbitrary memory traffic:

$$w_t = 101481 \cdot (\delta_t)^{-0.964}, \ (\delta_t \geq 2000 \text{ MB/s}). \tag{11}$$

Figure 73 visualizes these examples of Δ and Ω. Ω may return nonsensical numbers when δ_t is small. Recall assumption 5. In such a small δ range where the LLC misses per instruction is very small, β_t will be 1. In this implementation, we set the cutoff 2000 MB/s.

6.4.5 Details of the Microbenchmark for w_t Prediction

This section elaborates the details of the microbenchmark to obtain Δ and Ω on a particular machine. The design of the microbenchmark is particularly important in our memory model. We summarize the goals for the benchmark:

1. This generates a wide range of memory traffic, from small (less than 1000 MB/s) to large traffic (enough to saturate the memory controller);

2. This can measure not only the total execution cycles but also must subtract the memory access overhead from the total cycles to compute ω.

Controlling memory traffic For the first goal, we exploit two techniques: (1) tight pointer-chasing code to maximize the traffic and (2) changing the shape of linked-list chains to control LLC miss ratio.

```
1  void kernel(int64_t trip_count, CACHE_TEST* tester)
2  {
3    CELL* cur = tester->head
4    while (--trip_count)
5      cur = cur->next;
6  }
```

Figure 74: The kernel loop of the microbenchmark.

The main kernel loop has only linked-list traverse code as shown in Figure 74. This loop will be compiled (with optimization) to only three x86-64 instructions: (1) loop decrement, (2) pointer chasing (`mov (%rax), %rax`), and (3) a branch. `CELL` is a size of `void*` where only a single pointer can be placed. Each `CELL` is allocated on separate cache line. If `CELL` is linked to the next `CELL`, the loop will generate one memory request for every loop iteration (three instructions). Because there is a single memory instruction per the iteration, full LLC misses (cold miss) will be observed, resulting in maximized memory traffic. However, we soon discovered that the measured traffic was far below our expectation such as only 900 MB/s. The reason is because the throughput of the pointer-chasing code is limited by loop-carried flow dependences. No memory-level parallelism is exploited. To address this problem, we let *independent* linked-list chains together in the kernel loop. We also run the kernel in multiple threads. Figure 75 shows the improved kernel loop with the independent chains.

Notice that we used 14 independent chains. x86-64 has 16 general purpose registers. Except for the stack pointer (`rsp`) and a register to hold the `trip_count`,

```
1  void kernel(int64_t trip_count, CACHE_TEST* tester)
2  {
3    CELL* cur[14];
4    for (int i = 0; i < 14; ++i)
5      cur[i] = tester->heads[i];
6    while (--trip_count) {
7      cur[ 0] = cur[ 1]->next;
8      cur[ 1] = cur[ 2]->next;
9      ...
10     cur[13] = cur[13]->next;
11   }
12 }
```

Figure 75: The improved kernel loop of the microbenchmark: 14 independent CELL chains are traversed to maximize the memory traffic without register spill.

up to 14 registers can be used in the kernel loop to hold cur[k]. When using more than 14 independent chains, register spill occurs that will decrease the traffic volume. We verified that 14 independent chains created the maximum traffic.

The idea to control LLC miss ratio is sketched in Figure 76. We deliberately manipulate the shape of the pointer chains so that a portion of memory accesses are cache hits. LINE is a set of CELL to fit a single cache line. In a x86-64 machine, the sizes of CELL and LINE are 8 and 64 bytes, respectively. We also define REGION, a set of LINE that fits in the size of LLC.

Figure 76 shows the case of creating 50% cache miss ratio. First, until reaching the boundary of the LLC size, the accesses to CELL, ❶, ❷, and ❸, will be cache misses. We then rewind to the first LINE, but the *second* CELL. This CELL is still in

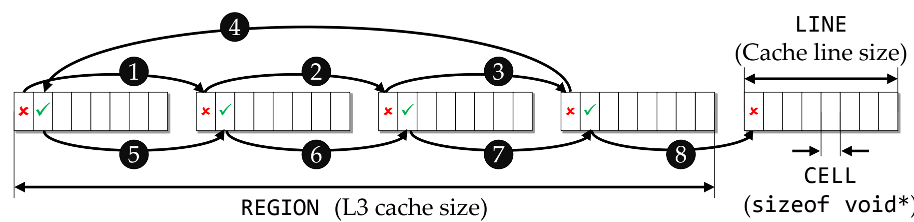


Figure 76: The microbenchmark controls LLC cache miss ratio by manipulating pointer chains. A set of 8-byte (sizeof void*) cells are linked with the repetition count of 1 so that 50% LLC miss ratio is created in the memory traffic. ✓ and ✗ indicates L3 cache hit and miss, respectively.

```
1  double miss = 0;
2  double hit  = 0;
3  // Initial repetition count
4  int repeat  = max(1, min(100 / target_miss_ratio, kNumCellInLine));
5  for (char* region = lb; region < ub; region += kRegionSize)
6  {
7    for (int i = 0; i < num_independent_chains; ++i)
8      BuildLinksInRegion(region + i*kCacheLineSize, repeat);
9
10   miss += kNumSegment;
11   hit  += kNumSegment * (repeat - 1);
12
13   // Adjust the repetition count based on current ratio.
14   double cur_miss_ratio = (miss * 100.) / (miss + hit);
15   if (cur_miss_ratio < target_miss_ratio)
16     repeat = max(1, repeat - 1);
17   else if (cur_miss_ratio > target_miss_ratio)
18     repeat = min(kNumCellInLine, repeat + 1);
19 }
```

Figure 77: Self-adjusting code to create an arbitrary cache miss ratio

LLC so that the access ❹ will be a cache hit. In turn, the accesses ❺, ❻, and ❼ will be also hits. When reaching the boundary again, now exit this region and move to the next `LINE`, as shown in ❽. We say that the repetition count is 1 in this case. The same pattern is then repeated. Finally, we can achieve 50% LLC miss ratio and control the memory traffic volume.

With the repetition count of 2, we can easily see $1/3 = 33.3\%$ miss ratio would be created. The maximum repetition count in x86-64 is 7, thereby $1/8 = 12.5\%$ is the minimum miss ratio that can be created by this technique. However, miss ratio under 10% provides only small memory traffic volume, which we do not care. Creating an arbitrary LLC miss ratio from 0.125 to 1.0 is straightforward. We allocate sufficient number of `REGION` and set different repetition count for each `REGION` by simple adjusting code shown in Figure 77.

Extracting the memory access overhead So far, we discussed several techniques to create controlled memory traffics so that we can obtain many sample point to build Δ, such as the example shown in Figure 73(a). However, building Ω imposes

one more challenge. We can easily measure the total cycles for a given arbitrary memory traffic, but ω_t is the average cycles (per instruction) *only* due to memory accesses. From our performance model in Equation (4), we can compute ω once we know $CPI_{\$}$, which is the execution time when all memory accesses are served by LLC. Hence, we measure the execution cycles by making all memory accesses to be LLC hits. We call this case as the reference point

One may think that the reference point can be obtained simply by a single `CELL` that points itself, making it perfect cache hits. However, this approach is not correct because L1 and L2 caches become hits. Observe that in Figure 76, regardless whether or not L3 hit, all memory accesses on `CELL` make L1 and L2 cache misses. On a L3 miss, it is obvious that L2 and L3 are also cache misses. However, even on a L3 hit due to the repetition, L1 and L2 still make cache misses because the size of `REGION` is sufficiently larger than L2 cache size. A hitting `LINE` in LLC is already evicted from L2 cache by capacity miss. Hence, to obtain the reference point correctly, we need to make 100% L3 hits, but full L1 and L2 misses. The naive approach using a single `CELL` will not account the cost of L1 and L2 accesses.

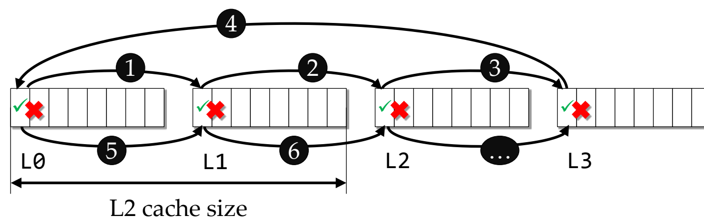
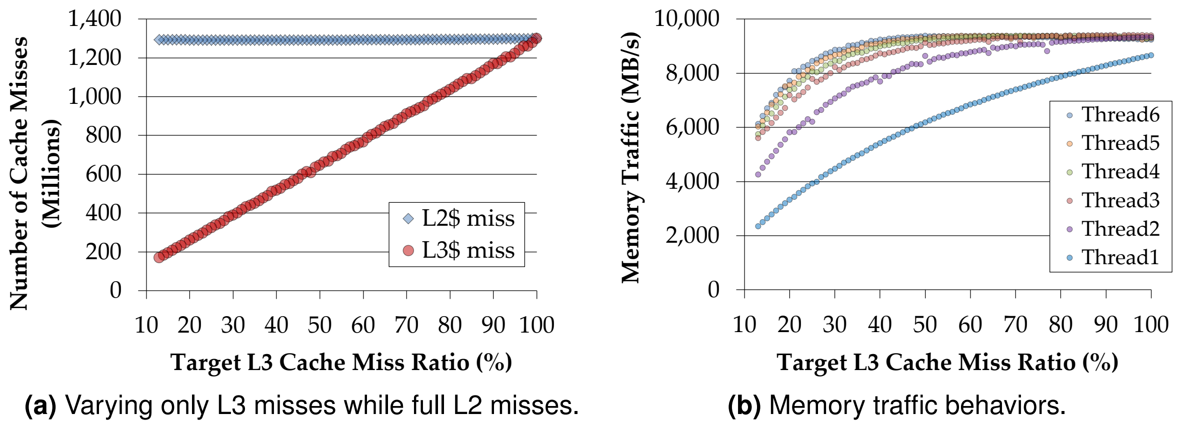


Figure 78: The shape of links to compute $CPI_{\$}$, the reference point: All accesses to L3 are cache hits (except the very first cold misses), but L1 and L2 always make cache misses. ✓ and ✖ indicates a L3 hit and a L2 miss, respectively.

Figure 78 has the correct shape of the linked list to obtain the reference point, i.e., to compute $CPI_{\$}$. After access ❸, all `LINE`s are in the L3 cache, but L2 has `L2` and `L3` while `L0` and `L1` are evicted. Hence, all accesses in the figure make L2 cache misses. We finally then can compute the pure memory access overhead, ω, for an arbitrary memory traffic, which yields Ω, for example, Figure 73(b).



(a) Varying only L3 misses while full L2 misses.

(b) Memory traffic behaviors.

Figure 79: Verifications of the microbenchmark.

Figure 79(a) verifies that our microbenchmark makes full L2 cache misses while L3 cache misses are varied from 12.5% to 100%. Figure 79(b) shows an example of various memory traffics by L3 cache miss ratio and the number of threads. We observe linearly increasing memory traffic on two threads, but soon memory traffics are saturated due to limited memory bandwidth. The machine of Figure 79(b) has only a single DRAM channel. The maximum bandwidth of the machine is approximately 10 GB/s while a dual DRAM channel provides 16 GB/s.

6.4.6 Summary of the Memory Performance Model

We summarize how to compute a burden factor for a particular thread number via the memory profiling and the presented memory performance model:

1. For a target parallel machine, run the microbenchmark to obtain Δ and Ω.

2. For a given serial program, perform the memory profiling per each dynamic instance of a top-level section. N, T, D, and MPI are collected. The memory traffic of the serial program, δ, is also computed by Equation (9).

3. Equation (7) estimates δ_t, which is followed by Equation (8) that estimates ω_t.

4. Finally, Equation (6) yields the burden factor.

6.5 *Implementation*

We implement Parallel Prophet as producer-consumer architecture similar to SD^3: (1) a tracer that performs instrumentation and profiling and (2) an analyzer that builds a program tree and emulates. Two processes communicate via shared memory. The tracer is based on Pin [85] and Intel Performance Counter Monitor [55]. The PAPI [47, 96] library can be used instead to retrieve hardware performance counters.

6.5.1 Implementation of Annotation System

Our annotations are implemented as C macros calling corresponding stub functions, which is detected by Pin. The approach is sketched in Figure 80.

A program to be profiled is compiled with the annotation interface file,

```
1  /* The singleton that holds pointers to stub functions */
2  static struct __HOOK_FUNC_TABLE {
3    __hook_lock_begin_t  pf_lock_begin;
4    __hook_lock_end_t    pf_lock_end;
5  } __func_table;
6
7  static void __init_func_table() {
8    /* Load DLL and fill __func_table */
9    ...
10 }
11
12 define PAR_LOCK_BEGIN(int lock_id) __func_table.pf_lock_begin(\
13   lock_id, "Lock@" __FILE__ "(" #__LINE__ ")")
14 define PAR_LOCK_END(int lock_id)   __func_table.pf_lock_end(\
15   lock_id)
```

(a) An excerpt from `annotate.h`: At the runtime, `__func_table` is first initialized.

```
1  void __cdecl __hook_lock_begin(const char* name, int lock_id)
2  { /* No operation */ }
3
4  void __cdecl __hook_lock_end(int lock_id)
5  { /* No operation */ }
```

(b) An excerpt from `annotate.c`, which is built as `annotate.dll` (`.so`): When building the DLL, we do not perform link-level optimization to prevent from collapsing separate stub functions.

Figure 80: An example of the annotation system.

annotate.h. The stub function, such as __hook_lock_begin, is provided by the hook, annotate_hook.dll (or .so). When an annotated program starts to run, a singleton declared from the annotation header file initializes the function pointer table, __func_table, by dynamically loading the hook DLL. In the runtime, when a stub function is called, Pin detects this moment and captures the arguments if any. The Pin tracer transfers the annotation name and arguments to the analyzer. The analyzer will then build a program tree.

We tried to use Pin's probe mode[5] that patches a binary on the loading time and executes natively. This mode is desirable because the overhead is minimized, and the interval and memory profiling can be performed simultaneously. However, due to some implementation difficulties, we currently use Pin's JIT mode.

6.5.2 Implementation of Interval Profiling

This section discusses several implementation issues for the interval profiling.

Using instruction count as the length unit Section 6.2.2 mentioned that the unit of the length in the interval profiling can be either (1) the number of dynamically executed instructions, or (2) the elapsed time, between a pair of annotation.

We initially attempted to use instruction numbers as the length. This approach provides a couple of strengths. First, we can precisely count the instruction count while excluding any profiling overhead, including annotation and instrumentation overhead. Second, the number of instructions is not affected by system fluctuation, which provides stable profiling and inherent characteristics of the program.

However, we must correctly consider the latencies of different instructions including cache and memory latencies. Such latencies of x86 processors are not

[5]Regarding the probe and JIT mode of Pin, please refer to the Pin manual: http://software.intel.com/sites/landingpage/pintool/docs/49306/Pin/html

typically provided by the vendors although unofficial information can be found.[6] We collected instruction mix and applied the best-known latencies information for instruction, cache, and memory. However, prediction results were not accurate as expected, especially for a program with diverse instruction mix and a lot of cache misses. Another downside of using instruction count is the time overhead. Collecting the instruction count requires at least basic-block-level instrumentation.

Using time as the length unit We alternatively use time (or cycles) as the unit in this work. Time-based length information automatically captures the latencies of all kinds of executed instructions inside annotations. However, two important challenges must be addressed in this approach: (1) obtaining precise timing information and (2) building a program tree while excluding profiling overhead.

First, to collect the length as precise as possible, we use `rdtsc()` that reads a processor's time stamp counter, which is the highest resolution. However, a couple of problems in using `rdtsc()` are known on modern multicore processors. To minimize the negative effects on time stamp counter, (1) we override the thread and processor affinity so that a thread is pinned on a specific processor; (2) dynamic frequency scaling such as Intel SpeedStep and Intel TurboBoost should be disabled. In Windows platform, we can use `QueryPerformanceCounter()` that hides these hardware complexity.

Second, building a program tree with the time unit requires a caution. When measuring the elapsed time, overhead due to annotations and profiling itself can be included. We should exclude such overhead to build a precise program tree, especially for a frequently annotated program. An example is shown in Figure 81. In the parallel task from line 3 to 18, there are annotation overhead of the lock and the nested loop. We can also observe profiling overhead due to the

[6] Torbjörn Granlund, "Instruction latencies and throughput for AMD and Intel x86 processors," `http://gmplib.org/~tege/x86-timing.pdf`, 2012-02-13.

```
1: PAR_SEC_BEGIN("loop1");
2: for (i = 0; i < N; ++i) {
3:   PAR_TASK_BEGIN("t1");
4:   Compute(p1);
5:   LOCK_BEGIN(1 /*lock id*/);
6:   Compute(p2);
7:   LOCK_END(1);
     blocked by the analyzer
8:   if (p3) {
9:     PAR_SEC_BEGIN("loop2");
10:    for (j = 0; j < M; ++j) {
11:      PAR_TASK_BEGIN("t2");
12:      Compute(p4);
13:      PAR_TASK_END();
14:    }
15:    PAR_SEC_END(true);
16:  }
17:  Compute(p5);
18:  PAR_TASK_END();
19: }
20: PAR_SEC_END(true);
```

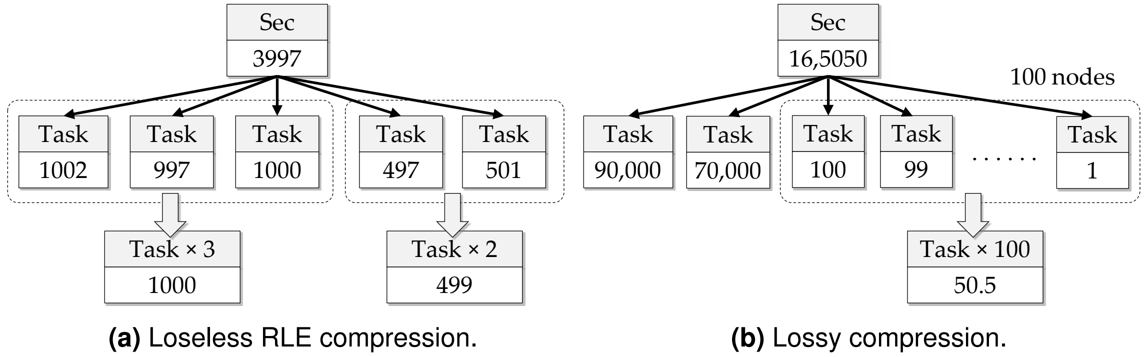
Figure 81: We must exclude overhead due to annotation and profiling. This figure shows such overhead between `PAR_TASK_BEGIN` and `PAR_TASK_END` at line 3 and 18, respectively. (1) Annotation overhead: between the task, there are lock and nested-loop annotations (lightly shared boxes). (2) Profiling overhead: due to the separate tracer and analyzer architecture, there could be a blocked time. The tracer attempted to transfer the event after `LOCK_END`, but the shared queue is still full because the analyzer is busy.

implementation of the tracer and analyzer. We track such annotation and interprocess communication overhead and subtract them from the gross time, which gives the *net* execution time.

6.5.3 Compression of the Program Tree

A program tree may consume large memory because all lengths of loop iterations are recorded. Many benchmarks in our experiments use moderate memory consumption (less than 500 MB). However, for example, IS in the NPB benchmark consumes approximately 10 GB to build a program tree. To solve this memory overhead problem, we apply compression techniques as depicted in Figure 82.

When lengths of loop iterations do not vary significantly, we perform *lossless* compression using a simple RLE(Run-length encoding) method with a simple dictionary-based algorithm. This simple lossless compression is quite effective for



(a) Loseless RLE compression.

(b) Lossy compression.

Figure 82: Two compression schemes for the memory overhead of program tree: Losess compression allows a small tolerance for easily fluctuating time-based lengths. Lossy compression's example shows 100 nodes are compressed to a single node whose length is the average of the compressed nodes.

many cases. The program tree of CG in NPB (with 'B' input) can be compressed into 950 MB from 13.5 GB (a 93% reduction). In our implementation, we allow 5 to 10% of variation to be considered as the same length. Recall that the unit of length is high-resolution cycle that tends to fluctuate.

Simple compressions may not feasible if iteration lengths are extremely hard to compress in a loseless way. As a last resort, we may use lossy compression. Fortunately, lossy compression was not needed in our experimentations. With lossless compression, 3 GB of memory is sufficient for all evaluated benchmarks in this thesis. We do not evaluate this scheme in this dissertation.

6.6 *Experimentation Results*

This section presents experimental results of Parallel Prophet. We present the results of the model verification. We then discuss the results of real benchmarks with our memory performance model (MPM). We also provide several experimental profiling results to support the assumptions of MPM. Finally, discussions regarding the time overhead and the limitations follows.

6.6.1 Experimentation Methodologies

The machine configuration we used for the experiments is summarized in Table 12. Disabling hardware prefetchers is critical for the evaluation of MPM. We do not currently model the effects of prefetchers. SpeedStep (DVFS) and TurboBoost (automatic overclocking) are also disabled to obtain stable single-thread results.

We evaluate eight benchmarks in OmpSCR [103] and NPB [64]. When running a multithreaded benchmark, we manually set process or thread affinity as shown in Table 13, which maximally utilizes two separate LLCs. The affinity can be set by either APIs such as `pthread_setaffinity_np` or environment variable of an OpenMP runtime such as `KMP_AFFINITY` in Intel's OpenMP.

6.6.2 Validation of the Prediction Model

We first conduct validation experiments to verify the prediction accuracy of Parallel Prophet without considering memory and caches. This step is important to quantify the correctness and diagnose the problems of Parallel Prophet before predicting realistic programs with our memory performance model.

We use randomly generated samples from two serial program patterns: `Test1` and `Test2`, shown in Figure 83. `Test1` exhibits (1) load imbalance, (2) critical

Table 12: System configuration.

CPU model	Intel Xeon E5645 @ 2.4 GHz
Architecture	Westmere (32nm)
L3 cache size	Shared 12 MB per socket
Cores/Threads	6/6 per socket (Hyper-threading is disabled)
Number of sockets	2 (Total 12 cores and 12 threads)
Main memory	2 × 4 GB per socket (Total 16 GB)
Memory channel	Dual-channel per socket
Memory configuration	SMP mode (No NUMA)
HW features	HW prefetchers, TurboBoost, SpeedStep are all disabled
Operating System	Windows 7 64-bit
Compilers	Intel C/C++ Compiler 12.0

Table 13: CPU affinity configurations for the experimentations on two six-core processors: ○ means an idle core, and ● indicates a thread is pinned.

Thread number	Socket 1	Socket 2
1	●○○○○○	○○○○○○
2	●○○○○○	●○○○○○
4	●●○○○○	●●○○○○
8	●●●●○○	●●●●○○
12	●●●●●●	●●●●●●

sections whose contentions and lengths are arbitrary, and (3) a high parallel overhead case: high lock contention. `Test2` additionally shows (1) a high parallel overhead case: frequent inner loop parallelism and (2) nested parallelism, as well as all the characteristics of `Test1`. The function `ComputeOverhead` in the code generates various workload patterns, from a randomly distributed workload to a regular form of workload, or a mix of several cases.

We generate 300 samples per each test case by randomly selecting the parameters. These randomly generated programs are annotated and predicted by Parallel Prophet. Predicted parallel model is a program parallelized by OpenMP running on 2, 4, 6, 8, 10, and 12 cores. Three OpenMP's scheduling policies are predicted: `(static,1)`, `(static)`, and `(dynamic,1)`. To verify the correctness, we parallelize the tester programs by OpenMP and measure real speedups. We finally plot the predicted speedups versus the real speedups to visualize the accuracy. The more dots closer to $y = x$, the more accurate result. Some of the results for 8 and 12 cores are shown in Figure 84.

The fast-forwarding emulation achieved high accuracy for `Test1`, shown in Figure 84 (a) and (b); the average error ratio is less than 4% and the maximum error ratio is 23% for a case of predicting 12-core.

Figure 84 (c) and (d) show the results of `Test2` with the FF. The average error ratio is 7% and the maximum was 68%, which are significantly higher than the results of `Test1`. Among the three schedulings, `(static)` shows more

```
1  // To be parallelize by OpenMP.
2  for (int64_t i = 0; i < i_max; ++i) {
3    overhead = ComputeOverhead(i, i_max, M, m, s);
4    FakeDelay(overhead * ratio_delay_1);
5    if (do_lock1) {
6      // To be protected.
7      FakeDelay(overhead * ratio_delay_lock_1);
8    }
9    FakeDelay(overhead * ratio_delay_2);
10   if (do_lock2) {
11     // To be protected.
12     FakeDelay(overhead * ratio_delay_lock_2);
13   }
14   FakeDelay(overhead * ratio_delay_3);
15 }
```

(a) `Test1`: workload imbalance + locks + high lock contention.

```
1  // To be parallelize by OpenMP.
2  for (int64_t k = 0; k < k_max; ++i) {
3    overhead = ComputeOverhead(k, k_max, M, m, s);
4    FakeDelay(overhead * ratio_delay_A);
5    if (do_nested_parallelism)
6      Test1(k, k_max);
7    FakeDelay(overhead * ratio_delay_B);
8  }
```

(b) `Test2`: `Test1` + inner-loop parallelism + nested parallelism.

Figure 83: Validation testers: `Test1` and `Test2`.

severe errors. Our investigation for such errors concludes that operating system-level thread scheduling can affect the speedups, which the FF currently does not precisely model. An example was discussed in Figure 70. We also discover that the overhead of OpenMP is not always constant, unlike the results from the previous work [13, 22]. The previous work claims that the overhead of OpenMP constructs can be measured as constant delays. However, our experimentation results defied the previous result: the overhead was also dependent on the trip count of a parallelized loop and the degree of workload imbalance. To address such difficulties, we have introduced the synthesizer. The synthesizer provides more accurate predictions for `Test2`, showing a 3% average error ratio and 19% at the maximum, shown in Figure 84(e). Note that such a 20% deviation in speedups

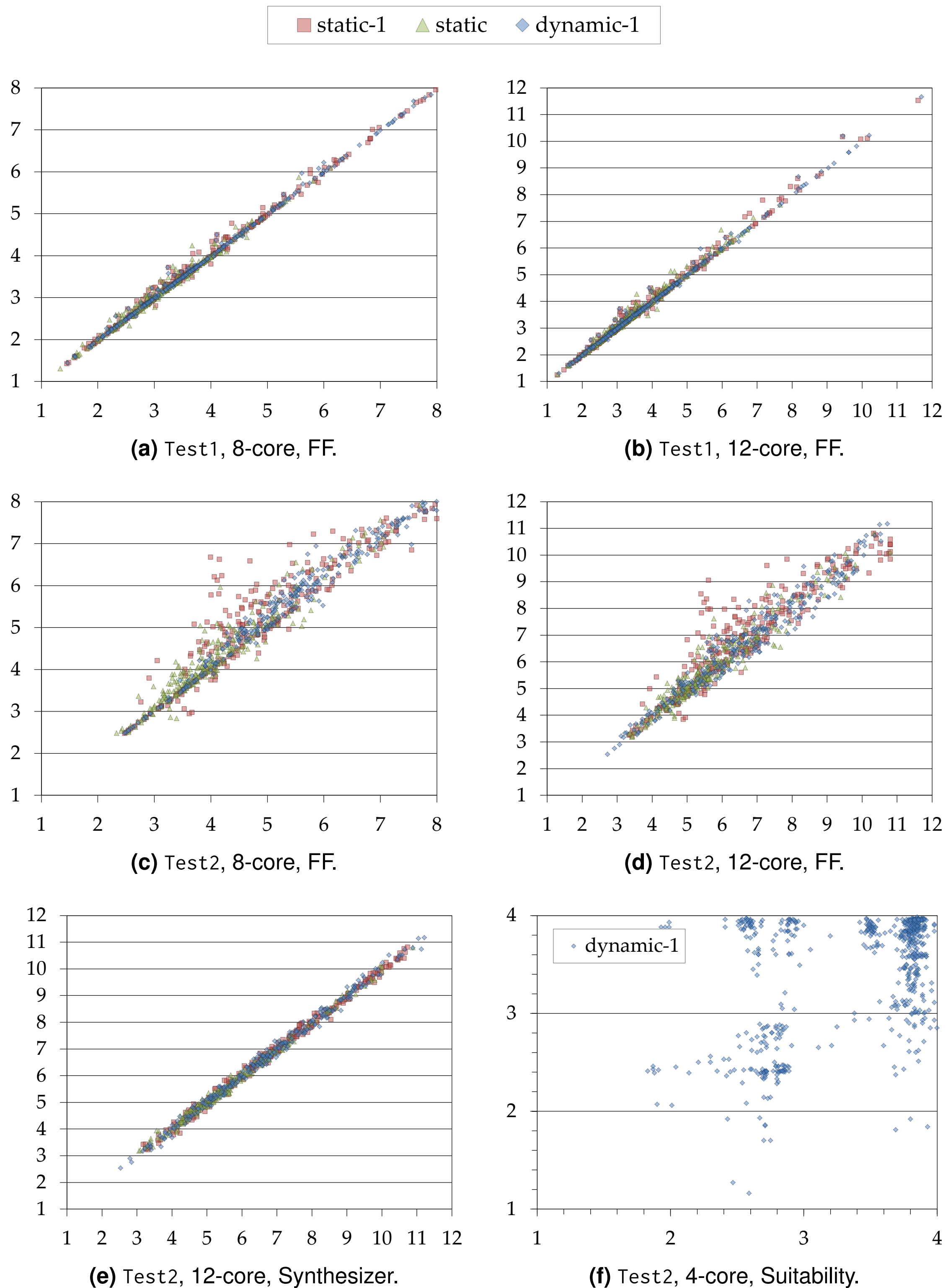


Figure 84: Prediction accuracy results of `Test1` and `Test2`, targeting 8 and 12 cores, and OpenMP, with the FF and synthesizer: X-axis is the measured *real* speedups, and Y-axis is the *predicted* real speedups. Sub-figure (e) shows that the synthesizer outperforms the FF when the core numbers are large. Suitability does not provide accurate results because the emulator of Suitability does not explicitly model OpenMP's scheduling policies.

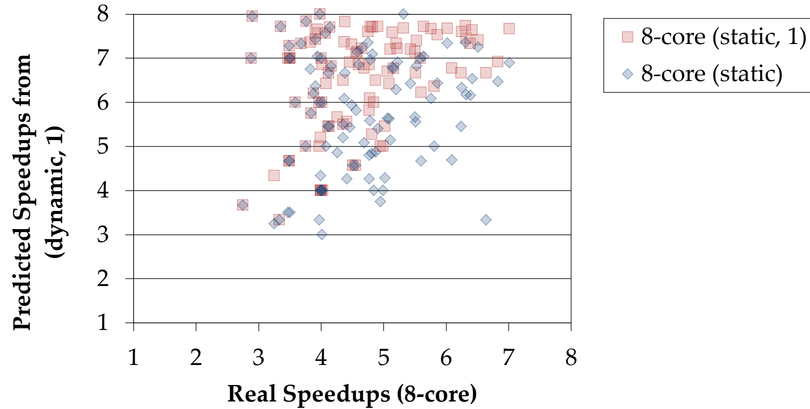


Figure 85: Inaccurate prediction of `Test2` without considering specific scheduling policies: Real speedups were measured using `(static)` and `(static, 1)`, but the predictions were based on `(dynamic, 1)`.

are often observed in multiple socket machines; thus we consider a 20% error as a boundary for reasonably precise results.

We also conduct validation tests with Intel Parallel Advisor's Suitability analysis. Due to constraints of the out-of-the-box tool, we are only able to predict `Test1` and `Test2`, up to a quad-core. The results of `Test1` (not shown in the thesis) are as accurate as ours. However, Figure 84(f) shows that it does not predict well for the cases of `Test2`. Suitability does not currently provide speedup predictions for a specific scheduling, but the emulator of Suitability is known to be close to the OpenMP's `(dynamic,1)`. We speculate the reasons for such inaccuracy would be (1) the limitations of the emulator discussed in Section 6.3.2, and (2) the lack of precise modeling of OpenMP's scheduling policies. This result says that explicit modeling scheduling policies is important. Figure 85 also supports this claim.

We intentionally mismatched predicted and measured speedups. We plot the real speedups of `Test2` with `(static)` and `(static, 1)` versus the predicted speedups with `(dynamic, 1)`. The figure clearly shows the prediction is not accurate. There are many points in the region of $y > x$, showing poor prediction because a dynamic scheduling generally achieves higher speedups. We argue that

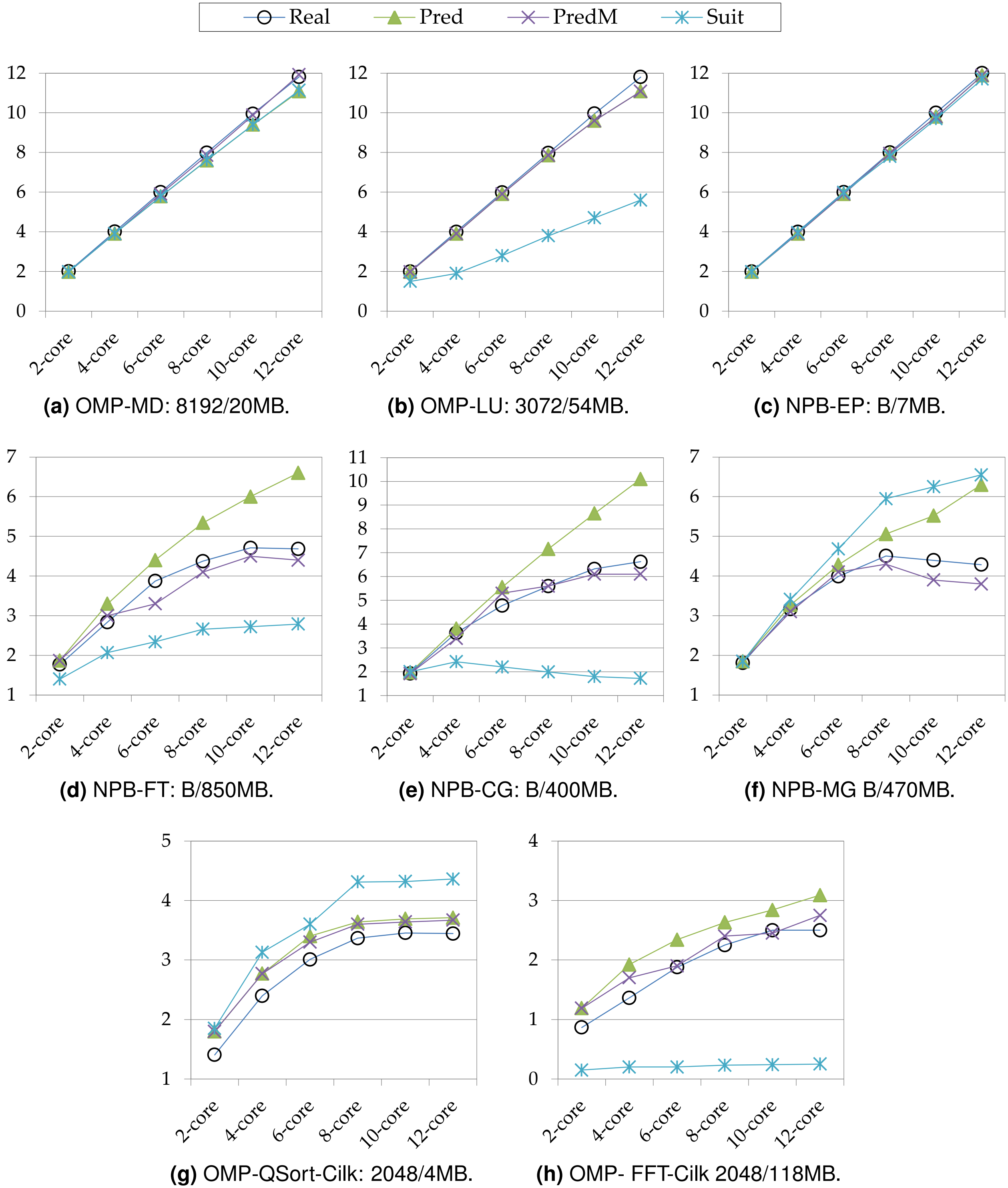
an emulation must explicitly model detailed scheduling policies.

6.6.3 Prediction Results with Memory Performance Model

We present the prediction results on four OmpSCR (MD, LU, FFT, QSort) and four NPB benchmarks (EP, FT, MG, CG) in Figure 86. Each benchmark is estimated by (1) the synthesizer *without* the memory performance model ('Pred'), (2) the synthesizer *with* the memory performance model ('PredM'), and (3) Suitability ('Suit'). We also measure the real speedups: OMP-QSort and OMP-FFT are parallelized by Cilk Plus to support efficient recursive parallelism; and the others are parallelized by OpenMP. The sub-captions of the figure identify corresponding the inputs and the maximum physical memory footprint sizes.

Four benchmarks, OMP-MD, OMP-LU, NPB-EP, and OMP-QSort-Cilk , show highly accurate prediction results, even without considering memory effects. They do not show large memory traffic, so our memory performance model (MPM) predicts burden factors of 1 in most cases. However, we slightly underestimate the speedups of OMP-MD and OMP-LU on 6, 8, 10, 12 cores. This may be explained by the increased effective L1 and L2 cache sizes. Our MPM does not currently consider such an optimistic but computes the burden factors of 1, meaning these benchmarks are not limited by memory bandwidth limitation. The result of Suitability for OMP-LU was poor. One of the reasons would be the fact that LU has a frequent parallelized inner loop.

We discuss the cases of NPB-FT, NPB-CG, and NPB-MG. We observe that the parallel performance of these three benchmarks are saturated and degraded on higher core numbers due to increased memory traffic. The memory footprints are fairly large, over 400 MB. Our additional experiment, which is presented in the next section, confirms that these benchmarks exhibit significant memory traffic in some phases. Hence, without the MPM, Parallel Prophet overestimates for these



(a) OMP-MD: 8192/20MB. **(b)** OMP-LU: 3072/54MB. **(c)** NPB-EP: B/7MB.

(d) NPB-FT: B/850MB. **(e)** NPB-CG: B/400MB. **(f)** NPB-MG B/470MB.

(g) OMP-QSort-Cilk: 2048/4MB. **(h)** OMP- FFT-Cilk 2048/118MB.

Figure 86: Predictions of OmpSCR and NPB benchmarks: 'Pred' and 'PredM' are the results without/with the memory performance model, respectively. 'Suit' is the results of Suitability. Suitability does not have a memory performance model and only provides speedups for 2^N CPU numbers. The predictions of Suitability for 6/10/12 cores are interpolated. The captions show the input set and its maximum memory footprint.

three benchmarks. The estimations in "Pred" show that the programs run scalably in higher core numbers.

However, the results of "PredM", the predictions with our MPM, are clearly more accurate than "Pred" and "Suit". The MPM computes burden factors for parallel sections with the memory profiling. For these three benchmarks, there are a number of sections whose burden factors are greater than 1. For example, the burden factors of NPB-FT show the range of 1.0 to 1.45 for two to 12 cores. Some burden factors (e.g., NPB-FT and NPB-MG) tend to be predicted conservatively. However, we claim that the results show the outstanding prediction ability.

Finally, two recursive parallelism patterns are demonstrated in OMP-QSort-Cilk and OMP-FFT-Cilk, which cannot be efficiently implemented by OpenMP 2.0. They are parallelized by Cilk Plus and predicted. As our synthesizer can easily model a different threading model, the results are also quite close to the actual speedups. For the case of OMP-FFT-Cilk, Suitability was unable to provide meaningful predictions.

6.6.4 A Simple Verification for the Memory Performance Model

We do a simple verification for the MPM with the microbenchmark used to derive Δ and Ω. The result is presented in Figure 87. We first measure the actual speedups of the benchmark when the parallelized version is running under various memory traffic and different thread numbers (3, 4, and 5 threads). Because the benchmark creates heavy memory traffic, the speedups are often saturated. We cannot observe a speedup greater than 3. Note that this experimentation was done on a single-channel DRAM to show the saturation easily, while the results in Figure 86 were obtained on dual-channel DRAMs. We then predict speedups by our MPM. As Figure 87 shows, most of the predictions are within 33% error bounds.

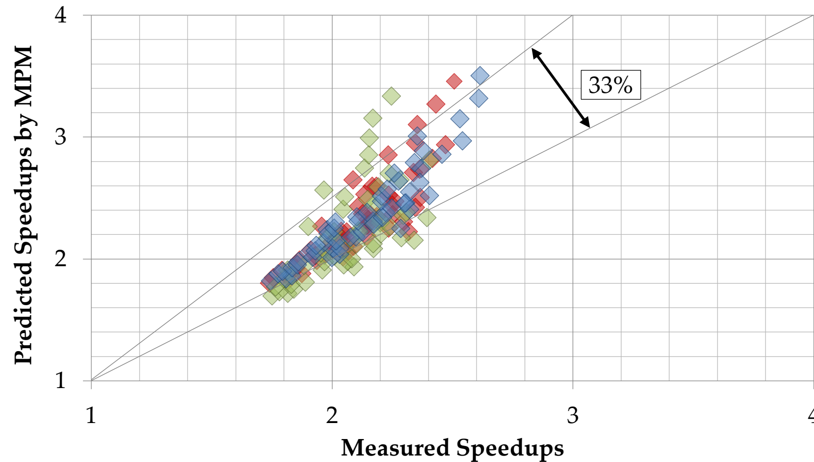


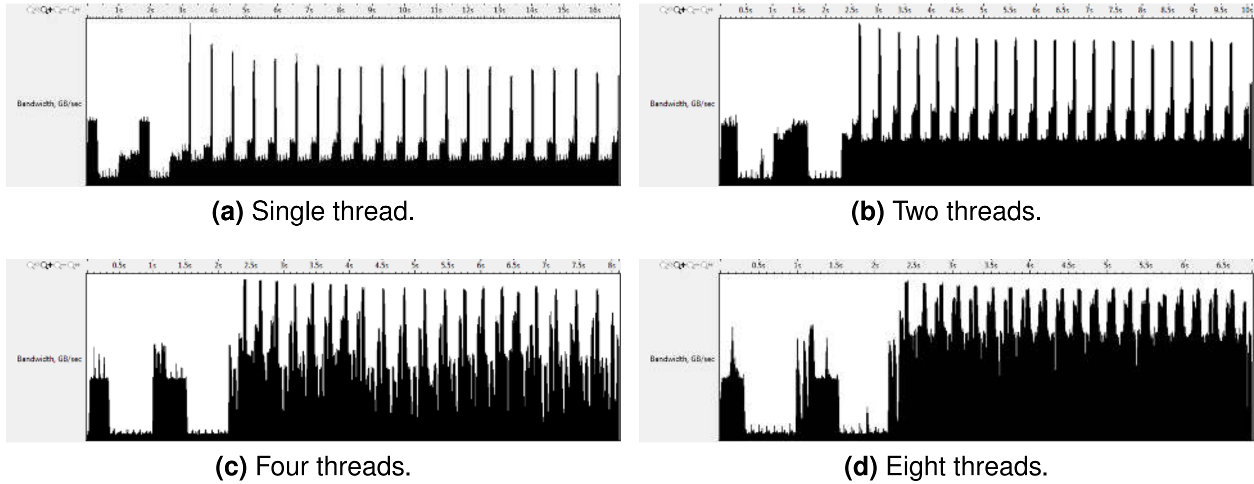
Figure 87: A simple verification test for memory performance model: We used our microbenchmark (Section 6.4.5) to test the MPM. Most of the predictions are within 33% error bounds. The experimentation was done for only 3, 4, and 5 threads. Different colors indicate different numbers of the independent chains (Figure 75).

6.6.5 Detailed Cache and Memory Behaviors of the Benchmarks

Section 6.4.1 detailed the assumptions of the MPM. One of the important assumptions is that the miss ratio of LLC does not significantly vary from a serial version to a parallel version. This section presents a number of supporting profiling results for the NPB-FT benchmark.

Figure 88 shows the overall bandwidth behavior of NPB-FT running on 1, 2, 4, and 8 threads, measured by Intel VTune [57]. The peak traffics do not increase even if the number of threads increased because the memory bandwidth is already saturated by single-thread traffic. From this observation, we can easily guess that the speedup would be saturated.

The bandwidth varies by functions. The periodic peaks shown in Figure 88 are from `evlove` function of NPB-FT while the following flat is from `cftts1` and the small hill before the peak is from `cftts2` and `cftts3`. Figure 89 presents the memory traffic behaviors of these three main routines of NPB-FT. This profiling data demonstrates well that the main bottleneck of the saturated speedups is because of `evlove` function.



(a) Single thread. **(b)** Two threads.

(c) Four threads. **(d)** Eight threads.

Figure 88: Memory traffic of NPB-FT measured by Intel VTune: X-axis is the elapsed seconds, and Y-axis is the bandwidth. The peak bandwidth are generated from `evolve` function of NPB-FT, followed by a low traffic from `cffts1` and a bit more traffic from `cftts2` and `cftts3`.

We verify the variation of LLC misses (MPKI) of NPB-FT's `evlove` function. We measure MPKIs from single and parallel versions of NPB-FT. MPKIs of the single thread are the block dots of Figure 90(a). For the parallel versions, we measured per-thread MPKIs on 2, 4, and 6 cores and plot each thread data. Except for a few outliers, the figure shows that the trends of the MPKI numbers well coincide. The geometric average of the maximum differences is MPKI of 2.12. Recall that `evolve` is the most memory consuming function. Figure 90(b) shows the results of `cffts2`,

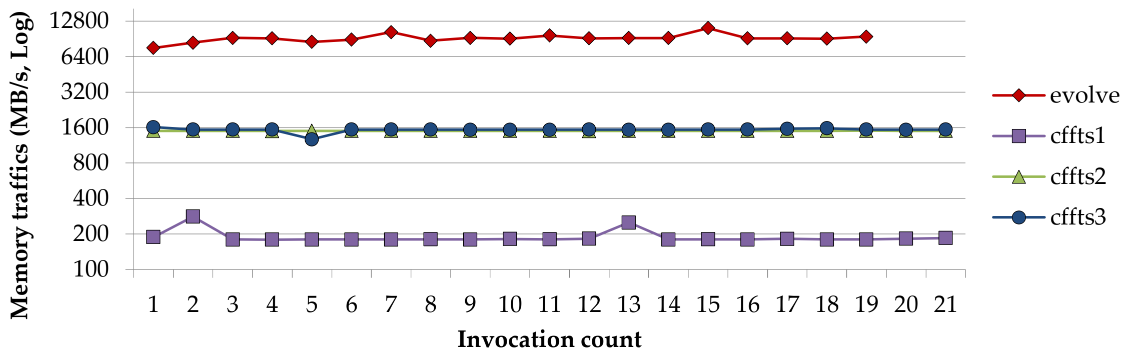
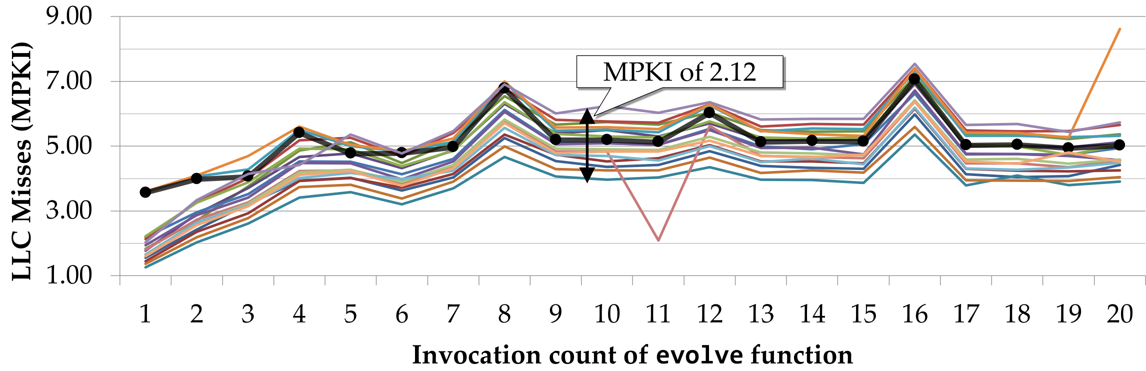
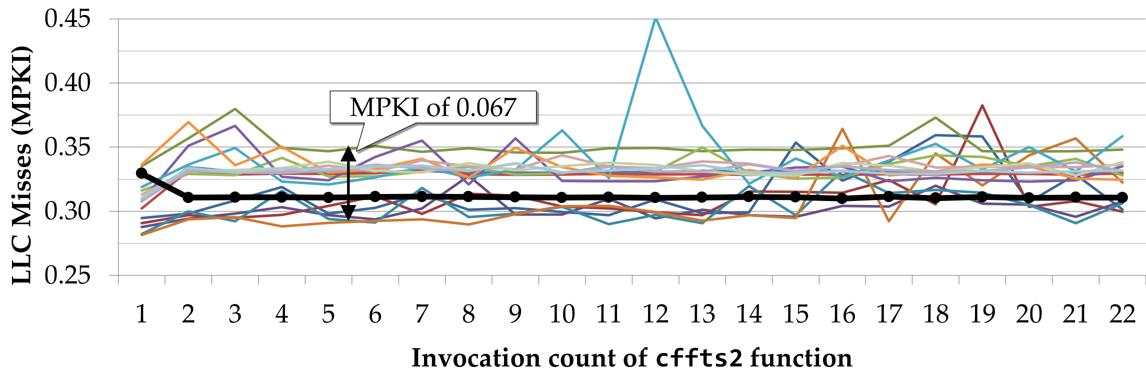


Figure 89: Memory traffics (shown in log scale) from the major routines of NPB-FT by the invocation count: `evlove` function generates peak bandwidth.



(a) NPB-FT's `evolve`.



(b) NPB-FT's `cffts2`.

Figure 90: LLC misses as MPKI of NPB-FT's `evlove` and `cftts2` in serial and parallel versions. The black dots and line are the numbers from the serial program. We obtained these MPKIs from the parallelized version running 2, 4, and 6 cores. Each thin line indicates MPKIs from a thread running in the parallelized program.

which demands the memory moderately as shown in Figure 89. In this function, the MPKIs of the serial version remain in constant: 0.31 of MPKI. The parallel versions' MPKIs, however, exhibit variations. Unlike the case of `evolve`, a common trend is not observed in the numbers; rather we can see a random fluctuation. Nevertheless, the MPKIs are mostly within the range of 0.25 to 0.35. The average of the maximum differences is only 0.067 by average: 22% of the serial MPKI. This data supports our assumption of the steady LLC miss ratio in the evaluated benchmarks.

6.6.6 The overhead of Parallel Prophet

The worst memory overhead in all experimentations in the thesis is 3 GB when only the loseless compression is used. However, some benchmark, such as FT in NPB only requires 5 MB for storing its program tree. The time overhead is mostly 1.1× to 3.5× slowdown *per each estimate*, except for the predictions of FFT and IS with the FF emulator. It shows more than 30× slowdown for these cases. Note that Suitability shows 200× slowdowns for FFT. Such high overhead comes from significant overhead of tree traversing and intensive computation using the priority heap in the FF emulator. Note that this time overhead represents a slowdown against the execution time of an unannotated serial program. The synthesizer only shows approximately 3.5× slowdowns for FFT and IS.

Note that quantifying the time overhead is somewhat complicated. The total time overhead is actually dependent on how many estimates programmers want. The more estimates desired, the more time is needed. In addition, the synthesizer actually runs a parallelized program, hence its time overhead per estimate is dependent on its estimated speedup. If an estimated speedup is 2.5, then the overhead would be at least 1.4× slowdown (=1+1/2.5). The total time of the synthesizer can be expressed:

$$T_{syn_total} = T_{profiling} + \sum_{i=1}^{K}(T_{traverse} + T_{serial}/S_i), \quad (12)$$

where T_{serial} is the execution time of the serial version, $T_{profiling}$ is the overhead of the interval profiling, $T_{traverse}$ is the overhead of tree-traversing in the emulator, K is the number of estimates to obtain, S is the measured speedup. Note that $T_{profiling}$ and $T_{traverse}$ are heavily dependent on the frequency of annotations.

6.6.7 Limitations of Parallel Prophet

Although we expanded the prediction coverage by employing the synthesizer and a memory performance model, Parallel Prophet has the following limitations:

- The speedup prediction is based on runtime profiling. The profiling result is dependent on an input like SD^3. Programmers should give a representative input to obtain faithful speedup predictions.
- We assume no I/O (file, disk, or network) operations in annotated regions.
- Current annotations only support task-level parallelism and mutex. However, we believe that supporting other types of synchronization and parallelism patterns is not challenging. For example, pipelining can be easily supported by extending annotations [139] and emulation algorithm.
- We already discussed the assumptions of the MPM. The assumptions become the current limitations of the MPM. For example, we do not model super-linear effects, data sharing, and changed cache behavior in LLC.
- We found two potential dimensions in the formula of Ω. During the experimentation using the microbenchmark, there were cases where the number of thread and the degree of the independent load (Figure 75) in the microbenchmark affected the shape of the correlations between ω and the bandwidth. In this thesis, we excluded low numbers of thread and independent load to obtain the formulas in Equations (10) and (11).
- The experimentation results for the MPM in Figure 86 required a fine tuning for NPB-MG and NPB-CG. They have many very short-period sites, so the obtained profiling data were highly fluctuated. We computed an averaged burden factor for a set of short-period sites.

6.7 *Summary of This Chapter*

This chapter presented Parallel Prophet, which predicts potential speedup of a parallel program by dynamically profiling and emulating from a serial program. It takes input as annotated serial programs and annotations describe parallel loops and critical sections. We provided detailed descriptions of two emulation algorithms: the fast-forwarding emulation and the synthesizer. In particular, the synthesizer is a novel method to model various parallel patterns easily and precisely. We also proposed a simple memory model to predict the slowdown in parallel applications resulting from memory resource contention.

We evaluated Parallel Prophet with a series of micro-benchmarks that show highly accurate prediction results (less than 5% errors in most cases). We also demonstrate how Parallel Prophet executes the prediction with low overhead. Finally, we evaluated Parallel Prophet with a subset of the OmpSCR benchmarks and the NPB benchmarks. We presented good accuracy against these benchmarks as well. Our memory model can predict performance degradations after parallelization and proved its effectiveness.

CHAPTER VII

ALGORITHMS TO EXTRACT PARALLELISM AND TRANSFORMATION ADVICE

This chapter presents the *post-analyzer*, a set of algorithms that further analyzes the raw dependence results, and thereby programmers easily understand the parallelism of their serial code and obtain hints for writing parallel code. The post-analyzer can be used to assist the manual annotation of Parallel Prophet.

7.1 Assumptions for the Post-Analyzer

First, we judge the parallelizability of a given serial program by the patterns and existence of the data dependence. The post-analyzer recommends a code transformation for a set of dependences if the dependence pattern can be handled in our algorithms: for example, for this kind of dependences, a reduction can remove the dependences. Sometimes dependences could be a conservative criterion to find parallelism. For example, implementation artifacts such as a memory allocator can create dependences while such dependences can be ignored easily. A study shows that some dependences may be ignored under certain limited conditions [143]. However, for our purpose in which code modification for parallelization is needed, we claim that data dependences are still the most critical factor. Parallelizability can also be defined by other criteria such as the commutativity of operations [119, 4, 112, 68] and the critical path [33].

Second, we assume that a transformation to avoid dependences is orthogonal to other transformations. In other words, we report a transformation advice per a particular set of dependences, but do not consider their relative ordering in the

whole transformations. We further discuss this issue in Section 7.2.5

Finally, we assume that there is no significant pre-modification in serial code. We aim to write an initial version of parallel code from the original serial code. It is true that programmers often perform pre-modification on serial code - other than parallelization - so that the parallelization is easily to be achieved. An example is fluidanimate in the PARSEC suite [8]. The programmer substantially modified the structure of the original serial code to write parallel code easily. One of the future work is to investigate common such pre-modification for more amenable parallelization.

7.2 *Overview of the Post-Analyzer*

Given a loop and dependence profiling raw result, we devise algorithms and heuristics that automatically and systematically extract parallelism and advice on writing parallel code. Ideally, we would like to give such hints: "*You can parallelize the loop of the function* `foo` *if you insert a mutex on this operation*".

We think that finding a generalized solution to avoid an arbitrary pair of data dependences is infeasible. Instead the post-analyzer attacks important data-dependence patterns that can be solved by well-known parallelization techniques, such as privatization, reductions and mutexes, barriers and condition variables, and pipelining. Table 14 summarizes the classification of data dependences and solutions to avoid them.

In the following sections, we detail each pattern and associated parallelization techniques. We finally present several case studies of the post analyzer, covering privatization, reduction, a simple mutex case, and pipelining. Condition variables, barriers, and mutexes are the future work.

Table 14: Classification of data dependences and its potential solutions for parallelization. We provide case studies for the patterns with ✔.

Kind	Loop-carried?	Implications for code modifications	
Anti	Don't care	Using a separate variable/array; Privatization	✔
Ouput	Independent	Using a separate variable/array	✔
Ouput	Carried	Privatization	✔
Flow	Carried	Reductions for commutative/associative operators	✔
		Mutexes when computation order does not matter	✔
		Barrier when computation order matters	
		Condition variables when computation order matters	
Flow	Independent	Pipelining (`DOACROSS`) for certain types	✔

7.2.1 False Dependences and Privatization

We first discuss false dependences (anti- and output dependences), which do not require significant changes as the dependences themselves are artifacts due to reuse of variables and arrays. Anti-dependences can be avoided by separate data structures. Loop-independent output dependences are also the same case. Although these kinds of dependences are easy to fix, optimizing redundant structures could be an issue for performance.

Loop-carried output dependences can be avoided for parallelization by the technique called *privatization*, which allocates a private copy of an object for each thread. The post-analyzer recommends that a set of dependences on an object (either a scalar variable, a structure, or an array) can be avoided by privatization if and only if:

- An object has only loop-carried output dependences followed by loop-independent flow dependences.

This criteria implies that the very first memory access on this object for an iteration of a loop is a write access. This definition can be also found in [117]. A simple example of privatization is illustrated in Figure 91. The dimension of the original global variables are increased by one. However, this simple code may

```
1  // Global variables
2  int  g_data;
3  int* g_vector = new int[100];
```

(a) Before privatization.

```
1  // Global variables
2  int  g_data[nthreads];
3  int* g_vector[nthreads];
4
5  for (int i = 0; i < nthreads; ++i)
6    g_vector[i] = new int[100];
```

(b) After privatization.

Figure 91: A naive example of privatization for output dependences: This code suffers from false sharing.

suffer from false sharing. Avoiding false sharing in this case is also simple by allocating per-thread data on cache line boundary.

Privatization may seem to be an easy case, but this is practically important. Legacy serial C code tends to use a lot of static and global variables. Class member variables in C++ can be considered as global variables with limited scope. If a programmer who is parallelizing is not the original author of the serial code, modifying all such variables correctly may be tedious and requires efforts to verify the correctness, particularly for large-scale software. Therefore, providing information on privatization is important. Prospector's results are informative in that such variables will be reported to have output dependences.

Many parallel programming languages and libraries have a specialized support for privatization. For example, OpenMP supports handy `private` and `threadprivate` pragmas, and Cilk Plus provides a `holder`. Some compilers do not currently support non-POD (plain old types) structures and classes in OpenMP's `threadprivate`. Even in such case, programmers can write a simple thread-local storage while avoiding false sharing.

7.2.2 Reductions and Mutexes

We discussed the issues of false dependences, but the real challenges are handling flow dependences, also known as true dependences. An extreme solution

for an arbitrary pair of true dependence would be enforcing the order of the computations by using a condition variable, event, and phaser [131]. However, such strict solution should be the last resort only for a particular case. Many other techniques could solve flow dependences while providing lucrative speedups. Reductions and mutexes (or critical sections) are such cases.

A reduction is suitable for the computations whose operations are commutative and associative, such as totaling all elements and finding the minimum element. In general, the definition of a reduction would include both scalar variables and class instances with general computations beyond primitive operators. However, the current post-analyzer, we only consider scalar variables and simple operations. The post-analyzer report a scalar variable as a potential reduction variable if the dependence patterns meet the following conditions (a similar definition of reduction can be found in [117]):

1. All three kinds of loop-carried dependences are observed on the variable;
2. Modifications on the variable must be the same for all control flows except for the case where an operation needs a branch, such as finding maximum or minimum;
3. The operation on the variable is one of reducible operations (i.e., an associative and commutative operation).

Sections 7.3.1 and 7.3.2 illustrate reductions. Resolving a reduction can be done by either manual transformation or a support from parallel programming paradigms. OpenMP's `reduction` clause and Cilk Plus's `reducer` are the examples. OpenMP currently supports only a scalar reduction. Extracting a reduction on a complex data structure whose commutative operations are implemented as a function will be a great challenge, which is a future work.

A mutex is a generalized reduction solution when a data structure (including non-POD types) has loop-carried dependences of all three kinds, *and* the operations on the structure can be executed in any order while atomicity must be preserved. An example would be a storage for items where the order of insertion and removal is not important. In this case, the insertion and removal operations will make data dependences on the internal data structures. However, a further analysis such as commutativity analysis [119, 4] could reveal the commutativity property. We found this case in Dijkstra benchmark of MiBench [37] and present in Section 7.3.3. A mutex or critical section solves this case. For a certain case where a scalar variable is modified, atomic operations should be preferred for the performance, such as `std::atomic<>` in C++11 [61] and `InterlockedIncrement()` in Win32 APIs.

7.2.3 Supporting condition variables and barriers

As discussed right before, loop-carried flow dependences involved in a commutative operation can be resolved by a reduction or a mutex. When the computation order matters, programmers must enforce the correct ordering of the computations to guarantee the correctness. Solutions for these patterns would be a traditional condition variable and barrier, and an advanced synchronization objects such as phaser in Hanabero [131] and `std::future<>` in C++11 [61]. The distance information of data dependences may be important as demonstrated in a related work [156]. SD^3 also provides the distance information. However, we remain this work as a future work with open questions: (1) How to differentiate the condition variable and barrier cases from the reduction and mutex cases; (2) How to further classify the condition variables and barriers; and (3) Predicting potential speedups by extending Parallel Prophet.

7.2.4 Pipelining

The final pattern that the post-analyzer presents is pipeline parallelism. Many streaming applications including transcoding audio and video, image processing, and vision algorithms can be parallelized by pipelining [139, 140]. A typical case of pipeline is shown in Figure 92.

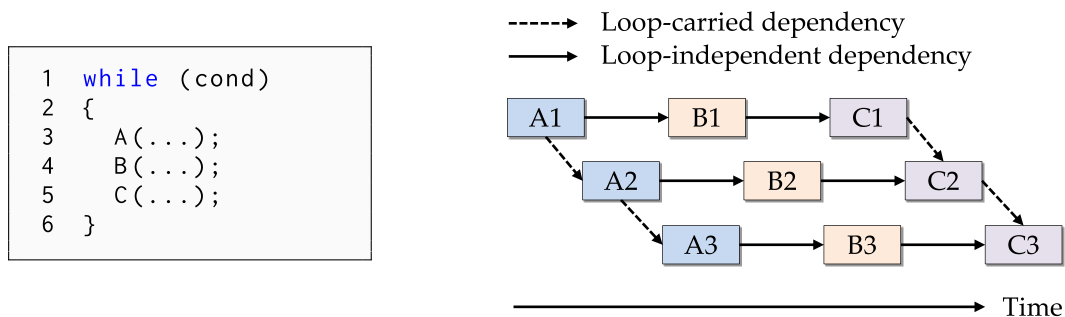


Figure 92: A loop that can be parallelized by pipelining: The dependences are shown in the right figure. Note that the function B does not have loop-carried dependences on itself, allowing a parallel execution.

For the simplicity, assume that each function forms a pipeline stage. In order to do pipelining this loop, the dependence pattern must satisfy some conditions. Figure 92 shows a case how loop-independent flow dependences define a pipeline configuration. Further, function B does not have loop-carried dependences on itself. This fact allows to exploit parallel execution on B. 256.bzip2 in SPEC 2000 is an illustration of the case of Figure 92, and we present in Section 7.3.4.

7.2.5 Ordering of Transformations

We assumed that transformations are orthogonal to other transformations. For example, consider Table 16 in the following section. In order to parallelize 179.art, we need to perform reduction, privatization, and some other minor modifications to remove data dependences. Notice that we do not provide the ordering of transformations; the transformations may be done in an arbitrary order. For all

case studies in Section 7.3, transformation ordering does not affect, at least, the correctness of the parallelization.

Such ordering, however, can be critical to guarantee the correct parallelization when the transformations involve synchronization. Suppose a program has several sets of dependences where each set is avoided by mutex, condition variable, and barrier, respectively. Here are open questions: What is the right ordering of these transformation? Do these transformations have *linearity* or *composability*? The linearity and composability in this context mean as follows: Suppose F is a function that takes a set of dependences and returns a solution to avoid them. In general, we speculate that $F(a + b) \neq F(a) + F(b)$ and $F(a) + F(b) \neq F(b) + F(a)$, when dependence sets a and b require synchronization. Some combination of transformations may lead a dead lock because locking is not composable unlike transactional memory. However, we believe that there are linearity and composability in transformations to remove false dependences although performance could be an issue. This thesis assumes the linearity and composability in transformations.

Although no formal proof is provided, we suggest a heuristic for the ordering of transformations as follows:

(1) Any transformation to avoid false dependences such as privatization;

(2) Any transformation to avoid true dependences related to reduction;

(3) Any transformation to avoid true dependences related to mutex;

(4) Finally, apply an appropriate parallelism (e.g., parallel-for or pipelining).

Recall that this thesis does not consider the case of other synchronization primitives such as condition variables and barriers. The ordering including such synchronization is not known.

7.3 *Case Studies of Prospector*

We present several case studies with Prospector's post-analyzer to demonstrate the discussed dependence patterns and parallelization techniques. Table 15 shows six benchmarks we parallelize. The benchmarks are selected from MiBench [37], an embedded benchmarks suite, and SPEC CPU2000 [133] are used. We use the LLVM-based Prospector in this experimentation because we need more detailed information of source code and some static analysis code.

Table 15: Parallelized benchmarks with Prospector's post-analyzer.

Benchmark	Parallelism model	Synchronizations
179.art (SPEC 2000)	Data parallel	Reduction, Privatization
Susan (MiBench)	Data parallel	No, except for output writing
Dijkstra (MiBench)	Data parallel	Mutex
256.bzip2 (SPEC 2000)	Pipelining + Data parallel	Pipelining
BasicMath (MiBench)	Data parallel	No
StringSearch (MiBench)	Data parallel	No

7.3.1 179.art in SPEC CPU2000

179.art in SPEC CPU2000 may be considered an easy case with a TLS technique, but production compilers cannot automatically parallelize the main loop. A programmer who does not know the algorithm of 179.art (1,300 lines) could take a couple of days to parallelize.[1] Prospector can help this process significantly.

We profile 179.art by SD^3 with the train input. The post-analyzer parses the raw input and makes summarization. Table 16 summarizes the processed result. We present how we interpret this result, which is also illustrated in Figure 93.

- Prospector finds that the loop `scan_recognize:5` in Figure 93 is the hottest loop with 79% execution coverage, only one invocation, and 20 iterations. Every

[1] A graduate student who did not know the details spent more than two days to parallelize 179.art only with a gprof-like profiler.

Table 16: Post-analyzed result of the loop, `scan_recognize:5`, in 179.art.

Loop Profiling	79% execution coverage; 1 invocation and 20 iterations; Standard deviation of iteration lengths: 3.5%
Dependence Profiling	Loop-carried WAWs on `f1_layer` ... Temporary variables on `i`, `k`, `m`, `n` Induction variable on `j` at line 5 Reduction variable on `highest_confidence`
	No Loop-carried RAWs: *may be parallelizable*

iteration has almost an equal number of executed instructions (implying good balance). The loop could be a good parallelization candidate.

- Regarding dependence profiling, the loop has *no* loop-carried *flow* dependences except a reduction variable and an induction variable (i.e., loop counter variables). We apply the same classification algorithm of reduction to find an induction variable.

- `scan_recognize:5` has many loop-carried output dependences on global variables such as `f1_layer` and `Y`. See also `reset_nodes()` in Figure 93. These variables are not automatically privatized to threads, so we perform privatization, explicitly allocating thread-local private copies of these global variables (not shown in the figure).

- Temporary variables in the scope of the loop at line 5 are found such as `i` at line 6. It is true that this is because old-fashioned variable declaration in C. We insert `i` and other variables (`k`, `m`, and `n`) to OpenMP's private list. (Or, in C99 and C++, we can simply declare `i` within the scope of `for`. We classify a set of dependences as a temporary variable if the very first time access in a loop iteration is a write (initialization) and then is followed by loop-independent flow dependences (uses).

```
 1: void scan_recognize(startx, starty, endx, endy, stride)
 2: {
 3:   ...
 4:   #pragma omp for private (i,k,m,n)
 5:   for (j = starty; j < endy; j += stride)
 6:     for (i = startx; i < endx; i += stride){
 7:       ...
 8:       pass_flag = 0;
 9:       match();
10:       if (pass_flag == 1) {
11:         if (set_high[tid][0] == TRUE) {
12:           highx[tid][0] = i, highy[tid][0] = j;
13:           set_high[tid][0] = FALSE;
14:         }
15:         if (set_high[tid][1] == TRUE) {
16:           ...
18:     }  // End of for-i
...
30: void match()
31: {
32:   reset_nodes();
34:   while (!matched) {
35:     ...
48:     int match_cnfd = simtest2();
41:     if ((match_cnfd) > rho) {
43:       pass_flag = 1;
44:       if (match_cnfd > highest_confidence[tid][winner]){
45:         highest_confidence[tid][winner] = match_cnfd;
46:         set_high[tid][winner] = TRUE;
47:       }
48:     ...
```

```
70: void reset_nodes()
71: {
72:   for (i = 0;i < numf1s; i++) {
73:     f1_layer[tid][i].W = 0.0;
74:     Y[tid][i].y = 0.0;
...
```

Need parallel reduction

Figure 93: Simplified parallelization steps of 179.art by Prospector's result: (1) Privatize global variables; (2) Insert OpenMP pragmas; (3) Add a reduction code (not shown here).

- A potential reduction variable of the loop at line 5, `highest_confidence`, is identified. The dependences on this variable have all three kinds of loop-carried dependences, which is easy to verify. However, identifying the operation on this variable is harder because it requires comparison rather than an arithmetic operation. We check whether a write immediately follows a read that is involved in a branch operation. Then, we can conclude that the variable is intended to calculate the maximum and this operation is a commutative operation. OpenMP does not support in this operation. We manually modify the code to obtain local results and compute the final answer.

Programmers do not need to know the very details of 179.art. By interpreting Prospector's results, programmers can easily understand and finally parallelize. It is true that Prospector cannot prove the parallelizability. However, when compilers cannot automatically parallelize the loop, programmers must prove its correctness by hand or verify it empirically using exhaustive tests. In such cases, Prospector's assistant can save programmers' efforts.

7.3.2 Susan in MiBench

In this case study, we expose raw results to show how the post-analyzer works in detail. Susan is a simple image processing program in MiBench. Susan performs smoothening, edge, corner detection algorithms. The code is approximately 2,200 lines, and the loop profiling result from Prospector shows that nested loops at `susan_smoothing` consume almost 100% of the entire execution time.

```
; Hot loops
;  1: susan_smoothing:724@1, 100%
;  2: susan_smoothing:729@2, 100%
;  3: susan_smoothing:737@3, 99.36%
;  4: susan_smoothing:739@4, 97.63%
;  5: enlarge:645@1, 0.002493%
```

Figure 94: Hot loop information of Susan in MiBench: @n denotes that this loop is n-level in its loop nest.

It is reasonable to investigate the parallelizability of the top-level loop at line 724, `susan_smoothing:724@1`. The detailed loop profiling (not shown in here) revealed that there is a single invocation of this loop, and the total trip count is 1,490, which is sufficiently large iterations to be parallelized.

Figure 95 shows an excerpt from the raw data-dependence result. Regarding the dependences of `susan_smoothing:724@1`, the variable `out` shows all loop-carried dependency kinds (RAW, WAR, WAW), but the other variables show only loop-carried output dependences (WAW). The post-analyzer classifies these

```
+(R, out, 752, 6, susan_smoothing) after (W, out, 750, 6, susan_smoothing) x737
+(W, out, 752, 6, susan_smoothing) after (R, out, 750, 6, susan_smoothing) x737
+(W, out, 752, 6, susan_smoothing) after (W, out, 750, 6, susan_smoothing) x737
...

+(W, tmp,   616, 5, median         ) after (W, tmp,   616, 5, median         ) x17
+(W, area,  731, 5, susan_smoothing) after (W, area,  731, 5, susan_smoothing) x1489
+(W, total, 732, 5, susan_smoothing) after (W, total, 732, 5, susan_smoothing) x1489
+(W, dpt,   733, 5, susan_smoothing) after (W, dpt,   733, 5, susan_smoothing) x1489
...
```

Figure 95: Data-Dependence profiling of Susan in MiBench. '+' indicates loop-carried dependences while '-' is for independent ones. Legends: (W/R, *variable name*, line#, column#, *function name*) x frequency.

dependences as temporary variables and suggests privatization. The variables such as `tmp` and `area` must be privatized for the correct parallelization. We declare these variables into the OpenMP's `private` clause. The code in Figure 96 is inserted to parallelize the top-level loop, `susan_smoothing:724@1`.

```
#pragma omp parallel default(none) \
    private(i, j, x, y)\
    private(area, total, centre, cp, brightness, tmp, dpt, ip)
```

Figure 96: Privatization for the variables that have output dependences (some anti-dependences may exist).

The only remaining problem for parallelizing this loop is the dependences on `out`. All three kinds of loop-carried dependence types are observed on `out`, and there is only a single modification for all control flows. Figure 97 shows the code how `out` is used. The post-analyzer concludes that `out` is potentially a reduction.

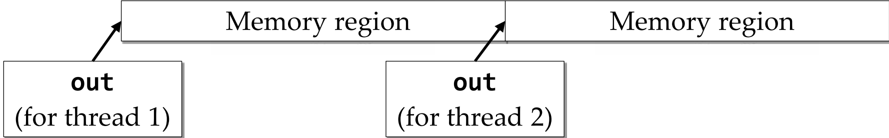
We attempted our first parallelization of `susan_smoothing:724@1` based on the

```
if (tmp==0)
  *out++=median(in,i,j,x_size);
else
  *out++=((total-(centre*10000))/tmp);
```

Figure 97: The variable `out` in Susan: Notice that `out` is dereferenced for the additional writing operation. We mistakenly identified as a reduction.

```
1  #pragma omp parallel default(none) private(...) shared(...)
2          firstprivate(out)
3  {
4    // Adjust out so that the final outcome are written correctly.
5    out += (trip_count_i*trip_count_j) / omp_get_num_threads() *
6      omp_get_thread_num();
7
8  #pragma omp for
9    for (i = 0; i < trip_count_i; i++) {
10     for (j = 0; j < trip_count_j; j++) {
11       ...
12       if (tmp==0)
13         *out++=median(in,i,j,x_size);
14       else
15         *out++=((total-(centre*10000))/tmp);
16     }
17   }
18 }
```

(a) A parallelized code.



(b) A visualization how to avoid dependences on out.

Figure 98: A parallelized version of Susan assisted by Prospector: Each thread has a separate version of out that points to a disjointed memory region.

information: (1) private lists and (2) a reduction on out. However, we soon found that the results from the parallelization were not correct, and data races were observed.

The reason of the failure is that the variable out is not a simple reduction variable. Observe that the type of out is a pointer type, which means out is used as a pointer to some memory region. Furthermore, this memory region is sequentially modified. Such sequential accesses to a specific memory region do not make any data dependence. This code is actually writing the final result sequentially. The post-analyzer did not consider such case. Notice that Intel Parallel Advisor simply ignored this case.

We suggest a following solution: if a reduction pattern is observed on a pointer variable and its access pattern is sequential, we suggest the programmer to divide

the memory accesses in multiple chunks based on (1) the number of threads, (2) the trip counts of the loops, and (3) the scheduling policy of the parallel loop. Although making a generalized solution would require more efforts (e.g., in case of a dynamic scheduling), Figure 98 illustrates a simple solution for the default OpenMP static scheduling. We make a separate and private pointer variable `out` and adjust them so that the original sequential semantic must be observed. This code returns a correct result.

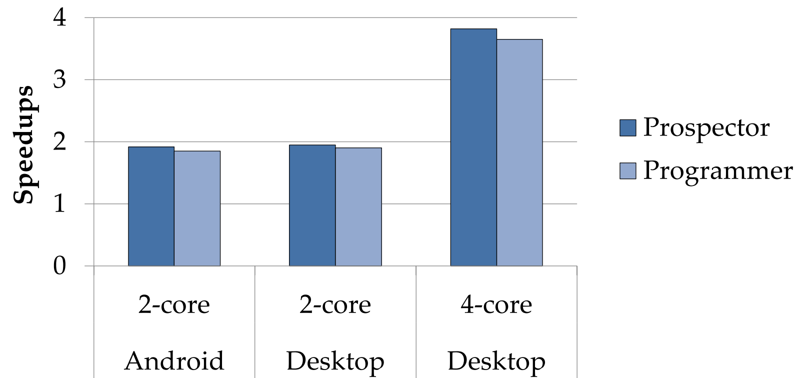


Figure 99: Speedups of parallelized Susan in MiBench on an Android and a desktop: (1) Prospector: parallelized solely based on Prospector's advice; (2) Programmer: manually parallelized version in ParMiBench [59].

We finally report measured speedups of Susan in Figure 99. "Prospector" indicates the speedups achieved solely based on Prospector's results while the speedups in "Programmer" are from a manually parallelized version found in ParMiBench [59]. We observe that our speedups slightly outperform those of the ParMiBench version. The ParMiBench version is written in the past using pthread. The code is much complex than our OpenMP-based implementation assisted by Prospector. Due to the problem of the variable `out`, we use the default static scheduling in OpenMP. We also measure speedups on an Android machine: a dual-core Nvidia Tegra 2 development platform [102]. As of the experimentation, we were unable to use OpenMP on this Android platform. We manually wrote a parallel implementation using pthread.

7.3.3 Dijkstra in MiBench

Dijkstra in MiBench is an implementation of the famous Dijkstra's shortest-path algorithm. The code is approximately 200 lines, and there are two hot loops as shown in the following result from LLVM-Prospector:

The topmost loop at line 118, `dijkstra:118@1`, spent 80% of all execution time followed by its inner loop, `dijkstra:122@2`. We should look into this loop nest to find parallelization opportunity. Unfortunately, but not surprisingly, the loops of the Dijkstra algorithm have loop-carried flow dependences.

Attempt to parallelize the top-level loop Figure 101 visualizes only loop-carried flow dependences of `dijkstra:118@1`. Showing dependences in function granularity can be easily done by grouping dependences based on variable and function names. Note that there are false dependences from `iPrev`, `iCost`, and `iNode`. However, they are local variables allocated inside the loop, so do not require any modification for parallelization. The critical dependences that determine the parallelizability are loop-carried RAWs between `dequeue` and `enqueue` operations, as shown in Figure 101. In addition, `enqueue` and `dequeue` have their own loop-carried RAWs within themselves.

Based on these facts, an initial parallelization would be inserting a critical section that spans from line 121 to line 134, which is the entire iteration of the loop. Hence, there is no benefit of parallelization.

```
; Hot loops
;  1: dijkstra:118@1, 80.49%
;  2: dijkstra:122@2, 80.42%
;  3: enqueue:68@1, 24.44%
;  4: main:157@1, 19.49%
;  5: main:158@2, 19.49%
```

Figure 100: Loop profiling for Dijkstra in MiBench.

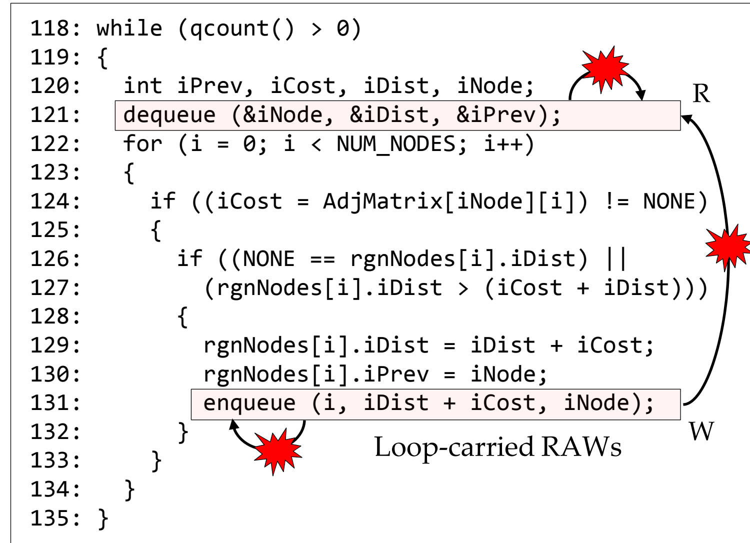


Figure 101: Data dependences of the top-level loop (at line 118) of Dijkstra in MiBench are shown in function granularity.

Parallelizing the inner loop Alternatively, we attempt to parallelize the inner loop, `dijkstra:122@2`. The discovered dependences for this loop are the same with `dijkstra:118@1`, but the interpretation can differ due to the different nesting level. We summarize the dependences in `dijkstra:122@2`:

- Loop-carried false dependences on `iPrev`, `iDist`, `iNode`, and `iCost`, including output dependences.
- Loop-carried dependences in `enqueue`, including flow dependences.

First, regarding the false dependences, the post-analyzer suggests privatization. However, in this case, these variables may be initialized before the loop. Hence, OpenMP's `firstprivate` should be used for them. Second, in order to avoid loop-carried flow dependences, we must check whether the computation order of `enqueue` is important or not. As discussed in Section 7.2.2, a commutative operation can be parallelized by mutex. Otherwise, we must enforce the computation order, for example, by condition variable. However, checking the

```
...
    #pragma omp parallel for firstprivate(iPrev, iDist, iNode, iCost)
122:  for (i = 0; i < NUM_NODES; i++)
...
    #pragma omp critical
140         enqueue(i, iDist + iCost, iNode);
...
```

Figure 102: Parallelizing Dijkstra in MiBench based on the post-analyzer.

commutativity is the future work. In this thesis, we assume that this computation order does not matter. We try to use a critical section to avoid flow dependences.

Figure 102 shows the parallelized Dijkstra solely based on the advice from the post-analyzer. We verify this parallelized code by comparing by results from the serial code. Fortunately, our assumption was correct; this parallelization works.

Assisted annotation for Parallel Prophet The code transformation information from the post-analyzer can also assist the manual annotation process of Parallel Prophet. Since the inner loop is going to be parallelized, estimating speedup would be informative. We first check the loop profiling data: This inner loop invoked 12,796 times and each invocation has 2,000 iterations. Although the invocation count is pretty high, 2,000 iterations may give some profitable parallelization. We then guess the expected speedup based on the serial time from loop profiling. The loop profiling result shows that 24% of the execution time has been spent in enqueue, which is a serial section. Amdahl's law gives a 1.61 speedup for two cores. However, the observed speedup was only 1.04.

We now use Parallel Prophet with the Synthesizer on a quad-core machine with hyper-threading (8 threads). The inner loop is annotated as a parallel loop while enqueue is protected. The estimated speedup from Parallel Prophet using the synthesizer is 1.3 on two cores, which is much closer to the observed speedup. The difference can be explained by the poor implementation of the queue, where

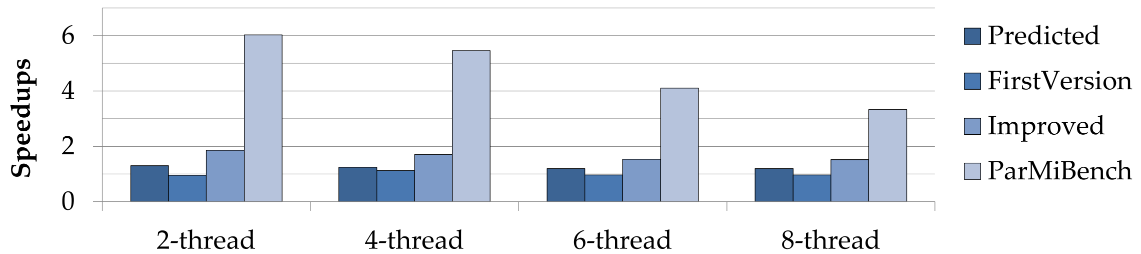


Figure 103: Speedups of parallelized Dijkstra in MiBench: (1) Predicted: Predicted speedups by Parallel Prophet using the Synthesizer; (2) FirstVersion: an OpenMP and critical-section based implementation based on Prospector's advice; (3) Improved: a TBB and concurrent-queue based implementation; (4) ParMiBench: The parallel version from ParMiBench [59].

false sharing occurs in the parallel code. Parallel Prophet does not model cache contention such as false sharing. The predicted speedups are shown in Figure 103.

Improving parallel performance In order to improve the speedup, we use a *concurrent queue*, a lock-free data structure provided by TBB [56], instead of using a naive a lock-based queue. It is well known that a lock-free data structure often provides better speedups. After the modification, a speedup of 1.86 is obtained on a dual-core, as shown in Figure 103. However, we still cannot obtain scalable speedups due to the contention on the shared queue. We can speculate that more speedup would be achievable if the contention of the shared queue is minimized, for example, using per-thread queue. This approach can be found in a parallelized version of Dijkstra by programmers can be also found in ParMiBench [59]. The programmer of ParMiBench attempts to minimize the contention on the queue.

We show the speedup results in Figure 103 from these three parallelized versions and the predictions: (1) Predictions from Parallel Prophet, (2) an OpenMP implementation by Prospector, (3) an improved version using concurrent queue in TBB, and (4) the manually parallelized version in ParMiBench. We perform this experimentation on a desktop machine with a quad-core with hyper-threading. The speedups of (2) and (3), which are assisted by Prospector, yield up to a

```
1   void compressStream(int stream, int zStream)
2   {
3     ...
4     beSetStream(...)
5     ...
6     while (true) {
7       ...
8       initialiseCRC();
9       loadAndRLEsource(...);
10      getFinalCRC(...);
11      ...
12      doReversibleTransformation(...);
13      ...
14      moveToFrontCodeAndSend(...)
15      ...
16    }
17    ...
18  }
```

Figure 104: The structure of 256.bzip2's `compressStream`.

1.86 speedup. The result of ParMiBench on two threads shows a super-linear result: a 6.04 speedup. This is because the code has been significantly changed by the programmers. The original naive queue implementation is replaced with a better approach where each thread can access its local queue. However, this implementation also does not provide good scalability because of inherent contention on the queue. This Dijkstra example illustrates how Prospector can give advice on a program that requires critical sections to be parallelized.

7.3.4 256.bzip2 in SPEC CPU2000

256.bzip2 is a compression and decompression algorithm in SPEC 2000, and the most complex source example in this chapter, around 4,300 lines. We do not have any prior information regarding the parallelizability of the algorithms. We parallelize 256.bzip2's compression logic with pipelining and data-level parallelism. 256.bzip2 will be parallelized by a 3-stage pipeline and the second stage permits further parallel execution as shown in Figure 92.

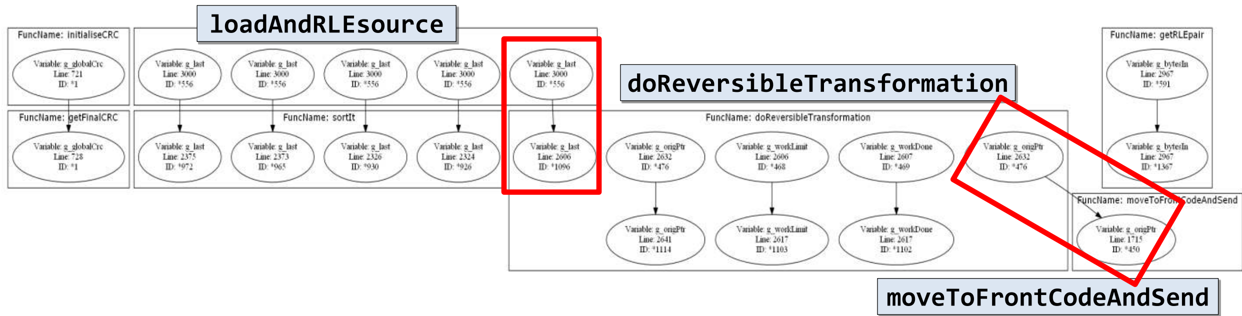


Figure 105: An excerpt of the data-dependence graph of `compressStream` in 256.bzip2: The dependences in the rectangles determine the pipeline stages.

Extracting parallelism Similarly to the previous studies, the loop profiling result gives an initial hint on the parallelization target. Since we are parallelizing the compression logic, the function `compressStream` is the target to parallelize. Because `compressStream` calls several sub-routines, we briefly show the overall structure of the function in Figure 104. `loadAndRLEsource`, `doReversibleTransformation`, and `moveToFrontCodeAndSend` also call other sub-routines.

The next step is analyzing the data dependences in this function. However, this case is challenging because there are hundreds dependence pairs including sub-the routines. Instead, we draw a data-dependence graph. This graph can be drawn by writing a script (Python) that parses the raw results from SD^3. In graph generation, clustering data dependences in function granularity is critical to understand the overall dependence flows. We also removed and collapsed dependence nodes based on variable names to have a more concise graph.

Figure 105 shows a part of the total data-dependence graph of the loop of `compressStream`. This figure shows only flow dependences including both loop-carried and loop-independent ones. We make a group data-dependence pairs from the same variable to make the graph more succinct. As explained in Figure 92, there are loop-independent flow dependences between potential pipeline stages. Hence, we search such patterns in this dependence graph. We highlight important

data dependences that determine the pipeline stages in the (red) rectangles. We can discover two types of parallelism:

- Three-stage pipelining: `loadAndRLEsource()` → `doReversibleTransformation()` → `loadAndRLEsource()`; and
- Data-level parallelism in the second stage: We find that there are no loop-carried flow dependences on `doReversibleTransformation()`.

Writing parallel code We use TBBs pipeline template [56], `tbb::pipeline`, for the parallelization. Since 256.bzip2 is written in C with many global variables, we also need appropriate privatization for them. We describe the steps as follows:

- Create a `struct CompressStreamData`, which holds all required global variables in the original source code. This structure is used to transfer data among pipeline stages. Declaring the global variables in this structure automatically solves the privatization problem in TBB.
- Create three pipeline stages, which are represented as a filter, `tbb::filter`. We further detail how to create these pipeline stages. First, we simply copy and paste the corresponding sub-routines into each pipeline stage class. However, because all global variables are now in `CompressStreamData`, the code accessing the variables is modified. The first `tbb:filter` object creates `CompressStreamData`, and the following two filters obtain this structure as a parameter.
- We are ready. Add these three filters to the runtime, and run the pipeline.

We were able to obtain the correct result by this pipelining. However, we soon observe that the speedup was very low, only 1.1 on a dual-core. This is because the

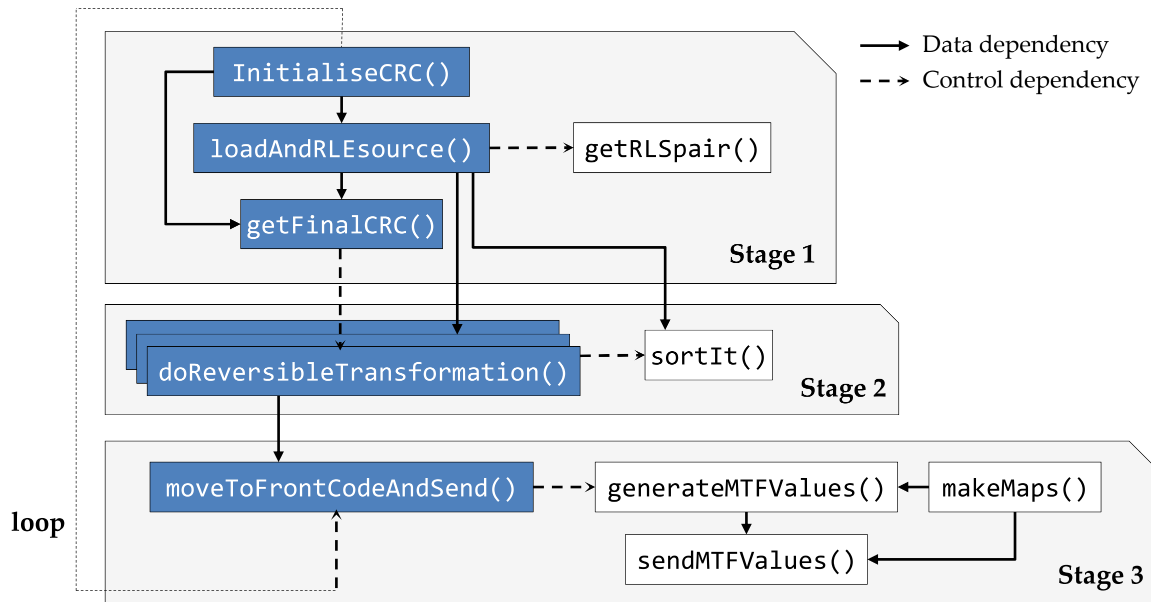


Figure 106: The three-stage pipeline in `compressStream`, where the second stage can be run in parallel.

pipeline is not well balanced: the first and third stages take a little time than the second stage. Balancing pipeline stages is the future work.

However, recall that the second stage does not exhibit any loop-carried flow dependences on itself. Exploiting data-level parallelism for the second stage is simply done by modifying a filter property instead of explicit parallelization. A TBB filter supports three operation modes: (1) `parallel` (multiple items are processed in parallel), (2) `serial_in_order` (only a single item is processed in the same order), and (3) `serial_out_of_order` (processes items one at a time, but in no particular order). We use the `parallel` mode in the second stage. We can also say that the second stage is *stateless* while the others are *stateful* stages.

Figure 106 illustrates the final pipeline and data-level parallelism in 256.bzip2s compression. Multiple blocks in the second stage means that data-level parallelism is exploited in this step.

Measuring speedup We measure the speedups of `compressStream` on a 12-core machine as shown in Figure 107. The speedups are only the improvement over `compressStream`, not the entire program. The speedups show almost the perfect scalability. It is true the most of speedup is from the data-level parallelism in the second stage. However, the parallel execution of the second stage requires the output of the first stage. Without pipelining, we cannot achieve the data-level parallelism because the second stage must await the next item from the first stage.

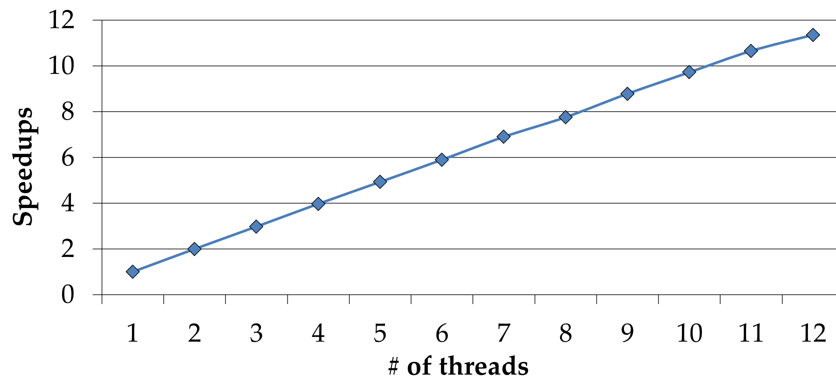


Figure 107: Speedups of `compressStream` in 256.bzip.

CHAPTER VIII

CONCLUSIONS AND FUTURE RESEARCH DIRECTIONS

8.1 Conclusions

The need for parallelization is ever increasing as almost all computing devices have multicore processors. However, writing parallel code is still challenging for many programmers. Among a lot of efforts that reduce the burden of parallel programming, we particularly focus on support from software tools. This thesis proposed Prospector, a programming tool suite that includes four new and enhanced components to assist parallelization. These four components, (1) loop profiler, (2) data-dependence profiler, (3) parallel speedup predictor, (4) post-analyzer, are built on several new program analysis algorithms.

- **Chapter IV** implemented an efficient and robust loop profiler that takes either a binary or source code. Understanding loop behavior is useful in parallelization because loops are main parallelization target. To achieve the optimal performance, our loop profiler minimally instruments instructions to capture loop behavior. However, such optimal instrumentation faced a number of challenges for a binary-level approach. We illustrated problems and propose solutions and heuristics. The performance of the loop profiler was outstanding: less than 4 times slowdown by average against the native execution.

- **Chapter V** introduced SD^3, an efficient and rich data-dependence profiler. Data dependences are the essential information to judge parallelizability. Data-dependence profiling has become an important component for parallelization-assistant tools and related research. However, we showed that the overhead is too high to be employed. We also showed that using simple sampling techniques

is not adequate because the correctness of the profiling could be lost. To address this overhead problem, SD^3 introduced two algorithms.

First, an algorithm that compresses a stream of memory addresses into a stride is proposed. The algorithm includes DYNAMIC-GDC that directly computes dependences with the stride format. Further, a number of data structures and other handling algorithms are needed to the stride-based algorithm works smoothly, such as the dynamic-allocation site optimization, an effective stride-merge technique, and handling killed strides to compute loop-independent dependences correctly.

For the time overhead, SD^3 parallelizes the stride-based algorithm itself. A hybrid approach using both pipelining and data-level parallelism is used. A couple of important design issues including revised stride detection are discussed. Finally, SD^3 successfully profiled 22 SPEC CPU2006 benchmarks on a 16 GB machine.

- **Chapter VI** presented Parallel Prophet, a parallel speedup predictor. Parallel Prophet is based on a dynamic algorithm to predict speedups from serial code, which overcomes previous analytical approaches. Parallel Prophet takes an annotated serial program, where annotations specify parallel and protected sections. Parallel Prophet performs the interval profiling to obtain the lengths of all annotation pairs and the memory profiling to collect specific hardware performance counters related to the bandwidth. The profiling results are recorded in a program tree that is the reflection of program execution.

  The emulators process this program tree to compute the expected parallel execution time. Two emulators are provided. The fast-forwarding method computes the estimated time per each core in a way similar to a simulator.

We explicitly implemented OpenMP's three scheduling policies in the fast-forwarding emulator. However, supporting complex and advanced parallel programming patterns, including nested and recursive parallelism, is infeasible in the FF. Hence, a new emulator called the synthesizer is introduced. This method automatically generates a corresponding real program code from a program tree, and measure an actual speedup on a real machine.

Parallel Prophet finally introduced a memory performance model to estimate the degradation of parallel performance due to increase memory traffic. The memory performance model was built upon an analytical model to estimate the impact of the increased memory traffic in a parallelized program. The coefficients of the model were trained by a special microbenchmark on a target machine.

- **Chapter VII** provided the post-analyzer of SD^3. Raw results from SD^3 are an enumeration of all discovered dependences, which is sometimes hard to interpret to obtain code transformation advice. The post-analyzer performs additional processing to extract several important dependence patterns that can be avoided without significant code modification. The patterns of interest include reductions, privatization, mutex, and pipelining. Four case studies were illustrated to show how post-analyzer works to assist parallelization. Parallel Prophet can also be assisted in the manual annotation step by the post-analyzer.

8.2 Future Research Directions

A number of potential future works based on Prospector concludes this thesis.

8.2.1 Future work for SD^3

One obvious approach to enhance the current SD^3 is to employ more static analysis in the instrumentation time. In particular, the LLVM-based SD^3 has many

potentials. Currently, we perform a basic static analysis to filter instrumentations for certain kinds of variables. For example, trivial induction variables and some function parameters are safe to be skipped for instrumentation. We believe that the idea of multi-slicing [153] can be applied in the static time to complement and enhance the DAS-ID optimization that is based on dynamic tracking.

SD^3 discovers dependences at instruction level, and then the post-analyzer can group them into coarse granularity such as function. However, SD^3 could directly control such granularity to further reduce the overhead. For example, if a programmer only wants to see dependences at function granularity, we may have an optimized table structure. SD^3 can be used to find vectorization opportunity, which can be seen a microscopic parallelism for a loop. In this case, we can do specialization for a quick and low-overhead profiling algorithm.

8.2.2 Future work for Parallel Prophet

Parallel Prophet can be used to diagnose the reasons of poor speedups. When Parallel Prophet performs the FF to estimate speedups, we may infer potential causes for speedups, such as large serial section, load unbalancing, and saturated memory performance.

Parallel Prophet has a potential to be extended to support GPGPU programming model. Suppose a CUDA-like parallelization model, where a loop iteration of a parallel loop forms a CUDA thread or a lane of a SIMD thread, but an input program is a legacy serial loop-based code. Our current annotation system may work, but could require revisions. In the current GPGPU architecture, divergent branches are performance penalties. We may need to record branch behaviors of each parallel task in a program tree.

We can extend the annotation system and emulators to broaden parallel programming models. For example, pipelining can be modeled and predicted:

(1) We introduce new annotations to specify pipeline stages; (2) Writing a simple synthesized emulator. A related work to emulate pipeline performance can be found in [5]. Barriers and condition variables can be modeled in this way.

8.2.3 Future work for the Post-Analyzer

The post-analyzer has many future research directions. We mentioned that the post-analyzer currently does not handle (1) the commutativity test, (2) a generalized case for mutex-based solution, and (3) patterns for barriers and condition variables. Besides these directions, we further discuss new ideas.

Enhancing supports for pipelining We illustrated how the post-analyzer can extract pipeline parallelism with 256.bzip2. However, there are many ways to improve the pipelining, comparing to the previous work [140, 124]. Currently, we assume that a pipeline stage consists of one or more functions. However, legacy code without good refactoring may not be handled by the post-analyzer due to this limitation. We would need a refinement algorithm to suggest potential pipeline stages by forming instructions into a function that can be a candidate for pipeline stage. We also do not explicitly consider the performance issue in the pipelining. One simple solution can be found in [140]. The proposed solution is replicating a pipeline stage. However, this is feasible only when a particular stage is replicable or parallelizable itself, as shown in the function `B` in Figure 92. The post-analyzer can investigate other options: (1) suggesting rebalancing the pipeline stages by combining or splitting the stages, and (2) some computation pipeline stages may be suitable for GPGPU. The post-analyzer will analyze the cost-benefit of such offloading a stage to devices.

Extracting pipeline stages requires a data dependence graph, which is also provided by SD^3. Unfortunately, such graphs are often so large that programmers may unable to understand them. Because many pipeline configurations should be

easy to understand, we believe that an insightful visualization is feasible. We may provide a visualization algorithm such as summarizing a large dependence graph and plotting dependences to show potential pipeline stages.

The previous work only assumes that a compiler consumes the pipeline information, and the pipelined code is generated automatically. For example, vague basic block numbers are used to express pipeline [124]. Programmers could hardly understand such information, especially for when the code is optimized and analyzed. No doubt an automatic code generation is an attractive solution, but we believe that many programmers would like to see how their serial code is changed for the case of pipelining. Programmers then can apply their own optimizations on a parallelized program and have better understanding of the problem. No previous work explicitly considers this case.

Integration with Parallel Prophet The post-analyzer can combine Parallel Prophet with the dependence profiler by assisting the annotation process. We presented a case in the Dijkstra case study.

Providing parallelization advice for transactional memory Transactional memory (TM) [43] and thread-level speculation (TLS) [132] are well-known alternatives to conventional parallel programming. This dissertation does not assume speculative-based approaches because most programs will be running on conventional multiprocessors in the near future. However, we believe that the post-analyzer can exploit the features of the new hardware and software support to provide more useful parallelization advice, such as Sun's ROCK processor [21], IBM Blue Gene/Q [41], Intel Haswell [118], AMD Advanced Synchronization Facility [17], and Intel C++ STM compiler [48].

REFERENCES

[1] ADHIANTO, L., BANERJEE, S., FAGAN, M., KRENTEL, M., MARIN, G., MELLOR-CRUMMEY, J., and TALLENT, N. R., "HPCTOOLKIT: tools for performance analysis of optimized parallel programs http://hpctoolkit.org," *Concurr. Comput. : Pract. Exper.*, vol. 22, pp. 685–701, Apr. 2010.

[2] ADVE, V. S. and VERNON, M. K., "Parallel program performance prediction using deterministic task graph analysis," *ACM Trans. Comput. Syst.*, vol. 22, pp. 94–136, February 2004.

[3] AHO, A. V., LAM, M. S., SETHI, R., and ULLMAN, J. D., *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.

[4] ALEEN, F. and CLARK, N., "Commutativity analysis for software parallelization: letting program transformations see the big picture," in *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, (New York, NY, USA), pp. 241–252, ACM, 2009.

[5] ALEEN, F., SHARIF, M., and PANDE, S., "Input-driven dynamic execution prediction of streaming applications," in *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '10, (New York, NY, USA), pp. 315–324, ACM, 2010.

[6] AMDAHL, G. M., "Validity of the single processor approach to achieving large scale computing capabilities," in *Proceedings of the April 18-20, 1967, spring joint computer conference*, AFIPS '67 (Spring), (New York, NY, USA), pp. 483–485, ACM, 1967.

[7] ANDERSEN, L. O., *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[8] BIENIA, C., KUMAR, S., SINGH, J. P., and LI, K., "The PARSEC benchmark suite: characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, (New York, NY, USA), pp. 72–81, ACM, 2008.

[9] BLUME, W., EIGENMANN, R., FAIGIN, K., GROUT, J., HOEFLINGER, J., PADUA, D. A., PETERSEN, P., POTTENGER, W. M., RAUCHWERGER, L., TU, P., and WEATHERFORD, S., "Polaris: Improving the effectiveness of parallelizing compilers," in *Proceedings of the 7th International Workshop on*

Languages and Compilers for Parallel Computing, LCPC '94, (London, UK, UK), pp. 141–154, Springer-Verlag, 1995.

[10] BLUME, W., EIGENMANN, R., HOEFLINGER, J., PADUA, D., PETERSEN, P., RAUCHWERGER, L., and TU, P., "Automatic detection of parallelism: A grand challenge for high-performance computing," *IEEE Parallel Distrib. Technol.*, vol. 2, pp. 37–47, Sept. 1994.

[11] BLUMOFE, R. D., JOERG, C. F., KUSZMAUL, B. C., LEISERSON, C. E., RANDALL, K. H., and ZHOU, Y., "Cilk: an efficient multithreaded runtime system," in *Proceedings of the fifth ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPOPP '95, (New York, NY, USA), pp. 207–216, ACM, 1995.

[12] *Boost C++ Libraries*. http://www.boost.org/.

[13] BULL, J. M. and O'NEILL, D., "A microbenchmark suite for OpenMP 2.0," *SIGARCH Comput. Archit. News*, vol. 29, pp. 41–48, December 2001.

[14] CENG, J., CASTRILLON, J., SHENG, W., SCHARWÄCHTER, H., LEUPERS, R., ASCHEID, G., MEYR, H., ISSHIKI, T., and KUNIEDA, H., "MAPS: an integrated framework for mpsoc application parallelization," in *Proceedings of the 45th annual Design Automation Conference*, DAC '08, (New York, NY, USA), pp. 754–759, ACM, 2008.

[15] CHAKARAVARTHY, V. T., "New results on the computability and complexity of points–to analysis," in *Proceedings of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '03, (New York, NY, USA), pp. 115–125, ACM, 2003.

[16] CHEN, T., LIN, J., DAI, X., HSU, W.-C., and YEW, P.-C., "Data dependence profiling for speculative optimizations," in *Compiler Construction* (DUESTERWALD, E., ed.), vol. 2985 of *Lecture Notes in Computer Science*, pp. 2733–2733, Springer Berlin / Heidelberg, 2004.

[17] CHUNG, J., YEN, L., DIESTELHORST, S., POHLACK, M., HOHMUTH, M., CHRISTIE, D., and GROSSMAN, D., "ASF: AMD64 extension for lock-free data structures and transactional memory," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 39–50, IEEE Computer Society, 2010.

[18] CORMEN, T. H., LEISERSON, C. E., RIVEST, R. L., and STEIN, C., *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd ed., 2009.

[19] CriticalBlue, *Prism: an analysis exploration and verification environment for software implementation and optimization on multicore architectures*. http://www.criticalblue.com.

[20] DAS, D. and WU, P., "Experiences of using a dependence profiler to assist parallelization for multi-cores," in *IPDPS Workshops*, pp. 1–8, 2010.

[21] DICE, D., LEV, Y., MOIR, M., and NUSSBAUM, D., "Early experience with a commercial hardware transactional memory implementation," in *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, (New York, NY, USA), pp. 157–168, ACM, 2009.

[22] DIMAKOPOULOS, V. V., HADJIDOUKAS, P. E., and PHILOS, G. C., "A microbenchmark study of OpenMP overheads under nested parallelism," in *Proceedings of the 4th international conference on OpenMP in a new era of parallelism*, IWOMP'08, (Berlin, Heidelberg), pp. 1–12, Springer-Verlag, 2008.

[23] DING, C., SHEN, X., KELSEY, K., TICE, C., HUANG, R., and ZHANG, C., "Software behavior oriented parallelization," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, (New York, NY, USA), pp. 223–234, ACM, 2007.

[24] DORTA, A. J., RODRIGUEZ, C., SANDE, F. D., and GONZALEZ-ESCRIBANO, A., "The OpenMP source code repository," in *Proceedings of the 13th Euromicro Conference on Parallel, Distributed and Network-Based Processing*, (Washington, DC, USA), pp. 244–250, IEEE Computer Society, 2005.

[25] DU, Z.-H., LIM, C.-C., LI, X.-F., YANG, C., ZHAO, Q., and NGAI, T.-F., "A cost-driven compilation framework for speculative parallelization of sequential programs," in *Proceedings of the ACM SIGPLAN 2004 conference on Programming Language Design and Implementation*, PLDI '04, (New York, NY, USA), pp. 71–81, ACM, 2004.

[26] DWARF Standards Committee, *The DWARF Debugging Standard* . `http://dwarfstd.org/`.

[27] EADLINE, D., "Concurrent and parallel are not the same," *Linux Magazine*, July 2009.

[28] ESMAEILZADEH, H., BLEM, E., ST. AMANT, R., SANKARALINGAM, K., and BURGER, D., "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th annual international symposium on Computer architecture*, ISCA '11, (New York, NY, USA), pp. 365–376, ACM, 2011.

[29] EYERMAN, S. and EECKHOUT, L., "Modeling critical sections in Amdahl's law and its implications for multicore design," in *Proceedings of the 37th annual international symposium on Computer architecture*, ISCA '10, (New York, NY, USA), pp. 362–370, ACM, 2010.

[30] FAXÉN, K.-F., POPOV, K., JANSSON, S., and ALBERTSSON, L., "Embla - data dependence profiling for parallel programming," *Complex, Intelligent*

and Software Intensive Systems, International Conference, vol. 0, pp. 780–785, 2008.

[31] FRIGO, M., LEISERSON, C. E., and RANDALL, K. H., "The implementation of the Cilk-5 multithreaded language," in *Proceedings of the ACM SIGPLAN 1998 conference on Programming language design and implementation*, PLDI '98, (New York, NY, USA), pp. 212–223, ACM, 1998.

[32] *Future<V>interface in java.util.concurrent*. http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/Future.html/.

[33] GARCIA, S., JEON, D., LOUIE, C. M., and TAYLOR, M. B., "Kremlin: rethinking and rebooting gprof for the multicore age," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), pp. 458–469, ACM, 2011.

[34] GRAHAM, S. L., KESSLER, P. B., and MCKUSICK, M. K., "Gprof: A call graph execution profiler," in *Proceedings of the 1982 SIGPLAN symposium on Compiler construction*, SIGPLAN '82, (New York, NY, USA), pp. 120–126, ACM, 1982.

[35] GROPP, W., LUSK, E., and SKJELLUM, A., *Using MPI: portable parallel programming with the message-passing interface*. Cambridge, MA, USA: MIT Press, 1994.

[36] GUSTAFSON, J. L., "Reevaluating Amdahl's law," *Commun. ACM*, vol. 31, pp. 532–533, May 1988.

[37] GUTHAUS, M. R., RINGENBERG, J. S., ERNST, D., AUSTIN, T. M., MUDGE, T., and BROWN, R. B., "MiBench: A free, commercially representative embedded benchmark suite," in *Proceedings of the Workload Characterization, 2001. WWC-4. 2001 IEEE International Workshop*, (Washington, DC, USA), pp. 3–14, IEEE Computer Society, 2001.

[38] GUZ, Z., BOLOTIN, E., KEIDAR, I., KOLODNY, A., MENDELSON, A., and WEISER, U., "Many-core vs. many-thread machines: Stay away from the valley," *Computer Architecture Letters*, vol. 8, pp. 25 –28, jan. 2009.

[39] HA, J., ARNOLD, M., BLACKBURN, S. M., and MCKINLEY, K. S., "A concurrent dynamic analysis framework for multicore hardware," in *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, (New York, NY, USA), pp. 155–174, ACM, 2009.

[40] HARDEKOPF, B. and LIN, C., "The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '07, (New York, NY, USA), pp. 290–299, ACM, 2007.

[41] HARING, R. A., OHMACHT, M., FOX, T. W., GSCHWIND, M. K., SATTERFIELD, D. L., SUGAVANAM, K., COTEUS, P. W., HEIDELBERGER, P., BLUMRICH, M. A., WISNIEWSKI, R. W., GARA, A., CHIU, G. L.-T., BOYLE, P. A., CHIST, N. H., and KIM, C., "The ibm blue gene/q compute chip," *IEEE Micro*, vol. 32, pp. 48–60, 2012.

[42] HE, Y., LEISERSON, C. E., and LEISERSON, W. M., "The Cilkview scalability analyzer," in *Proceedings of the 22nd ACM symposium on Parallelism in algorithms and architectures*, SPAA '10, (New York, NY, USA), pp. 145–156, ACM, 2010.

[43] HERLIHY, M. and MOSS, J. E. B., "Transactional memory: architectural support for lock-free data structures," in *Proceedings of the 20th annual international symposium on computer architecture*, ISCA '93, (New York, NY, USA), pp. 289–300, ACM, 1993.

[44] HILL, M. D. and MARTY, M. R., "Amdahl's law in the multicore era," *Computer*, vol. 41, pp. 33–38, July 2008.

[45] HIND, M., "Pointer analysis: haven't we solved this problem yet?," in *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, PASTE '01, (New York, NY, USA), pp. 54–61, ACM, 2001.

[46] IBANEZ, L., SCHROEDER, W., NG, L., and CATES, J., *The ITK Software Guide*. Kitware, Inc. ISBN 1-930934-10-6, http://www.itk.org/ItkSoftwareGuide.pdf, first ed., 2003.

[47] The Innovative Computing Laboratory, University of Tennessee, *PAPI: Performance Application Programming Interface*. http://icl.cs.utk.edu/papi.

[48] Intel Corporation, *Intel C++ STM Compiler, Prototype Edition*. http://software.intel.com/en-us/articles/intel-c-stm-compiler-prototype-edition/.

[49] Intel Corporation, *Intel Cilk Plus*. http://software.intel.com/en-us/articles/intel-cilk-plus/.

[50] Intel Corporation, *Intel Compilers*. http://software.intel.com/en-us/intel-compilers/.

[51] Intel Corporation, *Intel Parallel Advisor*. http://software.intel.com/en-us/articles/intel-parallel-advisor/.

[52] Intel Corporation, *Intel Parallel Amplifier*. http://software.intel.com/en-us/articles/intel-parallel-amplifier/.

[53] Intel Corporation, *Intel Parallel Inspector*. http://software.intel.com/en-us/articles/intel-parallel-inspector/.

[54] Intel Corporation, *Intel Parallel Studio*. http://software.intel.com/en-us/intel-parallel-studio-home/.

[55] Intel Corporation, *Intel Performance Counter Monitor - A better way to measure CPU utilization*. http://software.intel.com/en-us/articles/intel-performance-counter-monitor/.

[56] Intel Corporation, *Intel Threading Building Blocks*. http://www.threadingbuildingblocks.org/.

[57] Intel Corporation, *Intel VTune Amplifier XE*. http://software.intel.com/en-us/articles/intel-vtune-amplifier-xe/.

[58] The International Technology Roadmap for Semiconductors, *The 2011 edition of the ITRS*. http://www.itrs.net/Links/2011ITRS/Home2011.htm/.

[59] IQBAL, S. M. Z., LIANG, Y., and GRAHN, H., "ParMiBench - an open-source benchmark for embedded multiprocessor systems," *IEEE Comput. Archit. Lett.*, vol. 9, pp. 45–48, July 2010.

[60] ISO/IEC, *ISO/IEC 9899:1999, Programming Language - C*. http://www.open-std.org/jtc1/sc22/wg14.

[61] ISO/IEC, *The C++ Standards Committee*. http://www.open-std.org/jtc1/sc22/wg21/.

[62] JEON, D., GARCIA, S., LOUIE, C., and TAYLOR, M. B., "Kismet: parallel speedup estimates for serial programs," in *Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications*, OOPSLA '11, (New York, NY, USA), pp. 519–536, ACM, 2011.

[63] JIN, G., SONG, L., ZHANG, W., LU, S., and LIBLIT, B., "Automated atomicity-violation fixing," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), pp. 389–400, ACM, 2011.

[64] JIN, H., JIN, H., FRUMKIN, M., FRUMKIN, M., YAN, J., and YAN, J., "The OpenMP implementation of nas parallel benchmarks and its performance," tech. rep., NASA, 1999.

[65] JOHNSON, N. P., KIM, H., PRABHU, P., ZAKS, A., and AUGUST, D. I., "Speculative separation for privatization and reductions," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '12, (New York, NY, USA), pp. 359–370, ACM, 2012.

[66] KARP, A. H. and FLATT, H. P., "Measuring parallel processor performance," *Commun. ACM*, vol. 33, May 1990.

[67] Khronos OpenCL Working Group, *The OpenCL Specification Version 1.2, 2011*. http://www.khronos.org/opencl/.

[68] KIM, D. and RINARD, M. C., "Verification of semantic commutativity conditions and inverse operations on linked data structures," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, PLDI '11, (New York, NY, USA), pp. 528–541, ACM, 2011.

[69] KIM, M., KIM, H., and LUK, C.-K., "Prospector: Helping parallel programming by a data-dependence profile," in *The 2nd USENIX conference on Hot topics in parallelism*, HotPar '10, (Berkeley, CA, USA), USENIX Association, 2010.

[70] KIM, M., KIM, H., and LUK, C.-K., "SD^3: A scalable approach to dynamic data-dependence profiling," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 535–546, IEEE Computer Society, 2010.

[71] KIM, M., KUMAR, P., KIM, H., and BRETT, B., "Predicting potential speedup of serial code via lightweight profiling and emulations with memory performance model," in *Proceedings of the 2012 IEEE International Parallel & Distributed Processing Symposium*, IPDPS '12, (Washington, DC, USA), IEEE Computer Society, 2012.

[72] KNOBE, K. and SARKAR, V., "Array ssa form and its use in parallelization," in *POPL '98: Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, (New York, NY, USA), pp. 107–120, ACM, 1998.

[73] KONG, X., KLAPPHOLZ, D., and PSARRIS, K., "The I test: An improved dependence test for automatic parallelization and vectorization," *IEEE Trans. Parallel Distrib. Syst.*, vol. 2, pp. 342–349, July 1991.

[74] KOTHA, A., ANAND, K., SMITHSON, M., YELLAREDDY, G., and BARUA, R., "Automatic parallelization in a binary rewriter," in *Proceedings of the 2010 43rd Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO '43, (Washington, DC, USA), pp. 547–557, IEEE Computer Society, 2010.

[75] LARUS, J. R., "Loop-level parallelism in numeric and symbolic programs," *IEEE Trans. Parallel Distrib. Syst.*, vol. 4, pp. 812–826, July 1993.

[76] LARUS, J. R., "Whole program paths," in *Proceedings of the ACM SIGPLAN 1999 conference on Programming Language Design and Implementation*, PLDI '99, (New York, NY, USA), pp. 259–269, ACM, 1999.

[77] LATTNER, C. and ADVE, V., "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime*

Optimization, CGO '04, (Washington, DC, USA), pp. 75–, IEEE Computer Society, 2004.

[78] LATTNER, C., LENHARTH, A., and ADVE, V., "Making context-sensitive points-to analysis with heap cloning practical for the real world," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '07, (New York, NY, USA), pp. 278–289, ACM, 2007.

[79] LEE, C., POTKONJAK, M., and MANGIONE-SMITH, W. H., "MediaBench: a tool for evaluating and synthesizing multimedia and communicatons systems," in *Proceedings of the 30th annual ACM/IEEE international symposium on Microarchitecture*, MICRO 30, (Washington, DC, USA), pp. 330–335, IEEE Computer Society, 1997.

[80] LEIJEN, D., SCHULTE, W., and BURCKHARDT, S., "The design of a task parallel library," in *Proceeding of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, (New York, NY, USA), pp. 227–242, ACM, 2009.

[81] LIN, J., CHEN, T., HSU, W.-C., YEW, P.-C., JU, R. D.-C., NGAI, T.-F., and CHAN, S., "A compiler framework for speculative optimizations," *ACM Trans. Archit. Code Optim.*, vol. 1, pp. 247–271, September 2004.

[82] LIU, W., TUCK, J., CEZE, L., AHN, W., STRAUSS, K., RENAU, J., and TORRELLAS, J., "POSH: a TLS compiler that exploits program structure," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and Practice of Parallel Programming*, PPoPP '06, (New York, NY, USA), pp. 158–167, ACM, 2006.

[83] LU, S., PARK, S., SEO, E., and ZHOU, Y., "Learning from mistakes: a comprehensive study on real world concurrency bug characteristics," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, (New York, NY, USA), pp. 329–339, ACM, 2008.

[84] LUK, C.-K., "Tolerating memory latency through software-controlled pre-execution in simultaneous multithreading processors," in *Proceedings of the 28th annual international symposium on Computer architecture*, ISCA '01, (New York, NY, USA), pp. 40–51, ACM, 2001.

[85] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., and HAZELWOOD, K., "Pin: building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 2005 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '05, (New York, NY, USA), pp. 190–200, ACM, 2005.

[86] MARATHE, J., MUELLER, F., MOHAN, T., MCKEE, S. A., DE SUPINSKI, B. R., and YOO, A., "METRIC: Memory tracing via dynamic binary rewriting to identify cache inefficiencies," *ACM Trans. Program. Lang. Syst.*, vol. 29, April 2007.

[87] MARINO, D., MUSUVATHI, M., and NARAYANASAMY, S., "LiteRace: effective sampling for lightweight data-race detection," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '09, (New York, NY, USA), pp. 134–143, ACM, 2009.

[88] MELLOR-CRUMMEY, J., FOWLER, R., and WHALLEY, D., "Tools for application-oriented performance tuning," in *Proceedings of the 15th international conference on Supercomputing*, ICS '01, (New York, NY, USA), pp. 154–165, ACM, 2001.

[89] Microsoft, *C++ Accelerated Massive Parallelism (C++ AMP) Overview*. http://msdn.microsoft.com/en-us/library/hh265136.aspx.

[90] Microsoft, *CHESS: Find and Reproduce Heisenbugs in Concurrent Programs*. http://research.microsoft.com/en-us/projects/chess/.

[91] Microsoft, *Concurrency Visualizer*. http://msdn.microsoft.com/en-us/library/dd537632.aspx/.

[92] Microsoft, *PDB: Program database*. http://msdn.microsoft.com/en-us/library/t6tay6cz.aspx/.

[93] MOORE, G. E., "Cramming more components onto integrated circuits," *Electronics*, vol. 38, p. 4, 1965.

[94] MOSELEY, T., CONNORS, D. A., GRUNWALD, D., and PERI, R., "Identifying potential parallelism via loop-centric profiling," in *Proceedings of the 4th international conference on Computing frontiers*, CF '07, (New York, NY, USA), pp. 143–152, ACM, 2007.

[95] MOSELEY, T., SHYE, A., REDDI, V. J., GRUNWALD, D., and PERI, R., "Shadow profiling: Hiding instrumentation costs with parallelism," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, (Washington, DC, USA), pp. 198–208, IEEE Computer Society, 2007.

[96] MUCCI, P. J., BROWNE, S., DEANE, C., and HO, G., "PAPI: A portable interface to hardware performance counters," in *In Proceedings of the Department of Defense HPCMP Users Group Conference*, pp. 7–10, 1999.

[97] MUCHNICK, S. S., *Advanced compiler design and implementation*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997.

[98] NETHERCOTE, N. and SEWARD, J., "Valgrind: a framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 2007 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '07, (New York, NY, USA), pp. 89–100, ACM, 2007.

[99] NEVILL-MANNING, C. G. and WITTEN, I. H., "Linear-time, incremental hierarchy inference for compression," in *Proceedings of the Conference on Data Compression*, DCC '97, (Washington, DC, USA), pp. 3–, IEEE Computer Society, 1997.

[100] NICKOLLS, J., BUCK, I., GARLAND, M., and SKADRON, K., "Scalable parallel programming with cuda," *Queue*, vol. 6, pp. 40–53, Mar. 2008.

[101] NIGHTINGALE, E. B., PEEK, D., CHEN, P. M., and FLINN, J., "Parallelizing security checks on commodity hardware," in *Proceedings of the 13th international conference on Architectural support for programming languages and operating systems*, ASPLOS XIII, (New York, NY, USA), pp. 308–318, ACM, 2008.

[102] NVIDIA, *Tegra 2 Mobile Processor*. http://www.nvidia.com/object/tegra-superchip.html.

[103] *OmpSCR: OpenMP Source Code Repository*. http://sourceforge.net/projects/ompscr/.

[104] *OpenMP for construct*. http://msdn.microsoft.com/en-us/library/b5b5b6eb.aspx.

[105] *OProfile*. http://oprofile.sourceforge.net/about/.

[106] OTTONI, G., RANGAN, R., STOLER, A., and AUGUST, D. I., "Automatic thread extraction with decoupled software pipelining," in *Proceedings of the 38th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 38, (Washington, DC, USA), pp. 105–118, IEEE Computer Society, 2005.

[107] PARK, S., LU, S., and ZHOU, Y., "CTrigger: exposing atomicity violation bugs from their hiding places," in *Proceeding of the 14th international conference on Architectural support for programming languages and operating systems*, ASPLOS '09, (New York, NY, USA), pp. 25–36, ACM, 2009.

[108] PETERSEN, P. M. and PADUA, D. A., "Static and dynamic evaluation of data dependence analysis," in *Proceedings of the 7th international conference on Supercomputing*, ICS '93, (New York, NY, USA), pp. 107–116, ACM, 1993.

[109] The Portland Group, *PGI Workstation: Fortran and C/C++ for 64-bit x64 processor-based systems*. http://www.pgroup.com/products/pgiworkstation.htm/.

[110] POSTIFF, M., TYSON, G., and MUDGE, T., "Performance limits of trace caches," *Journal of Instruction-Level Parallelism*, vol. 1, 1999.

[111] *Parallel Patterns Library (PPL)*. http://msdn.microsoft.com/en-us/library/dd492418.aspx.

[112] PRABHU, P., GHOSH, S., ZHANG, Y., JOHNSON, N. P., and AUGUST, D. I., "Commutative set: a language extension for implicit parallel programming," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), pp. 1–11, ACM, 2011.

[113] PRICE, G. D., GIACOMONI, J., and VACHHARAJANI, M., "Visualizing potential parallelism in sequential programs," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*, PACT '08, (New York, NY, USA), pp. 82–90, ACM, 2008.

[114] PUGH, W., "Definitions of dependence distance," *ACM Lett. Program. Lang. Syst.*, vol. 1, pp. 261–265, Sept. 1992.

[115] RAMASESHAN, R., "Trace-based dependence analysis for speculative loop optimizations," Master's thesis, North Carolina State University, 2007.

[116] RANGAN, R., VACHHARAJANI, N., VACHHARAJANI, M., and AUGUST, D. I., "Decoupled software pipelining with the synchronization array," in *Proceedings of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, (Washington, DC, USA), pp. 177–188, IEEE Computer Society, 2004.

[117] RAUCHWERGER, L. and PADUA, D., "The lrpd test: speculative run-time parallelization of loops with privatization and reduction parallelization," in *Proceedings of the ACM SIGPLAN 1995 conference on Programming language design and implementation*, PLDI '95, (New York, NY, USA), pp. 218–232, ACM, 1995.

[118] REINDERS, J., "Transactional synchronization in Haswell," *Intel Software Network*, February 2012.

[119] RINARD, M. C. and DINIZ, P. C., "Commutativity analysis: a new analysis technique for parallelizing compilers," *ACM Trans. Program. Lang. Syst.*, vol. 19, pp. 942–991, November 1997.

[120] ROGERS, A., CARLISLE, M. C., REPPY, J. H., and HENDREN, L. J., "Supporting dynamic data structures on distributed-memory machines," *ACM Trans. Program. Lang. Syst.*, vol. 17, pp. 233–263, March 1995.

[121] Rogue Wave Software, *ThreadSpotter*. http://www.roguewave.com/products/threadspotter.aspx/.

[122] RONSSE, M. and DE BOSSCHERE, K., "RecPlay: a fully integrated practical record/replay system," *ACM Trans. Comput. Syst.*, vol. 17, no. 2, pp. 133–152, 1999.

[123] RUL, S., VANDIERENDONCK, H., and DE BOSSCHERE, K., "Extracting coarse-grain parallelism in general-purpose programs," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, (New York, NY, USA), pp. 281–282, ACM, 2008.

[124] RUL, S., VANDIERENDONCK, H., and DE BOSSCHERE, K., "A profile-based tool for finding pipeline parallelism in sequential programs," *Parallel Comput.*, vol. 36, pp. 531–551, September 2010.

[125] SACK, P., BLISS, B. E., MA, Z., PETERSEN, P., and TORRELLAS, J., "Accurate and efficient filtering for the intel thread checker race detector," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, ASID '06, (New York, NY, USA), pp. 34–41, ACM, 2006.

[126] SAGIV, M., REPS, T., and WILHELM, R., "Parametric shape analysis via 3-valued logic," in *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '99, (New York, NY, USA), pp. 105–118, ACM, 1999.

[127] SAVAGE, S., BURROWS, M., NELSON, G., SOBALVARRO, P., and ANDERSON, T., "Eraser: a dynamic data race detector for multithreaded programs," *ACM Trans. Comput. Syst.*, vol. 15, no. 4, pp. 391–411, 1997.

[128] SCHREIBER, S. B., *Undocumented Windows 2000 secrets: a programmer's cookbook*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[129] SEILER, L., CARMEAN, D., SPRANGLE, E., FORSYTH, T., ABRASH, M., DUBEY, P., JUNKINS, S., LAKE, A., SUGERMAN, J., CAVIN, R., ESPASA, R., GROCHOWSKI, E., JUAN, T., and HANRAHAN, P., "Larrabee: a many-core x86 architecture for visual computing," in *ACM SIGGRAPH 2008 papers*, SIGGRAPH '08, (New York, NY, USA), pp. 18:1–18:15, ACM, 2008.

[130] SEREBRYANY, K. and ISKHODZHANOV, T., "ThreadSanitizer: data race detection in practice," in *Proceedings of the Workshop on Binary Instrumentation and Applications*, WBIA '09, (New York, NY, USA), pp. 62–71, ACM, 2009.

[131] SHIRAKO, J., PEIXOTTO, D. M., SARKAR, V., and SCHERER, W. N., "Phasers: a unified deadlock-free construct for collective and point-to-point synchronization," in *Proceedings of the 22nd annual international conference on Supercomputing*, ICS '08, (New York, NY, USA), pp. 277–288, ACM, 2008.

[132] SOHI, G. S., BREACH, S. E., and VIJAYKUMAR, T. N., "Multiscalar processors," in *Proceedings of the 22nd annual international symposium on*

Computer architecture, ISCA '95, (New York, NY, USA), pp. 414–425, ACM, 1995.

[133] Standard Performance Evaluation Corporation, *SPEC CPU2000*. http://www.spec.org/cpu2000/.

[134] Standard Performance Evaluation Corporation, *SPEC CPU2006*. http://www.spec.org/cpu2006/.

[135] STEENSGAARD, B., "Points-to analysis in almost linear time," in *Proceedings of the 23rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '96, (New York, NY, USA), pp. 32–41, ACM, 1996.

[136] STEFFAN, J. G., COLOHAN, C. B., ZHAI, A., and MOWRY, T. C., "A scalable approach to thread-level speculation," in *Proceedings of the 27th annual international symposium on Computer architecture*, ISCA '00, (New York, NY, USA), pp. 1–12, ACM, 2000.

[137] SUTTER, H., "The free lunch is over," *Dr. Dobb's Journal*, vol. 30, March 2005.

[138] The OpenMP Architecture Review Board, *OpenMP*. http://openmp.org/.

[139] THIES, W., CHANDRASEKHAR, V., and AMARASINGHE, S., "A practical approach to exploiting coarse-grained pipeline parallelism in C programs," in *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 40, (Washington, DC, USA), pp. 356–369, IEEE Computer Society, 2007.

[140] TOURNAVITIS, G. and FRANKE, B., "Semi-automatic extraction and exploitation of hierarchical pipeline parallelism using profiling information," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, (New York, NY, USA), pp. 377–388, ACM, 2010.

[141] TOURNAVITIS, G., WANG, Z., FRANKE, B., and O'BOYLE, M. F., "Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping," in *Proceedings of the 2009 ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '09, (New York, NY, USA), pp. 177–187, ACM, 2009.

[142] TU, P. and PADUA, D. A., "Automatic array privatization," in *Proceedings of the 6th International Workshop on Languages and Compilers for Parallel Computing*, (London, UK, UK), pp. 500–521, Springer-Verlag, 1994.

[143] UDUPA, A., RAJAN, K., and THIES, W., "ALTER: exploiting breakable dependences for parallelization," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '11, (New York, NY, USA), pp. 480–491, ACM, 2011.

[144] VANDIERENDONCK, H., RUL, S., and DE BOSSCHERE, K., "The paralax infrastructure: automatic parallelization with a helping hand," in *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, PACT '10, (New York, NY, USA), pp. 389–400, ACM, 2010.

[145] Vector Fabrics, *Pareon: Optimize applications for multicore phones, tablets and x86 in hours.* `http://www.vectorfabrics.com/`.

[146] VENKATESH, G., SAMPSON, J., GOULDING, N., GARCIA, S., BRYKSIN, V., LUGO-MARTINEZ, J., SWANSON, S., and TAYLOR, M. B., "Conservation cores: reducing the energy of mature computations," in *Proceedings of the fifteenth edition of ASPLOS on Architectural support for programming languages and operating systems*, ASPLOS '10, (New York, NY, USA), pp. 205–218, ACM, 2010.

[147] VOLOS, H., TACK, A. J., SWIFT, M. M., and LU, S., "Applying transactional memory to concurrency bugs," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '12, (New York, NY, USA), pp. 211–222, ACM, 2012.

[148] VON PRAUN, C., BORDAWEKAR, R., and CASCAVAL, C., "Modeling optimistic concurrency using quantitative dependence analysis," in *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '08, (New York, NY, USA), pp. 185–196, ACM, 2008.

[149] WALLACE, S. and HAZELWOOD, K., "SuperPin: Parallelizing dynamic instrumentation for real-time performance," in *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, (Washington, DC, USA), pp. 209–220, IEEE Computer Society, 2007.

[150] WILSON, R. P., FRENCH, R. S., WILSON, C. S., AMARASINGHE, S. P., ANDERSON, J. M., TJIANG, S. W. K., LIAO, S.-W., TSENG, C.-W., HALL, M. W., LAM, M. S., and HENNESSY, J. L., "Suif: an infrastructure for research on parallelizing and optimizing compilers," *SIGPLAN Not.*, vol. 29, pp. 31–37, Dec. 1994.

[151] WU, P., KEJARIWAL, A., and CAŞCAVAL, C., "Languages and compilers for parallel computing," ch. Compiler-Driven Dependence Profiling to Guide Program Parallelization, pp. 232–248, Berlin, Heidelberg: Springer-Verlag, 2008.

[152] YAO, E., BAO, Y., TAN, G., and CHEN, M., "Extending amdahl's law in the multicore era," *SIGMETRICS Perform. Eval. Rev.*, vol. 37, pp. 24–26, October 2009.

[153] YU, H. and LI, Z., "Multi-slicing: A compiler-supported parallel approach to data dependence profiling," in *Proceedings of the 2011 International*

Symposium on Software Testing and Analysis, ISSTA '12, (New York, NY, USA), ACM, 2012.

[154] YU, Y., RODEHEFFER, T., and CHEN, W., "RaceTrack: efficient detection of data race conditions via adaptive tracking," in *Proceedings of the twentieth ACM symposium on Operating systems principles*, SOSP '05, (New York, NY, USA), pp. 221–234, ACM, 2005.

[155] ZHANG, X. and GUPTA, R., "Whole execution traces," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, (Washington, DC, USA), pp. 105–116, IEEE Computer Society, 2004.

[156] ZHANG, X., NAVABI, A., and JAGANNATHAN, S., "Alchemist: A transparent dependence distance profiling infrastructure," in *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, (Washington, DC, USA), pp. 47–58, IEEE Computer Society, 2009.

[157] ZHAO, Q., CUTCUTACHE, I., and WONG, W.-F., "PiPA: pipelined profiling and analysis on multi-core systems," in *Proceedings of the 6th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '08, (New York, NY, USA), pp. 185–194, ACM, 2008.