RICE UNIVERSITY

Application Insight Through Performance Modeling

by

Gabriel Marin

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

Doctor of Philosophy

Approved, Thesis Committee:

John Mellor-Crummey, Chair
Associate Professor, Computer Science

Walid Taha
Assistant Professor of Computer Science

Peter J. Varman , Professor of
Electrical and Computer Engineering

Houston, Texas

December 2007

Application Insight Through Performance Modeling

Gabriel Marin

Abstract

Tuning the performance of applications requires understanding the interactions between code and target architecture. Hardware counters, present in all modern processors, can identify possible causes of performance problems and can pinpoint sections of code that execute at a low fraction of machine peak performance. However, the information provided by hardware counters is often insufficient to understand the causes of poor performance or to realistically estimate the potential for performance improvement.

This thesis presents techniques to measure and model application characteristics independent of the target architecture. Using information gathered from both static and dynamic analysis, this approach not only makes accurate predictions about the behavior of an application on a target architecture for different inputs, but also provides guidance for tuning by highlighting the factors that limit performance at different points in a program. We introduce several new performance analysis metrics that estimate the maximum gain expected from tuning different aspects of an application, or from using hardware accelerator coprocessors. Our approach models the most important factors affecting application performance and provides estimates of unfulfilled performance potential due to a mismatch between an application's characteristics and

the resources present on the target architecture. We model an application's instruction execution cost and memory hierarchy utilization, and we identify performance problems arising from insufficient instruction-level parallelism or poor data locality.

To demonstrate the utility of this approach, this thesis presents the results of analyzing and tuning two scientific applications. For Sweep3D, a three-dimensional Cartesian geometry neutron transport code benchmark from the DOEs Accelerated Strategic Computing Initiative, our analysis identified opportunities for improvement that shortened execution time by 66% on an Itanium2-based system. For the Gyrokinetic Toroidal Code (GTC) from Princeton Plasma Physics Laboratory, a particle-in-cell code that simulates turbulent transport of particles and energy in burning plasma, our techniques identified opportunities for improvement that shortened execution time by 33% on the same Itanium2 system.

Acknowledgements

I am grateful to John Mellor-Crummey, my advisor through most of my graduate studies at Rice, for both his technical guidance and financial support during the years. His door has been always open for answering questions, and on more than one occasion he provided feed-back or help with my work at late hours or on weekends. John's careful reading of the draft of my thesis greatly improved the clarity of the final version.

I want to thank Walid Taha and Peter Varman for their contribution as members of the committee of this thesis. They have been very responsive to my questions and accommodating to my situation. I am indebted to Ken Kennedy and Moshe Vardi for helping me at a very difficult moment in my second year at Rice. Without their support I would have left the graduate program after my first year.

I thank Richard Hanson for his advice regarding modeling strategies in Matlab. I am thankful to Adolfy Hoisie, Darren Kerbyson, Olaf Lubeck and Scott Pakin for their insights during my internship with the Performance and Architecture Laboratory at Los Alamos National Laboratory.

I am fortunate to have studied in a small department with professors and students working in a close environment and in a unique building. I want to thank Nathan Tallent for providing comments on my thesis, Seth Fogarty for helping debug and compile a Haskell program for drawing railroad diagrams, and my many office mates

along the years, Daniel Chavarria-Miranda, Anirban Mandal, Arun Chauhan, Qing Yi, and Samah Mahmeed, for many helpful and enjoyable conversations. Other members of our group, Robert Fowler, Guohoa Jin, Apan Qasem, Cheryl McCosh, Jason Eckhart and Tim Harvey, have provided help at various times.

My parents have always encouraged me and guided me to independence. I am grateful to them for their generosity.

This work was supported in part by the Department of Energy's Office of Science (Cooperative Agreement Nos. DE-FC02-06ER25762, DE-FC02-07ER25800), the National Science Foundation (Grant No. ACI 0103759), and the Department of Energy's Los Alamos National Laboratory (Contract Nos. 03891-001-99-4G, 74837-001-03-49, and 86192-001-04-49).

Contents

List of Figures ix

List of Tables xiv

List of Algorithms xv

1 Introduction **1**

1.1 The Modeling Framework 4

1.2 Gaining Insight into Performance Bottlenecks 8

2 Background **10**

2.1 Performance Prediction Techniques 11

2.1.1 Profile-Based Approaches 11

2.1.2 Simulation-Based Approaches 14

2.1.3 Pencil-and-Paper Analytical Methods 16

2.2 Memory Performance Analysis 17

2.2.1 Understanding How Caches Work 18

2.2.2 General Techniques for Memory Performance Analysis 22

2.3 Techniques for Understanding Performance Bottlenecks 27

3 Static Analysis **31**

3.1 Static Analysis of Memory References 33

3.2 Associating Variable Names with Access Patterns 38

4 Dynamic Analysis **43**

4.1 Collecting Execution Frequency Histograms 44

4.2 Monitoring Memory Access Behavior 48

5 Building Scalable Models **53**

5.1 Models of Execution Frequency 54

5.2 Models of Memory Access Behavior 56

5.2.1 Modeling MRD Histograms 57

5.2.2 Considerations When Modeling MRD at Reference Level . . . 63

5.3 Evaluation of MRD Models 69

5.3.1 Predictions for Fully-Associative Caches 70

5.3.2 Predictions for Set-Associative Caches 72

5.4 MRD Modeling Results 75

5.4.1 Predictions for Itanium2 78

5.4.2 Predictions for MIPS R12000 81

6 Cross-Architecture Predictions of Execution Time **84**

6.1 Scheduler Front End 86

6.2 Machine Description Language 90

6.2.1 Execution Units 90

6.2.2 Instruction Execution Templates 91

6.2.3 Instruction Replacement Rules 93

6.2.4 Bypass Latencies 96

6.2.5 Memory Hierarchy Characteristics 97

6.3 Scheduler Implementation 98

6.3.1 Minimum Schedule Length Due to Resource Contention 100

6.3.2 Minimum Schedule Length Due to Recurrences 101

6.3.3 Preliminary Priority Metrics 102

6.3.4 A Hybrid Iterative-Recursive Scheduling Algorithm 108

6.3.5 Selecting the Next Node to Be Scheduled 110

6.3.6 Properties of Super-Structures 114

6.3.7 An Algorithm for Computing Super-Structures 118

6.4 Empirical Analysis of the Scheduling Cost 130

6.5 Cross-Architecture Prediction Results 134

6.5.1 Predictions for Itanium2 135

6.5.2 Predictions for MIPS R12000 139

7 Understanding Performance Bottlenecks 143

7.1 Understanding Instruction Schedule Inefficiencies 144

7.1.1 Assessing the Potential for Improving the Instruction Schedule 147

7.2 Understanding the Unfulfilled Memory Reuse Potential 149

7.2.1 Identifying Reuse Patterns With Poor Data Locality 150

7.2.2 Understanding Fragmentation in Cache Lines 155

7.3 Interpreting the Performance Data 159

8 Case Studies 164

8.1 Analysis and Tuning of Sweep3D 164

8.1.1 Identifying Performance Inefficiencies Due to Insufficient ILP . 166

8.1.2 Identifying Opportunities for Improving Data Reuse 171

8.2 Analysis and Tuning of POP 177

8.3 Analysis and Tuning of GTC 181

8.3.1 GTC Performance on the Itanium2 186

8.3.2 GTC Performance on the Opteron 189

8.3.3 Itanium2 vs. Opteron Running the GTC Code 196

9 Conclusions **198**

9.1 Contributions 199

9.2 Limitations and Open Issues 203

A Enhancements to the EEL Library **217**

B Routine Level Cache Miss Predictions on IA64 **221**

C Routine Level Cache Miss Predictions on R12K **239**

D Grammar of the Machine Description Language **256**

List of Figures

1.1 Modeling framework diagram. 5

2.1 A loop nest to multiply a pair of matrices. 21

3.1 Static analysis example 36

4.1 Step by step example for inserting counters into a CFG 47

4.2 Example of reuse distance histogram. All references with reuse distance less than the cache size are hits. 48

5.1 Modeling MRD histograms 59

5.1 Modeling MRD histograms (con't) 60

5.2 Another example of MRD modeling 64

5.3 Distribution of first accesses to cache lines for a matrix multiply code 65

5.4 Memory reuse distance collected for (a) $A[i,k]$ and (b) $A[i,k+1]$, from the matrix multiply code presented in Figure 3.1. 66

5.5 Aggregated memory reuse distance histograms of references to array A. 67

5.6 Distribution of sizes of reference groups under various aggregation scenarios 69

5.7 Fully-associative cache miss predictions example 71

5.8 Cache miss predictions for Itanium2 79

5.9 Cache miss predictions for MIPS R12K 82

6.1 Example of dependence graph and instruction replacement 88

6.2 MDL constructs for defining the execution units and restrictions between units. 91

6.3 MDL constructs for declaring instruction execution templates. 92

6.4 MDL constructs for declaring instruction replacement rules. 94

6.5 Example of MDL constructs for defining bypass latency rules. 96

6.6 Example of MDL constructs for describing the memory hierarchy levels. 97

6.7 Sample dependence graph with hierarchical *super-structures*. 105

6.8 Number of distinct recurrences in GTC with and without *super-structures* 106

6.9 Sample DAG of SCCs 111

6.10 Example of diamond-like structures. 115

6.11 Dominator tree of the dependence graph presented in Figure 6.7. . . . 119

6.12 Post-dominator tree of the dependence graph presented in Figure 6.7. 120

6.13 Hierarchy of super-structures for the dependence graph in Figure 6.7. 129

6.14 Final hierarchy of super-structures for the dependence graph in Figure 6.7. 130

6.15 Number of edges vs. the number of nodes 131

6.16 Number of recurrences vs. the number of nodes 131

6.17 Scatter plot of schedule cost vs. graph size 132

6.18 Itanium2 execution time predictions for 5 NAS benchmarks and Sweep3D 136

6.19 MIPS R12000 execution time predictions for 5 NAS benchmarks and Sweep3D 140

6.20 Sweep3D predicted execution time assuming different memory latencies 142

7.1 (a) Example of data reuse carried by an outer loop; (b) transformed example using loop interchange. 151

7.2 Cache line fragmentation example. 156

8.1 Main computation loops in Sweep3D. 165

8.2 Sweep3D: understanding potential for improvement from increased ILP. 166

8.3 Performance of the transformed Sweep3D code relative to the original version on an Itanium2 (lower is better). 169

8.4 Performance of the transformed code relative to the original version, for an Itanium2 like machine with issue width from 3 to 8. 170

8.5 Number of carried misses in Sweep3D 171

8.6 Accesses to `src` and `flux`. 173

8.7 Sweep3D: `jkm` iteration space 174

8.8 Diagram of Sweep3D with blocked `mi` 175

8.9 Performance of the original and improved Sweep3D codes on an Itanium2 system. 176

8.10 Data arrays contributing the largest number of fragmentation L3 misses. 182

8.11 Program scopes carrying the most L3 cache misses. 183

8.12 Program scopes carrying the most TLB misses. 186

8.13 GTC performance after each code transformation on an Itanium2 system. 187

8.14 Program scopes carrying the most cache misses in the modified code at 100 particles/cell. 188
8.15 Performance of the GTC code on an Opteron 275 190
8.16 Reuse pattern generating most L1TLB misses on Opteron. 193
8.17 GTC performance on an Opteron 275 machine after additional data transformations. 194

B.1 Routine-level memory predictions for sweep on IA64 221
B.2 Routine-level memory predictions for bt23 on IA64 222
B.2 Routine-level memory predictions for bt23 on IA64 (con't) 223
B.2 Routine-level memory predictions for bt23 on IA64 (con't) 224
B.3 Routine-level memory predictions for bt30 on IA64 225
B.3 Routine-level memory predictions for bt30 on IA64 (con't) 226
B.4 Routine-level memory predictions for lu23 on IA64 227
B.4 Routine-level memory predictions for lu23 on IA64 (con't) 228
B.5 Routine-level memory predictions for LU hp2d NPB 3.0 on IA64 . . . 229
B.5 Routine-level memory predictions for LU hp2d NPB 3.0 on IA64 (con't) 230
B.6 Routine-level memory predictions for LU hp3d NPB 3.0 on IA64 . . . 231
B.6 Routine-level memory predictions for LU hp3d NPB 3.0 on IA64 (con't) 232
B.7 Routine-level memory predictions for LU orig NPB 3.0 on IA64 . . . 233
B.7 Routine-level memory predictions for LU orig NPB 3.0 on IA64 (con't) 234
B.8 Routine-level memory predictions for sp23 on IA64 235
B.8 Routine-level memory predictions for sp23 on IA64 (con't) 236

B.9 Routine-level memory predictions for sp30 on IA64 237

B.9 Routine-level memory predictions for sp30 on IA64 (con't) 238

C.1 Routine-level memory predictions for sweep on R12K 239

C.2 Routine-level memory predictions for bt23 on R12K 240

C.2 Routine-level memory predictions for bt23 on R12K (con't) 241

C.3 Routine-level memory predictions for bt30 on R12K 242

C.3 Routine-level memory predictions for bt30 on R12K (con't) 243

C.4 Routine-level memory predictions for lu23 on R12K 244

C.4 Routine-level memory predictions for lu23 on R12K (con't) 245

C.5 Routine-level memory predictions for LU hp2d NPB 3.0 on R12K . . 246

C.5 Routine-level memory predictions for LU hp2d NPB 3.0 on R12K (con't) 247

C.6 Routine-level memory predictions for LU hp3d NPB 3.0 on R12K . . 248

C.6 Routine-level memory predictions for LU hp3d NPB 3.0 on R12K (con't) 249

C.7 Routine-level memory predictions for LU orig NPB 3.0 on R12K . . . 250

C.7 Routine-level memory predictions for LU orig NPB 3.0 on R12K (con't) 251

C.8 Routine-level memory predictions for sp23 on R12K 252

C.8 Routine-level memory predictions for sp23 on R12K (con't) 253

C.9 Routine-level memory predictions for sp30 on R12K 254

C.9 Routine-level memory predictions for sp30 on R12K (con't) 255

List of Tables

5.1 Memory hierarchy characteristics for the testbed machines. 75

6.1 Classification of SPARC native instructions into generic RISC classes. 89

7.1 Recommended transformations for improving memory reuse. 162

8.1 Breakdown of L2 misses in Sweep3D. 172

8.2 POP: top routines based on the improvement potential from additional ILP. 178

List of Algorithms

1 Main step of computing *super-structures*. 123

2 Initialize *domset* and *pdomset* for node v. 124

3 FindLCA(v,thisset,otherset,thisDS,otherDS,minLCA) 125

4 FindLCA(v,thisset,otherset,thisDS,otherDS,minLCA) (con't) 126

5 The refinement step of computing *super-structures*. 128

Chapter 1

Introduction

Over the past two decades, there has been a dramatic increase in the peak performance of microprocessors, due to both architectural innovations and improvements in semiconductor technology. At the same time, it has become increasingly difficult for applications to sustain a significant fraction of peak performance. This is due, in part, to an imbalance between the resources offered by super-scalar architectures and the actual needs of applications. The need to understand how to better tailor an application to a particular architecture, combined with a growing interest in reconfigurable architectures, heterogeneous processors [29], and grid computing [39], make the need for performance analysis and modeling tools more acute than ever.

Application performance modeling fills this need. Models of application performance have many uses, including understanding how performance scales under different conditions, guiding mapping of application components to a collection of heterogeneous resources, predicting how an application would perform on proposed future architectures, and providing insight into the design of custom architectures. In addition, accurate models of application performance can identify mismatches between application and architecture characteristics, pinpoint sources of inefficiency and iden-

tify opportunities for tuning. However, building accurate application performance models is difficult because a large number of variables affect execution behavior, such as application specific factors, architectural characteristics, and data input parameters. Moreover, these factors interact in complex ways, making it difficult to understand what limits performance at different points in a program.

Hardware counters, present on all modern microprocessors, provide a low overhead mechanism for observing resource utilization during an application's execution. However, while they can highlight possible causes of performance problems, such as identifying loops where high fractions of cache misses occur, this information is rarely sufficient to help one understand and correct the underlying problem. For example, performance measurements based on hardware counter sampling can identify loops that fail to sustain a satisfactory fraction of peak performance by monitoring the number of instructions retired per machine clock cycle. However, they fail to provide the insight necessary to distinguish between a problem due to a mismatch between the application's instruction mix and the machine's execution units, and one caused by the presence of recurrences, which limit the amount of instruction-level parallelism in tight loops.

When faced with a performance problem, we must first understand the causes for poor performance before we can design an appropriate solution. For example, if recurrences limit instruction-level parallelism in a frequently executed loop, we must transform the code to increase the amount of fine-grain parallelism. If, on the other hand, there is plenty of instruction-level parallelism but the instruction mix does not match the number and type of execution units present on the target architecture, we

can improve performance either by moving to a different more appropriate architecture, or by adjusting the architecture, which is possible when the architecture is a field-programmable gate array (FPGA).

To address the shortcomings of current techniques, this thesis presents an approach based on detailed modeling of the main factors affecting performance. Our goal is to complement the information provided by hardware performance counters with additional metrics that aid tuning by highlighting the factors that limit performance at different points in a program.

The thesis of this work is that it is possible to characterize applications in an automatic way (1) to predict their performance on different platforms with reasonable accuracy, and (2) to automatically identify several types of performance bottlenecks, provide insight into ways of addressing them and estimate the potential gains from doing so.

In this thesis we consider performance problems that arise from poor data locality, insufficient instruction-level parallelism, or a mismatch between the application's instruction mix and the resources offered by the target architecture. Our goal is to answer questions about an application's performance such as: Is a program slow due to a costly algorithm? Does the program have insufficient instruction-level parallelism, or is there a mismatch between the number and type of resources provided on the target machine, and the type of resources required by the most frequently executed loops of the application? Is the application bandwidth starved, or is it limited by the memory latency? How much bandwidth at each level of the memory hierarchy is needed to fully utilize the execution units if we could perfectly prefetch all data?

1.1 The Modeling Framework

We use static and dynamic analysis of binaries as the basis for performance modeling and prediction. Because we analyze and instrument object code, our tools are language independent and naturally handle applications with modules written in different languages. In addition, binary analysis works equally well on optimized executables; we do not need our own optimizer to predict performance of optimized code. Finally, object code is more concrete for performance evaluation; it is easier to predict performance of a mix of machine instructions with predictable latencies than to estimate the execution cost of high-level language constructs.

Working on binaries, however, does have drawbacks. First, a tool that models performance of application binaries is limited by the binary analysis library it uses. We mitigate this by measuring and modeling application characteristics that are machine independent, and then using those models to predict performance on arbitrary RISC-like architectures. Second, certain high-level information is lost when the source code is translated to low-level machine code, or may require more thorough analysis to extract.

To analyze and model application performance, we developed a toolkit that 1) analyzes a program's object code to collect measurements of the main application characteristics affecting performance, 2) builds scalable models of the measured dynamic application characteristics, and 3) uses these models to predict the performance of applications on different target architectures. The toolkit's binary analysis capabilities are based upon the Executable Editing Library (EEL) [32], which supports analysis

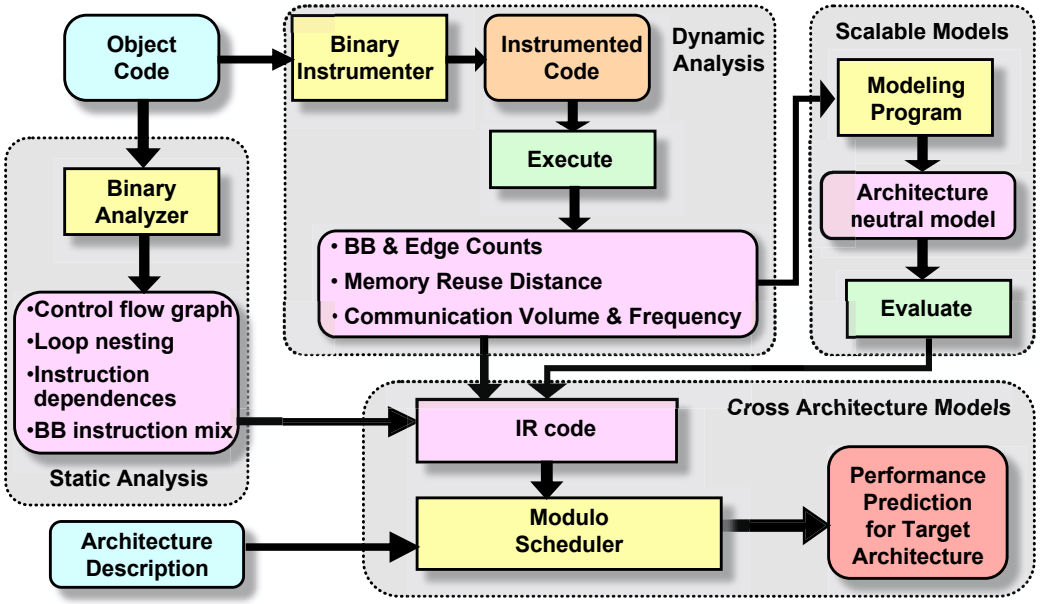


Figure 1.1: Modeling framework diagram.

and instrumentation of SPARC binaries. The focus on SPARC binaries is not a significant limitation since our toolkit uses binary analysis to construct architecture-neutral models for predicting application behavior and performance on arbitrary RISC-like architectures by combining these models with a description of the resources of a target machine. Because the EEL library has not been publicly maintained since 1995 and since both the Sun Forte compiler and the GNU GCC compiler have been updated continually since that time, we had to enhance EEL to understand and manipulate the control flow idioms used by today's compilers. Our enhancements to the EEL library are described in Appendix A.

Figure 1.1 shows an overview of our modeling framework adopted by our toolkit. There are four main functional components: static analysis, dynamic analysis, an architecture-neutral modeling tool, and a modulo-scheduler for cross-architecture per-

formance predictions and bottleneck identification.

The *static analysis* subsystem shown in Figure 1.1 is not a standalone application but rather a component of every program in our toolkit. We employ static analysis to recover high-level program information from application binaries, including reconstructing the control flow graph (CFG) for each routine, identifying natural loops and loop nesting in each CFG using interval analysis [63], and understanding the names of the program variables referenced by each memory instruction. Other uses of static analysis involve modeling aspects of an application that are important for its performance but are independent of its execution characteristics, such as, understanding the instruction mix in loop bodies, computing dependences between instructions that determine the instruction-level parallelism (ILP), identifying irregular and indirect memory access patterns, and finding opportunities for structure or array splitting. We tackle two particularly difficult problems, understanding what program variable names are associated with each memory instruction and computing memory dependences within loops from machine code, by computing symbolic formulas that describe the memory locations accessed by each memory reference. This process is presented in more detail in section 3.1.

Often, many important characteristics of an application's execution behavior can only be understood accurately by measuring them at run time using *dynamic analysis*. Our toolkit uses binary rewriting to augment an application to monitor and log information during execution. To understand the nature and volume of computation performed by an application for a particular program input, we collect histograms indicating the frequency with which particular control flow graph edges are traversed

at run time. To understand an application's memory access patterns, we collect histograms of reuse distance [7]—the number of unique memory locations accessed between a pair of accesses to a particular data item—observed by each load or store instruction. These measures quantify key characteristics of an application's behavior upon which performance depends. By design, these measures are independent of architectural details and can be used to predict the behavior of a program on an arbitrary RISC-like architecture.

Collecting dynamic data for large problem sizes can be expensive. To avoid this problem, we construct *scalable models* of dynamic application characteristics. These models, parameterized by problem size or other input parameters, enable us to predict application behavior and performance for data sizes that we have not measured [41, 42].

To obtain *cross-architecture performance predictions*, we combine information gathered from static analysis and dynamic measurements of execution behavior. We then map this information onto an architectural model constructed from a machine description file. This process has two steps. First, we predict the program's memory hierarchy behavior by translating memory reuse distance models into predictions of cache miss counts for each level of the memory hierarchy on the target architecture. We predict capacity and compulsory misses directly from reuse distance models and estimate conflict misses using a probabilistic strategy [42]. Second, we predict computation costs by identifying paths in each routine's CFG and their associated frequencies, and mapping instructions along these paths onto the target architecture's resources using a modulo instruction scheduler.

1.2 Gaining Insight into Performance Bottlenecks

We extended the modeling framework described in the previous section to identify performance bottlenecks and determine the potential for improvement from correcting them. We designed a modulo instruction scheduler that attributes each clock cycle of the execution cost to either application dependences or contention for machine resources. Based on this work we introduce new metrics that describe the potential for improvement from increasing the amount of instruction-level parallelism, or from additional machine resources.

To understand if an application is bandwidth starved, we present the minimum bandwidth requirements at each level of the memory hierarchy necessary to sustain the computation's speed if we could ideally prefetch all data, and we compute execution delays caused by insufficient bandwidth availability.

To understand the causes for poor data locality, we collect reuse information separately for each data reuse pattern. This approach enables us to understand not only where we experience a high number of cache misses, but also where data was last accessed before it was evicted from cache, and what program loop causes the data to be repeatedly accessed.

We present an algorithm to identify poor spatial reuse caused by layout of data in memory. We call the fraction of data in a memory block that is not accessed the *fragmentation factor*. We compute fragmentation factors for each reference and each loop nest in the program by analyzing the strides with which data arrays are traversed. We compute metrics that show how many cache misses at each memory

level can be eliminated by transformations such as array or structure splitting.

We present general guidelines for interpreting these performance metrics and we exemplify their use by describing the steps involved in tuning of three scientific applications.

The rest of the thesis is organized as follows. Chapter 2 presents background information and an overview of related work. Chapter 3 describes the application characteristics that can be understood through static analysis. Chapter 4 presents our binary instrumentation infrastructure and describes the application characteristics that we measure using dynamic analysis. Chapter 5 explains the process of building scalable architecture-neutral models of edge execution frequencies and memory reuse distance histograms. Chapter 6 presents a machine description language used for modeling a target architecture, and describes the implementation of a modulo instruction scheduler used for computing cross-architecture predictions of computation cost. Chapter 7 describes extensions to the modeling framework and introduces new performance metrics for gaining insight into performance bottlenecks. Chapter 8 describes the process of analyzing and tuning three scientific applications. Chapter 9 summarizes the contributions of this dissertation, discusses the limitations of this work, and outlines several open problems related to this work.

Chapter 2

Background

Performance analysis and modeling comprise the process of building mathematical constructs to describe performance characteristics of a computer system. Performance prediction entails estimating the execution time of a program on a given architecture. Predicting the execution time of an application under varying conditions is one of the most important, and yet most difficult, aims of performance analysis research.

Performance analysis research is important to several groups of people. Compiler writers, application developers and computer architects have the most interest in this research area. Application developers need performance analysis to understand the causes of inefficiency in their programs, to make procurement decisions or to guide the mapping of application components onto a set of heterogeneous resources. Compiler writers may use performance tools to understand opportunities for optimization, estimate potential payoff of optimizations on today's architectures or to evaluate the performance of the code they generate for the architectures of tomorrow that are in the design stage. Finally, computer architects may use performance models to understand the causes of performance inefficiencies, estimate their impact, and explore the impact of architectural changes.

Existing performance prediction methods range from back-of-the-envelope estimates to detailed analytical models or cycle-accurate simulations. Active areas of research in this field are on finding performance prediction approaches that reduce the time overhead required by full-scale simulators while maintaining a reasonable level of accuracy, and methods for attribution of execution inefficiencies to application features or to application/architecture interactions which provide insight into application tuning. Next section summarizes some of the recent related work in this field. Section 2.2 presents background information and a summary of the existing techniques for analyzing memory access behavior. Section 2.3 describes related work in the area of performance bottleneck identification and application tuning.

2.1 Performance Prediction Techniques

We can divide the existing performance prediction techniques into three main categories: profile-based approaches, simulation-based approaches and pencil-and-paper analytical methods.

2.1.1 Profile-Based Approaches

Profile-based performance prediction methods use hardware performance counters or code instrumentation to collect performance data during an application's execution. Afterwards, a post-processing tool analyzes this data to determine places where most time is wasted and, hence, are the most profitable to be optimized, or to compute an estimate for the application's execution time.

Hardware performance counters are a set of special registers available in most modern microprocessors. These registers count events that take place inside the processor or the memory subsystem when application software is executed. Hardware performance counters can capture statistics about executed instructions or memory hierarchy behavior with a minimal time overhead. Hardware performance counters are especially useful in gathering data about the interaction between the application and the hardware it is running on. The method presented in this thesis constructs models for characteristics of an application that are independent of the hardware; therefore, hardware counters are not used by our toolkit.

Alternatively, code instrumentation can be used to collect information about an application's execution behavior. Code instrumentation can be performed either on source code or on object code. Object code instrumentation can be classified further as dynamic instrumentation, link-time instrumentation, or static instrumentation. In recent years, several tools/libraries for binary instrumentation have emerged. DynInst [11] is a portable application program interface (API) that enables development of tools and applications that require run-time code patching. Dynamo [5] is a run-time dynamic optimization system that focuses on optimization opportunities which manifest themselves only at runtime and, hence, are difficult for a static compiler to exploit. Purify [22] is a well known tool for memory error and leak detection that uses object code instrumentation at link-time to monitor memory allocation, deallocation and load/store operations performed by an application. The category of static binary instrumentation is represented by EEL [32], ATOM [61], Etch [58] and Pin [37] libraries that enable tools and applications to analyze and modify binary

programs without being concerned with low-level architectural details.

Snavely, Wolter and Carrington [59] start from the premise that a parallel application's performance is based on two major factors: its single node performance and its use of a communication network. They consider that the network interconnect contribution can be estimated by a network simulator. For single processor performance, their investigations focus on memory-bound codes, such as the NAS Parallel Benchmarks [4] (NPB) kernels. Therefore, they use the "rule of thumb" that the per-processor performance of an application is predominantly a function of how it exercises the memory subsystem. Their performance prediction method consists of collecting Machine Signatures – characterizations of the rates at which a machine can execute fundamental operations independent of the particular application – and Application Profiles – summaries of the fundamental operations to be carried out by the application in abstraction of any particular machine. Next, they use a convolution method to combine application profiles with machine signatures to determine the performance of an application on a particular architecture. For the NPB kernels they considered, the errors of the estimates are less than 20% in all cases. The modeling costs are a sixty-fold slowdown for collecting the Application Profiles and a roughly equal amount of time spent in gathering the Machine Signatures. However, the latter information must be collected once for each machine and can be used for multiple applications without any further overhead. This time overhead is favorable when compared to a cycle-accurate simulation, which can dilate execution time by as much as a factor of four to size orders of magnitude without a significant gain in accuracy for these memory-bound kernels.

However, considering the requirements of executing profiled applications at full scale to characterize node performance, and simulating communication traces at full scale to estimate the synchronization cost, we consider that even a sixty-fold slowdown is an important deterrent for using this method with large data sizes on which the original program might run for days. The memory-bound rule can be safely applied to many scientific applications, but this should not be a general assumption. Other factors, such as the instruction schedule dependences, can affect the estimates more than just a few percentage points. Snavely et al's work is the closest in concept to the method presented in this thesis.

Like Snavely et al, we focus on collecting application characteristics independently of the target machine. However, we focus on single node performance. Rather than considering only models for memory-bound applications, our models consider a variety of factors that can affect performance, including memory hierarchy latency, the instruction schedule dependences and instruction mix inside frequently executed loops. More importantly, we build scalable models of these application characteristics. As a result, we can predict the performance of an application for larger problem sizes than ones that are practical for monitoring and simulating at scale.

2.1.2 Simulation-Based Approaches

Performance prediction methods based on simulation consist of executing an application in conjunction with a program that emulates the target architecture. Since each dynamic instruction must be simulated, these methods have a significant time overhead. On the other hand, a detailed simulation can produce a very accurate pre-

diction of the application's performance. Trace-based simulators are a common style of tools supporting this type of method. To use a trace-based simulator, the application is instrumented using a profile-based tool to collect a trace of important events that occur during its execution. The resulting trace file is then fed into a program which simulates only a fraction of the dynamic instructions originally executed by the application. Hence, this method is much faster than a cycle-accurate simulation. However, speed comes at a cost and the trade-off in this case is the accuracy of the prediction and the level of detail about the phenomena taking place inside the simulated machine. A general drawback of simulators is that the user has to simulate the application under study for each data set of interest. Still, simulation or trace-based simulation can be used together with other techniques to construct scalable models of an application's performance.

The POEMS [1] project by Adve et al., is an environment for end-to-end performance modeling of complex parallel and distributed systems. The POEMS modeling effort spans the domains of application software, runtime libraries, operating system, and hardware architecture. The POEMS effort aims to assemble component models from these different domains into an end-to-end system model. The composition process is specified using a generalized graph model of a parallel system, combined with interface specifications describing the component behaviors and evaluation methods. The POEMS framework is a complex system that aims at building complete end-to-end models using a large range of modeling paradigms including analysis, simulation and direct measurement. The drawbacks of this approach are the high complexity of the system with possible long execution times for the simulators and the system's

dependency on the task graph that can be generated only by the Rice *dHPF* compiler [46]. This thesis explores another approach for building scalable models of an application's computation using dynamic analysis of application binaries without any special compiler support.

2.1.3 Pencil-and-Paper Analytical Methods

The pencil-and-paper method is the traditional technique for building performance models. It requires a deep understanding of the algorithms that are used in an application, details about their implementation and a good knowledge of the dynamic structure of an execution imposed by synchronization in the application. As a result, this method is not widely used in industry; it is used only by highly skilled researchers. However, it will continue to be used until more automated methods become mature enough to achieve the desired levels of accuracy. On a smaller scale, pencil-and-paper models can also be used to validate models constructed with other techniques.

Sundaram-Stukel and Vernon [62] analyze and construct a LogGP performance model for Sweep3D, a wavefront application with a complex synchronization structure. They focus on building accurate models of MPI communication primitives using micro-benchmarks on two or four nodes. They show that the LogGP model predicts with high accuracy the measured application execution time for large problem sizes running on 128 nodes or more. Stukel and Vernon found poor scaling beyond two thousand nodes for the Sweep3D application using the current algorithm, due to synchronization overhead.

Hoisie et al. [24] studied the performance of wavefront algorithms implemented

using message passing on two-dimensional logical processor arrays. Wavefront algorithms are widely used in parallel computing since they enable parallelism in computations that contain recurrences. Hoisie's group is focused on predicting performance of wavefront applications on cost-effective machines that have non-uniform network topologies, such as in a cluster of SMPs. In their work on wavefront applications, their models successfully captured the communication patterns of these applications. Their results show accurate performance predictions validated on a cluster of Origin 2000 machines with up to 1024 processors.

While the pencil-and-paper approach has produced some of the most accurate models to date, their obvious disadvantage is the human effort required to construct them.

2.2 Memory Performance Analysis

With the large gap between the peak performance of microprocessors and their memory systems, it is unanimously accepted that memory hierarchy response is the factor most limiting node performance for data intensive applications. Cache memories are small, fast buffers placed between the processor and the main memory to help hide the large latency of memory accesses. Caches are effective only if an application exhibits temporal or spatial data reuse[1]. For an application with a working set much larger than the cache size, a stream of random accesses to the working set will exhibit little or no data reuse. To improve the reuse of data in cache, one must optimize applications to exploit temporal reuse of data that is already in the cache before

[1] We define these terms on page 19.

evicting it, and to improve spatial reuse by placing data that is accessed together into the same cache block or into non-conflicting blocks. This section describes the main techniques for assessing how an application uses the memory hierarchy. However, first we review how caches work.

2.2.1 Understanding How Caches Work

Caches are characterized by three principal parameters: cache size, block size, and associativity [53]. The cache size is the total capacity of the cache in bytes. The cache is divided into a number of equally sized blocks called cache lines. All operations with the next level of the memory hierarchy are performed with a cache line granularity. The size of the cache line defines the number of bytes that are fetched from memory when a cache miss occurs, or the number of bytes written to memory when a modified cache line has to be evicted. The capacity of the cache (C) is equal to the number of blocks (N) times the block size (B).

Associativity represents the number of unique cache lines in which a memory block can reside. If a memory block can be loaded into one cache line only, the cache has a direct mapping. If the block can be loaded into any line of a cache, the cache is said to be fully associative. In practice, caches are often positioned somewhere between these two extremes. If a memory block can reside in any of a set of exactly k cache lines, then the cache is said to be k-way set-associative.

The level of associativity built into a cache memory affects the performance of the entire system. Fully associative caches are expensive and difficult to build; therefore, caches used in practice have a much lower level of associativity. To understand how

the associativity level can affect the performance, one must consider how memory blocks are mapped to cache lines depending on the type of cache. Let R be the ratio between the size of the memory and the capacity of the cache. Because the capacity of the cache is much smaller than the size of the memory, on average R memory blocks compete for each one of the N cache blocks. In a fully associative cache, any of the $R*N$ memory blocks can occupy any of the N cache blocks. In a k-way set-associative cache, $R * k$ memory blocks compete for a set of k blocks in the cache. Even if the ratios are equal, there is a difference in the size of the two sets. For a fully associative cache, the layout of data in memory is important only for clustering together data that has temporal affinity. For a system with a lower level of associativity, data layout algorithms must consider also conflicts between different memory blocks. Memory blocks that map to the same cache set, are said to conflict with each other. This is a problem only when more than k memory blocks of the working set map to a single set of k cache blocks, while other cache sets are underutilized. Therefore, more optimizations are necessary when compiling an application for a machine with a low level of cache associativity, to ensure that the most frequently accessed data structures are uniformly distributed over the N/k sets.

Whenever we access data that is already loaded into cache, we say we experience a cache hit. Cache hits are classified as temporal reuse or spatial reuse, depending on what memory location was previously accessed that caused the memory block to be loaded into cache. We say that a memory access has temporal reuse if a previous access to the same location fetched the memory block into the cache or kept it from being replaced. A memory access has spatial reuse in the cache if a previous access to

a different location from the same memory block caused the block to be loaded into the cache. A cache configuration with a cache line of size one has no spatial reuse.

Cache misses can be classified in three categories: compulsory misses, capacity misses and conflict misses. Compulsory misses, also known as cold misses, are produced when memory blocks are accessed for the very first time. If an application has to access each memory block only once during the entire execution, then all memory accesses will result in compulsory misses. Compulsory misses can be hidden by explicitly prefetching the data. Another (theoretical) solution is to reduce the number of blocks in memory by increasing the block size.

Capacity and conflict misses are said to be non-compulsory cache misses, because they occur on accesses to blocks that have been previously accessed but are no longer in cache. An access to a memory block b_i is a capacity miss if and only if at least N different other blocks were referenced since the last access to b_i. This is equivalent to saying that capacity misses are misses that would occur in a fully associative cache with an LRU replacement policy and that are not compulsory misses. The number of capacity misses can be reduced by either increasing the number of blocks in the cache or by restructuring the application to shorten the distance between accesses to same block if there is potential temporal reuse that is not being exploited.

A reference that hits in a fully associative cache and misses in a k-way set-associative one is called a conflict miss. It means the referenced block b_i was accessed in the recent past because the reference is a hit in a fully associative cache, but at least k other different memory blocks from the set of $R * k$ blocks that compete for same set of cache lines were also referenced since then, causing the eviction of block

b_i. Conflict misses are the most difficult to model or to predict since they are the result of a complex interaction between the characteristics of the cache sub-system, the layout of data in memory and the access pattern used by applications.

It is possible for an application to have a negative number of conflict misses. This means the program can exhibit more misses with a fully associative cache than with a set-associative cache. This behavior can be illustrated with an unoptimized matrix-multiply program. Figure 2.1 presents the "C" code for a simple matrix-multiply program. Arrays X, Y and Z are laid out in memory in row-major order - consecutive elements of a row occupy consecutive locations in memory. Let us consider a case where the matrix line size expressed in bytes, M, is less than the size of the cache, C, but the number of elements in a line or column, n, is greater than the number of blocks in the cache. We would expect to see only spatial reuse for consecutive stride one accesses to X performed in the inner loop and no temporal reuse from the middle loop using a fully associative cache.

```
for (i=0 ; i<n ; ++i)
   for (j=0 ; j<n ; ++j)
   {
      Z[i][j] = 0;
      for (k=0 ; k<n ; ++k)
         Z[i][j] += X[i][k]*Y[k][j];
   }
```

Figure 2.1: A loop nest to multiply a pair of matrices.

The program references a full row of matrix X and a full column of matrix Y before reusing the same row of X in the next iteration of the middle loop. We can compute how many blocks are accessed by the inner loop. A line of matrix X occupies M/B

blocks, where B is the block size in bytes. An entire column of Y occupies n blocks, where n is the number of elements in a column, because two consecutive elements of a column occupy two different memory blocks. Since n is greater than the number of blocks in the cache, referencing an entire column of Y in the inner loop causes eviction of the blocks allocated for X and Y in the first iterations of the loop. As a result, the code would experience no temporal reuse for accesses to X on consecutive iterations of the middle loop.

For a 2-way set-associative cache, accesses to array Y will generate a lot of conflict misses, many times causing the eviction of other elements of Y referenced not long before. This effect will cause some of the elements of X to remain in cache for the entire execution of the inner loop, and the application will experience some level of temporal reuse in the next iteration of the middle loop. Accesses to the Y array produce conflict misses but because they are already counted as capacity misses and X experiences more hits than in the case of a fully associative cache, by the definition above, accesses to X have a negative number of conflict misses.

2.2.2 General Techniques for Memory Performance Analysis

Existing techniques for understanding how an application uses the memory hierarchy include cache simulator methods, compile time analysis techniques, profile-based methods and dynamic code monitoring and optimization.

Cycle accurate simulators were described in section 2.1.2. They emulate the entire target machine, and therefore, they must simulate the cache in the process. Cache simulators do not need to simulate execution of all of an application's instructions,

only the memory references. The simulation can be performed either offline, on a trace of references collected with the help of a profiling tool, or on-the-fly by executing an instrumented version of the program. Except for compile time analysis methods, all the other methods make use of a cache simulator. The disadvantages of a trace-based cache simulator are the possibly large disk space needed to store the traces with tens or hundreds of millions of memory references, and the need to separately collect and simulate a trace of memory references for each configuration of input parameters and cache parameters that must be analyzed.

Recent work by Mueller et al. [50] proposes the use of regular section descriptors (RSD) and partial address traces to represent the data traces in constant space, solving the first disadvantage presented above. Mueller et al. use dynamic binary rewriting to collect the partial data traces. However, the presented algorithm deals only with regular access patterns of streams that have a constant stride. Extending the algorithm to handle arbitrary strides might be possible, but with a big cost increase.

This thesis presents another method for analyzing the data access pattern of an application by characterizing the memory reuse distance for each memory reference. Memory reuse distance is a measure of the number of distinct memory locations that are accesses between two references at the same datum. Comparing the reuse distance information seen by a memory reference and the number of blocks present in a cache sub-system (see Section 2.2.1) provides a direct indication if the memory access is a hit or not in the considered cache configuration. Although we collect detailed enough information to predict the miss ratio independently for each memory reference in

the program, the output is rather compact and requires an almost constant space as problem size increases. Also, we can deal with strides that are arbitrary polynomial functions, not only constants. Furthermore, this work presents a method to predict how the memory reuse data extrapolates at a different problem size that we did not collect data for.

The MHSim [47] memory simulator by Mellor-Crummey, Fowler and Whalley, uses source-to-source translation of Fortran programs to instrument all memory accesses with code that simulates a parameterized cache system on-the-fly. Next, the MHSim simulator processes the annotated data trace and correlates cache miss information to line numbers in the source code. However, since MHSim uses source-level instrumentation, the collected trace of memory references may not correspond to the actual order in which accesses are executed by an aggressively optimized binary. For instance, source-level instrumentation might prevent loop transformations such as tiling, that change the order in which data is accessed.

Lebeck and Wood [33] implemented CProf, another cache profiler. They use static instrumentation of application binaries to substitute the memory references with calls to a function that simulates caches online. We use a similar mechanism for static binary instrumentation, but instead process the stream of accesses online to create a compact representation of the data access pattern exhibited by the application instead of simulating the cache on-the-fly.

Hill and Smith [23] describe forest simulation, that takes advantage of set refinement and inclusion to simulate alternative direct-mapped caches that have these properties. In addition, they describe a probabilistic model for estimating set-associative

miss ratios from fully-associative miss ratios. We use this probabilistic model to predict cache miss counts for set-associative caches from distributions of memory reuse distance.

Over the years, memory reuse distance has been studied by many researchers investigating memory hierarchy management techniques [7, 44] or trying to understand data locality in program executions for individual program inputs [8, 17]. Recently, two other research groups have explored using memory reuse distance data from a few training runs to compute cache miss rate predictions for other program inputs. Zhong et al. [65] describe using two memory reuse distance histograms that are an aggregation of all accesses executed by a program as the basis for modeling. Fang et al. [20] use a similar modeling strategy but they collect data and build models on a per-instruction basis.

Our work differs from that of Zhong et al. and Fang et al. in six important ways. First, we characterize memory access patterns at the level of references groups determined through static analysis while the other two groups, respectively, build their models for the entire program or at the level of single instructions. Although we have never directly compared our models against those produced by either of the other two approaches, we have experimented with different levels of aggregation using our implementation. In those experiments, we found that building models from histograms constructed at the program or routine level for non-trivial programs results in significant errors with our automated method. Similarly, our first implementation of fine-grain modeling (at the instruction-level) performed no aggregation and its accuracy was hurt by complex interactions between multi-word memory blocks and

loop unrolling [40, pages 46–53]. Second, our modeling tool adaptively determines an appropriate partitioning of reuse distance histograms into bins while the other two groups use a fixed strategy based either on a constant number of bins (e.g. 1000) for every histogram, or on a logarithmic distribution of distances into bins. Third, we discover the appropriate modeling polynomials for each bin automatically and our models are linear combinations of a set of basis functions with a dynamically determined number of terms in each model. Zhong et al. and Fang et al. use combinations of only two terms where one is selected from a small set of pre-determined functions and the other is a free term. Fourth, we predict the actual number of cache misses for different input sizes rather than just a miss rate. Fifth, we predict cache miss counts for both fully-associative and set-associative caches. Finally, our models can be used to directly predict memory hierarchy responses for problem sizes not measured; the other aforementioned techniques require partial execution of using the problem size for which a prediction is desired to experimentally determine data sizes.

The category of compile time techniques is represented by the work of Ghosh, Martonosi and Malik [21]. They describe methods for generating and solving cache miss equations to get a detailed representation of cache misses in loop-oriented scientific codes. Ghosh et al.'s approach uses static analysis of the source code to generate a set of linear Diophantine equations whose solutions correspond to potential cache misses. This approach is suitable for compilers to guide memory optimizations.

2.3 Techniques for Understanding Performance Bottlenecks

Hardware counters provide a low overhead mechanism for monitoring the interactions between an application and its execution platform. A variety of tools use hardware performance counters to characterize the dynamic behavior of applications, e.g. VTune [27], HPCToolkit [48] and OProfile [51]. While these tools correlate resource utilization and architectural events with source programs, they don't provide direct insight into the factors that cause those events or into how performance would change if the machine characteristics were adjusted.

Many research groups use simulation or instruction schedulers to estimate application performance on a target architecture. Lam and Wilson [31] use trace simulation to understand the effects of control flow dependences on instruction-level parallelism. They simulate several architectures by relaxing different scheduling constraints and they evaluate the amount of parallelism exposed by each of those techniques. We use a similar technique to understand the performance improvement potential from increased instruction-level parallelism or from additional machine resources.

Chilimbi et al. [14] profile applications to monitor access frequency to structure fields. They classify fields as hot and cold based on their access frequencies. Small structures are split into hot and cold portions. For large structures they apply field reordering so that fields with temporal affinity are located in same cache block. Zhong et al. [66] describe k-distance analysis to understand data affinity and identify opportunities for array grouping and structure splitting. We use static analysis to understand fragmentation of data in cache lines, and find opportunities for structure

or array splitting.

Mellor-Crummey et al. [49] study the impact of data and computation reordering on data reuse and performance of irregular applications. They show that a coordinated approach involving both data and computation reordering exceeds approaches that perform either data reordering or computation reordering alone. They conclude that data reordering based on position along a space-filling curve is fast, and it probabilistically increases spatial locality. In this work, we classify data reuse patterns as strided or irregular, and we compute metrics that can identify opportunities for applying data or computation reordering strategies.

Beyls and D'Hollander [9] describe RDVIS, a tool for visualizing reuse distance information clustered based on the intermediary executed code (IEC) between two accesses to the same data, and SLO, a tool that suggests locality optimizations based on the analysis of the IEC. The capabilities of their tools are similar to those we describe in this paper. At the same time, our implementations differ significantly in the ways we collect, analyze and visualize the data. In addition to the histograms of reuse distances, Beyls and D'Hollander collect the sets of basic blocks executed between each pair of accesses to the same data. Afterwards, an offline tool clusters the different reuse patterns based on the similarity of the IEC. A second tool analyzes the IEC to determine the carrying scope of each reuse. In contrast, we directly determine the scopes where the two ends of a reuse arc reside, as well as the carrying scope based on a dynamic stack of program scopes. We cluster the reuse patterns based on their source, destination and carrying scope directly at run-time, which reduces the amount of collected data. Moreover, this approach enables us to leverage the modeling work

described in [42] to predict the scaling of reuse patterns for larger program inputs. Finally, our implementations differ also in the way data is visualized. RDVIS displays the significant reuse patterns as arrows drawn over the intermediary executed code between data reuses. In contrast, we focused on computing metrics that enable us to find the significant reuse patterns using a top-down analysis of the code, which we think it is more scalable to analyzing large codes where reuse patterns span multiple files. In addition, we identify reuse patterns due to indirect or irregular memory accesses, and inefficiencies due to fragmentation of data in cache lines, which enables us to pinpoint additional opportunities for improvement.

Callahan et al. [12] define two metrics, machine balance and loop balance. Machine balance characterizes the ratio between the peak memory and floating point performance. Loop balance represents the ratio between the number of words accessed and the number of floating point operations executed in one iteration of the loop. The authors compare the two metrics to determine if a loop is memory-bound or compute-bound. We compute similar metrics to identify execution delays due to insufficient memory bandwidth, except we consider the balance between memory bandwidth and the actual instruction schedule cost of a loop, not just its floating-point peak performance. We compare this value directly with either the peak or the sustainable bandwidth of a machine to determine if bandwidth is a limiting factor.

Trimaran [13] provides an infrastructure for investigating the interplay between architecture parameters, compiler technology and applications. It evaluates overall application execution time using simulation. SLOPE [18] provides compiler based sensitivity analysis and performance prediction. It classifies memory references as

strided or random and uses a simple list scheduler to compute instruction schedules for basic blocks, as the basis for static performance predictions. MonteSim [60], is a Monte Carlo simulator for predicting application performance on in-order microarchitectures. The simulation predicts the rate at which an application's instructions execute on a modeled architecture and how much time it will spend stalled. In contrast, our work uses both static and dynamic analysis to provide application-centric performance feedback useful for tuning in addition to computing predictions of memory hierarchy and execution behavior.

Chapter 3

Static Analysis

The static analysis subsystem shown in Figure 1.1 is not a standalone application but rather a component of every program in our toolkit. We employ static analysis to recover high-level program information from application binaries, including reconstruction of the control flow graph (CFG) for each routine, identifying natural loops and loop nesting in each CFG using interval analysis [63], or understanding the names of the program variables referenced by each memory instruction. Other uses of static analysis involve understanding and modeling aspects of an application that are important for its performance but are independent of its execution characteristics, *e.g.*, understanding the instruction mix in loop bodies, computing dependences between instructions that determine the instruction-level parallelism (ILP), identifying irregular and indirect memory access patterns, and finding opportunities for structure or array splitting.

There are two types of schedule dependences between instructions: register dependences (one instruction uses a register whose value is computed by a previous instruction) and memory dependences (two memory instructions access the same location and at least one of them is a write). Register dependences are easily identified.

We perform register renaming to ensure that each register definition (DEF) has a unique name and to remove false output- or anti-dependences. Following renaming, all register dependences between instructions are readily apparent as a DEF and subsequent USEs of the same register. Determining memory dependences accurately from machine code is a difficult problem. We tackle it by constructing symbolic formulas that describe the locations accessed by each reference. We use these symbolic formulas to understand which references access the same memory locations, and to compute the distances of the dependences [2].

On modern microprocessors, the location referenced by a memory instruction is first computed into a register by a series of integer arithmetic instructions. Disentangling these computations can be difficult because optimizing compilers often store in registers not only the starting address of a data structure, but addresses of particular fields as well. Also, multiple base registers used in a basic block may contain related addresses. In the next section, we describe in more detail the process of computing symbolic formulas that describe the memory locations accessed by memory references. The statically derived symbolic formulas are used to extract high level information from executables, including: a) understanding if two references are related and their reuse distance data must be modeled together (see Section 5.2.2), b) computing memory dependences as part of computing cross-architecture instruction schedule costs (see Section 6.1), or c) understanding poor spatial locality due to the layout of data in memory (see Section: 7.2.2). In Section 3.2, we describe yet another use of the symbolic formulas to compute a mapping between memory instructions and the names of the variables they access.

3.1 Static Analysis of Memory References

For each memory reference, we statically derive several symbolic formulas that describe the pattern of locations it accesses during execution. We perform this analysis intraprocedurally. We compute two types of formulas. For each reference in the program, we compute a formula for the *first location* it accesses. For references inside loops, we also compute formulas that describe how the accessed location changes from one iteration to the next. We describe first the process of computing the static symbolic formulas. At the end of this section we present a matrix-multiply example and the symbolic formulas derived for the references in the innermost loop.

We begin by constructing a *first accessed location* formula for each reference in a routine. For each register used in a reference's address computation, we recursively traverse the CFG backwards along USE-DEF chains until either (a) we encounter a load immediate value, (b) we cannot trace any further inside this routine (the traced register is the result of a function call or we reach the top of the routine), or (c) we determine that a formula for the register was already computed while analyzing a previous instruction. During this backward traversal of the CFG, we consider only forward CFG edges. As we return from recursion and unwind each step of our traversal, we compute a symbolic formula at each machine instruction along the traversed path by applying the instruction's operator to the formulas computed for its source registers. We cache every intermediate result so that we don't have to traverse the same chain of instructions a second time when analyzing another instruction.

During a USE-DEF chain traversal, if we find that a register is reloaded from the

register spill area, we trace backward along CFG edges for a corresponding register spill to the same location and then resume our USE-DEF chain traversal from the spill instruction. We generalized our mechanism for handling reloaded spill values to work with arbitrary loads after we encountered a situation in one executable in which the compiler had saved the value of a register in the data segment and later loaded it to compute the address of a memory reference. If the value of a register is defined by a load instruction, we trace backward along CFG edges for a store with a symbolic formula equal to the load's symbolic formula[1], and we continue tracing back from the register whose value was stored.

Formulas are restricted to sums of general terms including immediate values, loads from a constant address, loads from a constant offset from the stack frame pointer (an argument passed by reference or a local variable), loads from undetermined locations, and registers whose formulas cannot be written as a sum of these terms (e.g. a product of two non-constant formulas, etc.). Each term of a formula can have integer numerator and denominator coefficients. With these restrictions, any non additive (not an add or a sub) operation of two non-constant formulas will produce a register term. When at least one operand is a constant, several operations can be computed without simplification to a register value: multiply, divide, and left/right shift by a constant value can be performed by updating the coefficients of each term in the non-constant operand. If both operands are constants, all arithmetic and bitwise operators are computed precisely.

[1]Symbolic formulas for the instructions upstream of the one currently analyzed have been already computed.

For memory references inside loops, we compute additional formulas that indicate how the accessed location differs from one iteration to the next. We compute a *stride formula* for each loop surrounding the reference. For each level of a loop nest, we apply a recursive algorithm that traverses backward along CFG edges, similar to our method for computing *first accessed location* formulas. However, when computing stride formulas for a loop, we consider only forward CFG edges that are part of the loop plus the loop's back edge. This recursive search terminates when either (a) we encounter a load immediate operation, (b) we traced backwards through all of the instruction in the loop without finding a definition for this register (this register contains an invariant value with respect to this loop), or (c) we reach a definition for a second time. In this last case, the found instruction is part of a chain of instructions that update an index variable. As the recursion unwinds, the *stride formulas* are computed by applying the mathematical operators corresponding to each intermediate machine instruction to the non-invariant components of the source formulas.

The *stride formulas* are restricted to the same sums of general terms as the *first accessed locations* formulas. However, a *stride formula* has two additional flags that can be set by the recursive algorithm. The first flag indicates if an access has an irregular stride; this flag is set if the reference's stride is not constant for all iterations of that particular loop. The second flag indicates if an access is indirect and it is set if the formula for the accessed location depends on the value of another load which has a non-zero stride with respect to this same loop.

Figure 3.1(a) presents the source code for a naive implementation of matrix mul-

```
/* multiply two squared matrices */
void matrix_multiply(int N, double *A,
       double *B, double *C)
{
   int i, j, k;

   for ( i=0 ; i<N ; i+=1 )
     for ( j=0 ; j<N ; j+=1 ) {
       C[i*N+j] = 0;
       for ( k=0 ; k<N ; k+=2 )
         C[i*N+j] += A[i*N+k]*B[k*N+j]
            + A[i*N+k+1]*B[(k+1)*N+j];
     }
}
```

(a) matrix multiply source code

			first-location	k-stride	j-stride	i-stride
	L1: add	%l5,0x2,%l5				
A[i,k]	ldd	[%l3],%f4	%i1+16	16	0	8*%i0
	add	%l3,0x10,%l3				
	cmp	%l5,%o5				
	add	%l4,0x10,%l4				
B[k,j]	ldd	[%o3],%f2	16*%i0+%i2	16*%i0	8	0
A[i,k+1]	ldd	[%l4-0x10],%f8	%i1+24	16	0	8*%i0
B[k+1,j]	ldd	[%o7],%f6	24*%i0+%i2	16*%i0	8	0
	fmuld	%f4,%f2,%f10				
	fmuld	%f8,%f6,%f12				
	faddd	%f10,%f12,%f14				
	faddd	%f0,%f14,%f0				
C[i,j]	std	%f0,[%l2]	%i3	0	8	8*%i0
	add	%o3,%l1,%o3				
	bl,pt	%icc,L1				
	add	%o7,%l1,%o7				

(b) assembly for the innermost loop and the derived symbolic formulas

Figure 3.1: Static analysis example. The left subfigure presents the source code for a naive matrix multiply implementation, and on the right we have the SPARC assembly code for the innermost loop annotated with the symbolic formulas computed for each memory reference.

tiply. Because the program is written in C, the three matrices have been allocated as one-dimensional arrays such that the rows of each matrix are contiguous in memory. The three matrices are dynamically allocated and their size is passed as an argument to the *matrix_multiply* function, to exemplify how formulas are computed in the presence of symbolic values.

Figure 3.1(b) presents the SPARC assembly code for the innermost loop of the *matrix_multiply* algorithm. The five memory references have been annotated with the corresponding symbolic formulas we derived through static analysis. Each reference has a formula for the *first accessed location*, and three *stride formulas*, one for each level of the loop nest. Each reference is also annotated with the corresponding source code array access to make the code easier to understand. The k-loop was unrolled by hand once[2] and we compiled the binary with unrolling disabled to keep

[2]Additional code not included in the figure handles the remaining iteration of the k-loop when

the size of the assembly code small for the purpose of this example. The reason for unrolling the loop once is to show how this type of static analysis can uncover related references. Those familiar with the SPARC assembly language will recall that registers *%i0, ..., %i5* are used for passing the first six input arguments of a routine, and will notice that all symbolic formulas are correctly computed relative to the source code on the left. The compiler has peeled one iteration of the k-loop, therefore the *first location* formulas correspond to $i = 0, j = 0, k = 2$.

To exemplify, we are going to analyze in more detail the formulas computed for reference `A[i,k]`. The *first location formula* for `A[i,k]` is `%i1+16`. Register `%i1` holds the value of the second formal parameter according to the SPARC ABI. Thus, the value of this *first location* formula is equal to the starting address of `A` plus 16 bytes which corresponds to `A[0,2]`. The *k-stride* formula is 16 because the k-loop is unrolled once and we access array `A` with unit stride. The computed value of the *j-stride* formula is zero for this reference because `A[i,k]` does not depend on loop index `j`. Finally, the value of the *i-stride* formula is `8*%i0`. Remember that register `%i0` holds the value of the first formal parameter, which is the matrix size in our case. Because each double precision element occupies eight bytes, the stride with respect to the i-loop is $8 * N$. The correctness of the formulas computed for the other references can be similarly determined. Although each reference in the assembly code uses distinct address registers, the symbolic formulas we recover for references to the same array show they are related.

N is odd.

3.2 Associating Variable Names with Access Patterns

When analyzing a program's memory access behavior, often it is useful to understand not only which program scopes contain a high fraction of data accesses that miss in the various cache levels, but also which data objects have poor locality of reference within each scope. While data names are easily observable in the program source code, there is no direct way to map a memory instruction from an executable to a specific data object name even when the object code contains debugging information.

To support debugging, compilers include some information about data objects in an executable's symbol table and debug section. This information is necessary for debugging tools to recover the value associated with a program variable name whenever the program execution is interrupted during a debugging session. When a program is compiled with debugging information, compilers include entries in the symbol table about the type, size and starting location in the data section, for each global variable. Local variables and formal parameters of each routine are described separately into the stabs or the dwarf section[3], the location information being represented by an offset from the frame pointer. In addition, formal parameters are listed in program order, thus we can recover the index into the formal parameter list of each argument. For debuggers, this information is sufficient to obtain the value associated with a variable name at run-time. Searching the symbol table or the debug section for a variable name gives us the memory location associated with that name. A debugging tool can read the value from memory and display it according to the associated type

[3]Stabs and DWARF are formats for debugging information.

information[4].

To understand which variable is referenced by a specific memory instruction, we need to compute a reversed mapping, from memory location to object name. When parsing the symbol table and the debug section, we can look for entries associated with data objects and build a mapping from location information to object name. We construct one map for global variables, whose keys are ranges of absolute memory locations[5]. We construct separate maps for the local variables and formal parameters of each routine, in which keys are represented by ranges of stack offsets. For formal parameters of routines, we build also a mapping between parameter indices and parameter names.

One can compute a mapping from each load and store instruction to the name of the variable it accesses by monitoring the memory locations it accesses at run-time. Memory locations need to be collected only on the first execution of each memory instruction. Because of this, the run-time overhead should be small. A problem with such an approach is that for dynamically allocated data structures, only the pointer's location is stored in the debug section. Knowing the location of any element of a dynamically allocated array wouldn't tell us which data structure is referenced, unless we keep track of the location and size of dynamically allocated memory blocks as well. A second drawback is that by using a run-time monitoring approach, we end up collecting location information each time the instrumented code is executed,

[4]For optimized binaries the information stored at the location associated with a variable may not be the most up-to-date value if the register allocator assigns the variable to a register for the duration of the computation, or if the compiler decides to save the value in a special spill area on the stack and not update the permanent location unnecessarily.

[5]A range is defined by the starting address of an object and its ending address computed with the help of the object size information.

even though an association between a reference and a variable name is very static in nature.

Another approach is to use the symbolic formulas derived for each memory reference to understand what memory locations are accessed by a reference, and based on this information to derive a mapping between memory instructions and the names of the variables they access. As explained in section 3.1, we build these formulas by tracing backwards through the CFG of each routine along USE-DEF chains, until we either find that the definition of a register is a constant value, or we cannot trace back any further (*i.e.* we reach the top of the routine, or we determine that a register is defined by a function call). By tracing back up to the start of the routine, the formulas reflect the way the accessed locations are computed. Even when a reference accesses a field of an array of records, for which only a pointer to its starting location is stored in the debugging section, its *first accessed* symbolic formula will contain a term corresponding to the base pointer, plus additional terms to compute the starting address of an element of the array and then an offset corresponding to a field.

For a memory reference, the name resolution algorithm works by inspecting each term of its *first accessed location* formula as follows.

- If the term is a constant value and it is the only term of the formula, or if the term is a load from a constant address, then we search the global variables map for an address range containing the constant value. If found, we add the corresponding global name to a list of candidate names for this reference.

- If the term is a load from a stack offset, or if the formula consists of just

two terms (the stack frame register and a constant offset), we search the local variables and formal parameters map of the routine containing the memory reference, for a stack offset range encompassing the found offset. As before, if we find an entry, we add the corresponding name to the list of candidate names.

- Finally, if the term is a register that is used for argument passing on function calls according to the ABI[6], we compute the formal parameter index corresponding to this register and we search the formal parameters list of the routine containing the reference for the respective index. If an entry is found, we add the corresponding parameter name to the list of candidate names.

One will notice that the first two cases have two variants that are somewhat symmetric. Formulas that consist of just a constant value correspond to references to global variables that are statically allocated at compile time. A term that is a load from a constant address, typically corresponds to a global data structure which is dynamically allocated and only its header or a pointer to it is allocated at compile time in the data section. Similarly, the two variants of the second case correspond to automatic variables allocated on stack. In the first variant the entire object is located on the stack, while in the latter only a pointer to it is located on the stack.

The above algorithm may find several different names associated with a memory reference when, for example, we deal with a load from an array, where the element index is a function of other variables. In such cases we use two methods to filter out some of the names. First, we look at the stride formulas and applying the same

[6]On SPARC, registers $i0$–$i5$ contain the first six arguments of a function that has its own stack frame and register window. For optimized leaf procedures that do not have their own stack frame, registers $o0$–$o5$ contain the first six parameters.

algorithm as above on their terms, we remove the candidate names that are also found in the stride formulas, because we are interested in finding the name corresponding to the base term. Second, we filter out terms whose coefficients are greater than one, because we expect the base term of the formula to have coefficient one, while terms corresponding to an array index are usually scaled by a coefficient equal to the size in bytes of one array element.

The presented algorithm works well in practice. However, not all memory references can be successfully resolved to a name by the above algorithm. The main offenders are accesses to dynamically allocated local objects in optimized binaries. In such cases, our algorithm for computing symbolic formulas stops at the function call that allocates the dynamic memory, and we do not have the information to associate a function call with a specific local variable name. This happens in optimized binaries because the starting address of the allocated memory block, which is returned by the function call, is not saved by the compiler into any of the advertised stack offsets. We observed cases when the value was either kept in a register for the duration of the routine, or it was being temporarily stored in the spill area on the stack. Such cases could not be resolved at run-time either.

Chapter 4

Dynamic Analysis

Often, many of the important characteristics of an application's execution behavior, such as how it traverses data or how the amount of computation depends upon input parameters, can only be understood accurately by measuring them at run time. For instance, one cannot predict the behavior of a graph algorithm without knowing at least the graph size. Our toolkit uses binary rewriting to augment an application to monitor and log information about various aspects of its execution behavior.

To understand the nature and volume of computation performed by an application for a particular program input, we collect histograms indicating the frequency with which particular control flow graph edges are traversed at run time. To understand an application's memory access patterns, we collect histograms of the reuse distance [7, 44, 17]—the number of unique memory locations accessed between a pair of accesses to a particular data item—observed by each load or store instruction. These measures quantify key characteristics of an application's execution behavior upon which performance depends. By design, these measures are independent of any architectural details. We describe the approach for collecting each of these measures in turn.

4.1 Collecting Execution Frequency Histograms

The goal of our dynamic monitoring of computation is to produce a histogram of executed basic blocks. However, we do not need to insert a counter into each basic block to measure its precise execution frequency.

A routine's control flow graph has the same properties as a flow network. Each directed edge in a control flow graph has an execution frequency, just like each edge in a network flow has a stated capacity. The nodes in the graph are edge junctions. Except for the entry and exit nodes, all the nodes[1] in the graph have the property that the control flow that enters into the node must equal the control flow that leaves the node. This is the flow conservation property analogous to Kirchhoff's law for the physics of electrical current.

Using the conservation property and the observation that there is no need for more than one counter on a linear sequence of nodes and edges, Knuth and Stevenson [30] prove that a necessary and sufficient condition for measuring control flow graph edge and node execution frequencies is to add counters to a set of CFG edges such that each cycle in the undirected CFG has a counter on at least one edge. Ball noted [6] that a spanning tree of CFG edges has the maximum number of edges that do not contain a cycle. Building a spanning tree for the CFG and adding a counter to each edge not included in the spanning tree, monitors exactly one edge of each cycle—the minimum necessary.

[1]This is true in the vast majority of cases. Some programming mechanisms, such as setjmp/longjmp family of functions in "C", do not have the conservation property. However, these mechanisms are used in handling exceptions or unexpected errors and are seldom encountered in scientific codes.

Since an undirected graph with cycles does not have a unique spanning tree, the problem of placing counters on CFG edges does not have a unique solution. The solution we desire is one with counters on a set of edges that execute as infrequently as possible, to minimize the cost of runtime monitoring. To achieve this, we apply Kruskal's algorithm [16] to construct a maximum-weight spanning tree (MST) of the CFG—an acyclic subset of edges with maximum total edge weight that connects all vertices—with edges weighted by their expected execution frequency [6]. Our scheme for estimating execution frequency is described later in this section.

There are three remaining issues to address when adding counters: we want to count how many times a routine is entered, we want to have independent frequency measurement of edges that enter and exit an inter-procedural function call to get precise measurements in the presence of exceptions or setjmp/longjmp constructs, and our counter placement algorithm must avoid some of the CFG edges that cannot be easily instrumented. To address these issues, we exploit the property that counters are placed only on edges that are *not* part of the spanning tree. When we construct the maximum-weight spanning tree, rather than starting with an empty set of CFG edges, we initialize the tree to include all the uninstrumentable edges, as well as a virtual edge, e_v, added between the entry and exit nodes of the CFG. Adding e_v to the CFG ensures that there is at least one cycle, and by including it into the initial set of spanning tree edges ensures that a counter will be placed on a real, instrumentable CFG edge to count the routine's execution frequency. To get independent measurements across function calls, we also add virtual edges between each function call node and the CFG exit node, and we include all these edges into the

initial set of spanning tree edges. This will increase the number of inserted counters by one for each function call, and will ensure that the counter placement algorithm will provide independent measurements for the paths incoming and outgoing from a function call.

We use the following heuristic to estimate edge execution frequency, which we supply to the MST algorithm as edge weights:

- the entry node in the CFG has a weight of one;
- the weight of a vertex is divided equally among all its outgoing edges if none of these edges is a loop exit edge[2];
- each loop has a multiplicative factor equal to ten;
- the weight of a node is the sum of the weights of its incoming edges.

In addition to these four rules, a separate algorithm handles the loop exit edges and the nodes in which the exit edges originate. In most cases exit edges have their tail node in the program scope immediately outside the one that contains its head node. However, we have encountered cases in which an exit edge crosses several levels of a loop nest. To accommodate these cases, we apply the following algorithm to compute the estimated weight of an exit edge (e_i):

1. determine the outermost loop (L) for which this edge is an exit edge;
2. find the number (N_{exit}) of edges that exit loop L;

[2]A loop exit edge has its head node as part of a loop and its tail node outside the loop.

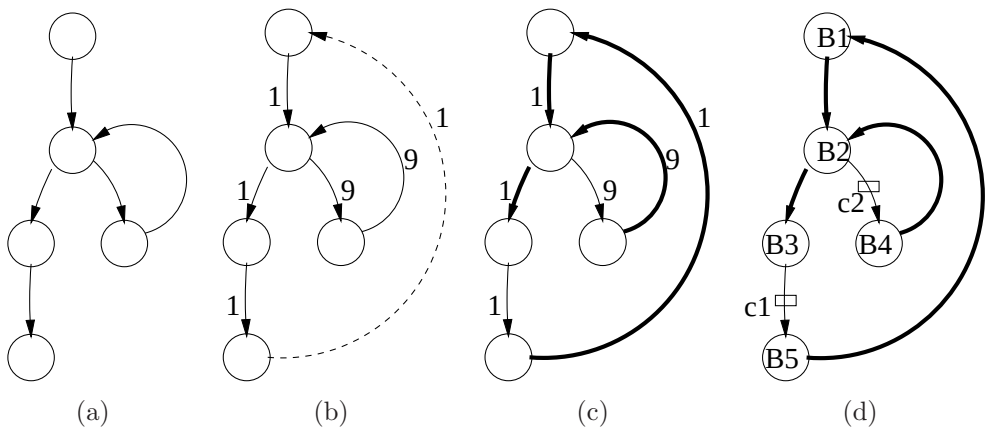


Figure 4.1: (a) Sample routine CFG; (b) Add a virtual edge from the EXIT node to the ENTRY node and estimate edges execution frequency; (c) Build the MST of the modified CFG; (d) Insert counters on edges that are not part of the MST.

3. the weight of e_i is the weight of the loop L's head divided by N_{exit};

4. all the other outgoing edges of the e_i's head node receive an equal fraction of the remainder weight of that node, after the newly computed weight of e_i is subtracted.

The final step consists of placing counters on the edges that are not part of the maximum spanning tree. Figure 4.1 presents a sample CFG with one loop and the steps that must be performed to determine the optimal insertion place for the counters. We count only the edges existent in the original CFG. From Figure 4.1(d) we can recover the execution frequency for all the blocks and edges in the CFG.

$$B1 = B3 = B5 = c1; B4 = c2; B2 = c1 + c2$$

4.2 Monitoring Memory Access Behavior

During execution, we characterize the memory access behavior of an application by monitoring the memory reuse distance seen by each reference. Characterizing memory access behavior in this way for programs has two important advantages. First, data reuse distance is independent of cache configuration or architecture details. Second, reuse distance is a scalable measure of data reuse which is the main determinant in cache performance.

For a fully-associative cache, one can predict if a memory access is a hit or a miss by comparing its reuse distance with the cache size (see Figure 4.2). Beyls and D'Hollander [8] show that reuse distance predicts the number of cache misses accurately even for caches with a low associativity level or direct mapped caches. However, reuse distance alone cannot predict conflict misses. In Section 5.3.2 we present an algorithm for approximating the number of conflict misses for a given cache architecture, using a probabilistic model.

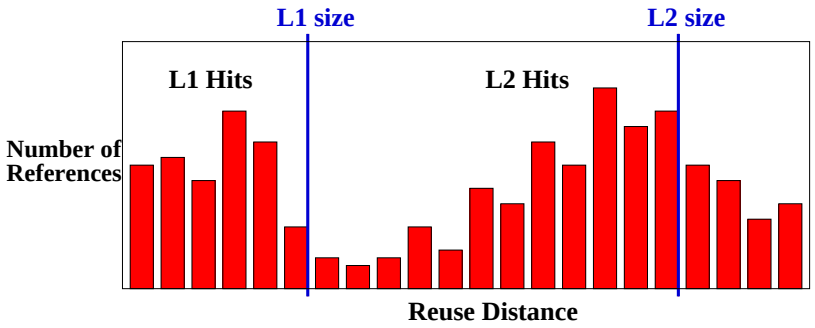


Figure 4.2: Example of reuse distance histogram. All references with reuse distance less than the cache size are hits.

We collect reuse distance information separately for each reference in the pro-

gram[3]. Before each memory reference we invoke a library routine that updates a histogram of reuse distance values (see Figure 4.2) for the reference. In addition to the address of the reference, the event handler needs to know the memory location that is referenced and the number of bytes touched by this instruction.

Our implementation of the event handler collects a complete histogram of the reuse distances seen by each memory reference. To compress the volume of output data, we coalesce each reference's histogram bins with similar distances before the data is written out. Our compression scheme has no noticeable effect upon the precision of our reuse distance models, but the reduction in space is often significant.

The event handler routine increments a logical clock by one each time a memory instruction is executed. A three level hierarchical block table is used to associate the logical time of last access with every memory block referenced by the program. The timestamp enables us to determine the reuse distance between a pair of accesses to the same datum. Alone, this data structure is only sufficient to count how many memory operations were executed since the last access to the same datum. To determine the number of **distinct** memory locations accessed between consecutive accesses to a particular datum, we use a balanced binary tree with a node for each memory block referenced by the program. The sorting key for each node in the tree is the logical time of the last access to the memory location represented by the node.

By using a unit size memory block, we can collect pure temporal reuse distance information. However, using this approach, we fail to observe spatial reuse in cache

[3]In section 5.2.2, we explain that we actually collect histograms for reference groups to improve modeling accuracy. Until that point, for simplicity we describe the process here as if each reference is handled separately.

lines. By using a non-unit memory block size, we can also measure spatial reuse because we collect the reuse distance of data blocks rather than data elements. To correctly account for spatial locality, we need to use a memory block size equal to the size of the cache line on the target architecture. Currently, to predict the performance of an application on arbitrary systems, we need to collect reuse distance data for all cache line sizes present on those architectures. The most common cache line sizes in use today are 32, 64 and 128 bytes. Because of the reuse distance data's dependence on cache line size, our characterizations of application behavior are not completely architecture independent. The size of the memory block used by our runtime library is defined by an environment variable; therefore collecting data for different cache line sizes does not require re-instrumenting the code or re-compiling the event handler routine.

On each memory access, our event handler executes the following pseudo-code, where $inst_k$ identifies the current memory instruction and $addr_i$ is the accessed memory location that maps to memory block b_i:

step 1 Set a logical timestamp t_i^{new} equal to the value of the *global timestamp counter* and increment the counter.

step 2 Find the entry into the hierarchical block table that corresponds to the memory block b_i (complexity $O(1)$). If b_i's entry has no assigned timestamp, then this is the first access to block b_i; increment the number of cold misses seen by $inst_k$, and record the t_i^{new} timestamp value into the block's table entry (all operations $O(1)$), and go to **step 5**.

step 3 b_i's table entry has a timestamp associated with it, and this access is a reuse of block b_i. Let timestamp t_i^{last} correspond to the previous access to b_i. Update the table entry for block b_i with the current timestamp t_i^{new}.

step 4 Delete the node with key t_i^{last} from the binary search tree. While searching for the node with key t_i^{last}, we count the number of nodes, D, with a key greater than t_i^{last}. Each node with key greater than t_i^{last} represents one distinct memory block that has been referenced since the previous access to b_i. Each node of the tree maintains a field *size* representing the number of nodes in the sub-tree rooted at it. For a binary search tree with larger keys to the right, D is a sum of terms $(node_j.size - node_j.leftChild.size)$, for every $node_j$ with key greater than t_i^{last} that is encountered on the path from the root of the tree to the node to be deleted. If the node with key t_i^{last} has a child to the right, we also add the value of its right child's size field to D. The delete and count operations have an aggregate $O(\log N)$ time complexity, where N is the size of the tree and represents the number of distinct memory blocks touched by the application. Record that $inst_k$ performed an access with reuse distance D.

step 5 Insert a node with key t_i^{new} into the binary tree. This step has complexity $O(logN)$.

The time complexity for computing the reuse distance seen by one memory access is $O(\log N)$. Overall, the overhead of collecting memory reuse distance information for the entire execution is $O(M \log N)$, where M is the number of memory accesses the program executes, and the space required by the data structures for monitoring

reuse distance is $O(N)$. Time and space complexities are both significant even with these optimized data structures.

The instrumentation infrastructure that we developed for collecting data about program behavior is quite flexible and can be easily adapted to collect other information or to perform different types of online analysis.

Chapter 5

Building Scalable Models

Measuring the dynamic characteristics of complex applications for large problem sizes can be expensive. To address this problem, for regular[1] scientific applications we collect dynamic data for small data inputs and we construct scalable models of the measured dynamic application characteristics. These models, parameterized by problem size or other input parameters, enable us to predict application behavior and performance for data sizes that would be too expensive to analyze at scale.

To build scalable models of dynamic application characteristics, we must collect data from multiple runs using different and determinable input parameters. To construct models parameterized by an input parameter, we use data from multiple runs in which that parameter is modified and other parameters are held constant.

[1] Applications whose algorithms are not adaptive based on the value of the input parameters, only the number of trips through loops and routines changes.

5.1 Models of Execution Frequency

To build scalable models for the basic block execution frequencies, we use quadratic programming [43] to model the execution frequency of associated counts of control flow graph edges using data collected from multiple runs with different input parameters. We use a modeling strategy implemented in Matlab to derive approximation functions written as a linear combination of a set of basis functions. The modeling program uses either a default monomial base or a set of user-provided bases in symbolic form. This approach enables logarithmic or other unusual terms to be considered when building the model.

The modeling program computes a set of coefficients to represent the linear combination of the basis functions that most closely approximates the collected data. Before it can compute the coefficients of the best fit function, the modeling program has to determine what basis functions are included in each model. We desire the minimum degree model that closely approximates the collected data. For this, we consider the basis functions sorted by degree, and the modeling program iteratively attempts to construct models of increasing degree, starting with a model of degree zero. At each step it computes an error of the fit. When there is no significant improvement in the accuracy of the fit by going to a higher degree, the iterative search stops, and the modeling program returns the model of the lowest degree that approximates the data well. We include restrictions to reduce or remove oscillations of the resulting fit and to ensure that the computed function is either convex or concave, depending upon the program characteristic that we are modeling.

Our approach works best with scientific codes that have predictable execution patterns, namely, ones that do not use adaptive algorithms. For such applications, we have computed accurate models of multiple parameters, although the process at this point is somewhat cumbersome [40, pages 46–53] and multiple parameter models could be computed more directly. For adaptive algorithms, we can produce an approximate model with reasonable accuracy in one parameter.

Execution frequency measurements recorded by our toolkit are first processed by a filter program that can be configured to either output counter frequencies or the number of executed instructions. Command line arguments control the output in each case. For example, counters can be sorted either by location (grouped by routines) or by frequency with the most executed ones first. Optionally, the output can be pruned relative to a significance threshold. Instructions are classified by type and can be aggregated at any level in the scope-tree of the program. We defined a set of generic RISC instruction classes and a module for translating native SPARC instructions into generic RISC instructions. The filter computes the number of executed instructions for each generic class and each basic block in the program. Using static analysis of an executable, the filter builds a scope-tree that reflects the program structure. There are three possible scope types that can be used to describe the program structure:

- *Program* scope - the root of the tree; its children are *routines*;
- *Routine* scope - the second level in the tree; its children are *loops*;
- *Loop* scope - can include any other number of *loops*.

If the binary contains source line mapping information, the routine and loop scopes

are annotated with source file information, including the name of the source file and the range of line numbers corresponding to that scope in the source file. When loops have overlapping ranges of line numbers and the same parent in the scope-tree, the filter performs a normalization step that folds together information for the loops. Overlapping ranges are the result of compiler optimizations such as loop fission, software pipelining, loop-invariant code motion, loop peeling, or tiling. A model of dynamic instruction count can be accumulated at program level, routine level or individual loop level.

5.2 Models of Memory Access Behavior

To predict an application's memory access behavior for a different problem size, we must model how each reference's reuse distance scales as a function of problem size. For this, we must first collect MRD data from multiple runs, with different and preferably small data sets.

Modeling memory access behavior is difficult. A single reference in the program may see cold misses and many distinct reuse distances. The simplest possible model of a memory reference's reuse distance would predict its average value for each problem size. However, such a model is almost always useless. Consider a reference performing stride one loads. Its first access to a cache line yields a long reuse distance; accesses to subsequent words yield short reuse distances. An average distance model can predict either all hits or all misses; neither prediction is accurate.

We need to model the behavior using histograms. A reuse distance histogram for

a reference contains a separate bin for each distinct distance encountered. We must model the structure and scaling of these histograms to understand the distribution of reuse distances as a function of problem size. Building meaningful models for histograms of reuse distance from executions with different problem sizes is challenging. Executions using different problem sizes result in histograms that each have a different number of bins and frequency counts; the varying number of bins complicates modeling.

One possible modeling approach is to divide each histogram for any reference or problem size into an identical number of bins using a fixed strategy regardless of the distribution of the data. How many bins to consider has an important impact on the size and accuracy of the models. A small number of bins will yield a compact model, but the model may lack precision. A large number of bins will improve model accuracy, but will add unnecessary complexity and cost to modeling for many references that use only a few different reuse distances. To avoid this problem, we examine a reference's collected data and pick an appropriate number of bins and their boundaries to adequately represent its histogram data across the range of problem sizes.

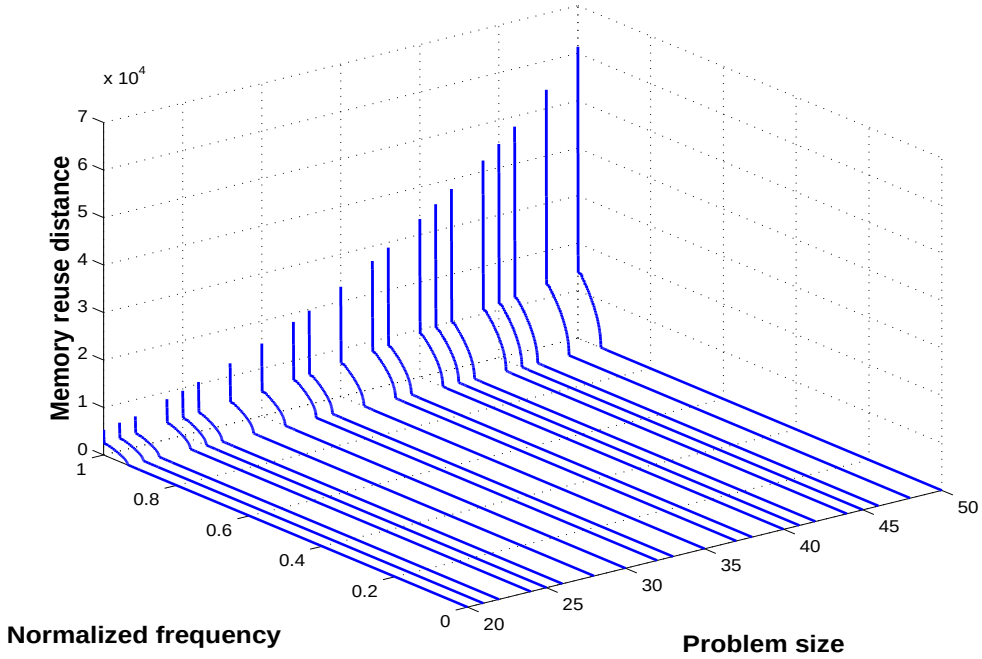
5.2.1 Modeling MRD Histograms

We sort the bins in each reference's MRD histogram by increasing distance. The first bin in a reference's histogram holds the smallest reuse distance for the reference. Figure 5.1(a) shows MRD histograms collected by our tool for different problem sizes for one of the most frequently executed memory accesses in the ASCI Sweep3D benchmark [19]. To illustrate our modeling algorithm, we plotted our models of MRD

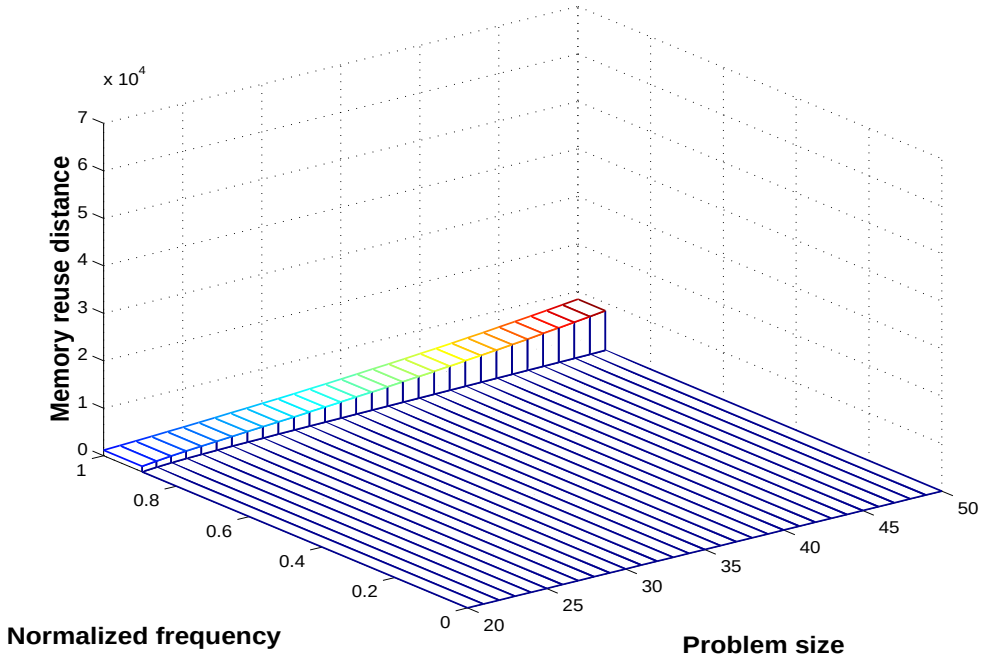
histograms in 3D Cartesian coordinates for each step of the algorithm. The `x` axis represents the problem size (from 20 to 50 in this case); the `y` axis represents the additive normalized execution frequency of the bins for each problem size, such that the total execution frequency of each histogram is one; and the `z` axis represents the reuse distance.

We begin our analysis by examining the leading bins of a reference's histograms for each problem size. If the leading bins have the same reuse distance across all problem sizes, it means they contain the fraction of accesses that have experienced reuse within the same or a fixed number of iterations of the innermost loop enclosing that reference. In general, this behavior is observed with strided memory accesses when we experience spatial reuse on consecutive iterations. The values of the constants depend upon the shape of the code in each particular loop, namely, how many accesses to other data structures are executed between two accesses to the same data structure during one iteration. These small distances are constant across all problem sizes because the shape of the code is invariant across problem sizes.

If a reference's histograms have such leading bins with a small constant distance, we model them separately (see Figure 5.1(b)), and remaining data is lumped together into one bin represented in the figure by a model of its average distance. Since the reuse distance of the leading bins is constant, we need to model only the execution frequency of these bins. The fraction of accesses that experience a fixed reuse distance is usually not constant across all problem sizes. The explanation for this is that for strided memory accesses, the regular reuse pattern across a fixed number of iterations d which produces the constant reuse distances, does not apply to the first or last d

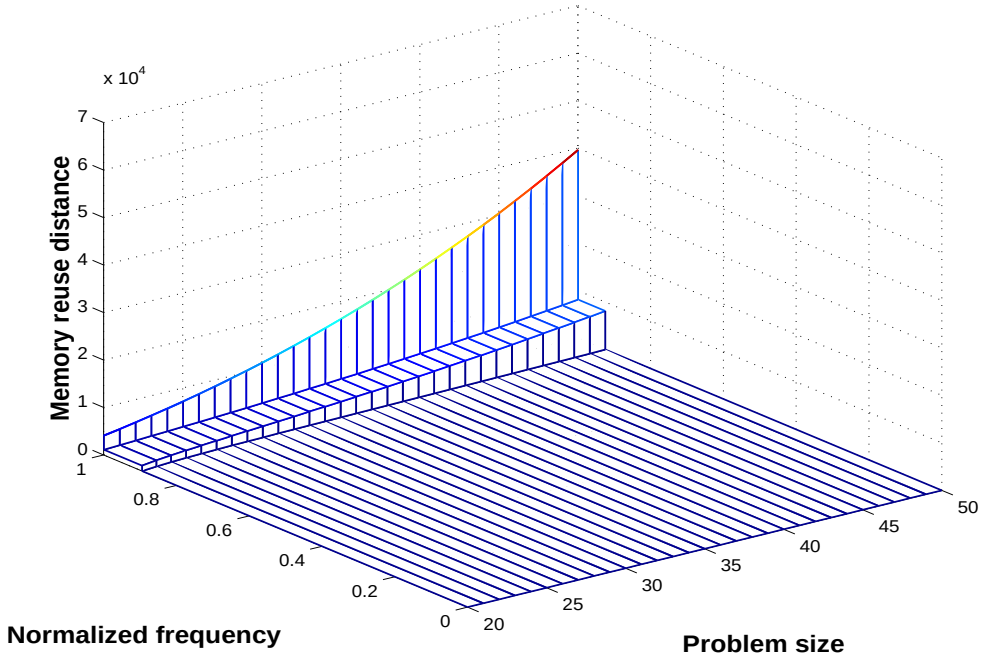


(a) MRD data collected for one reference in Sweep3D

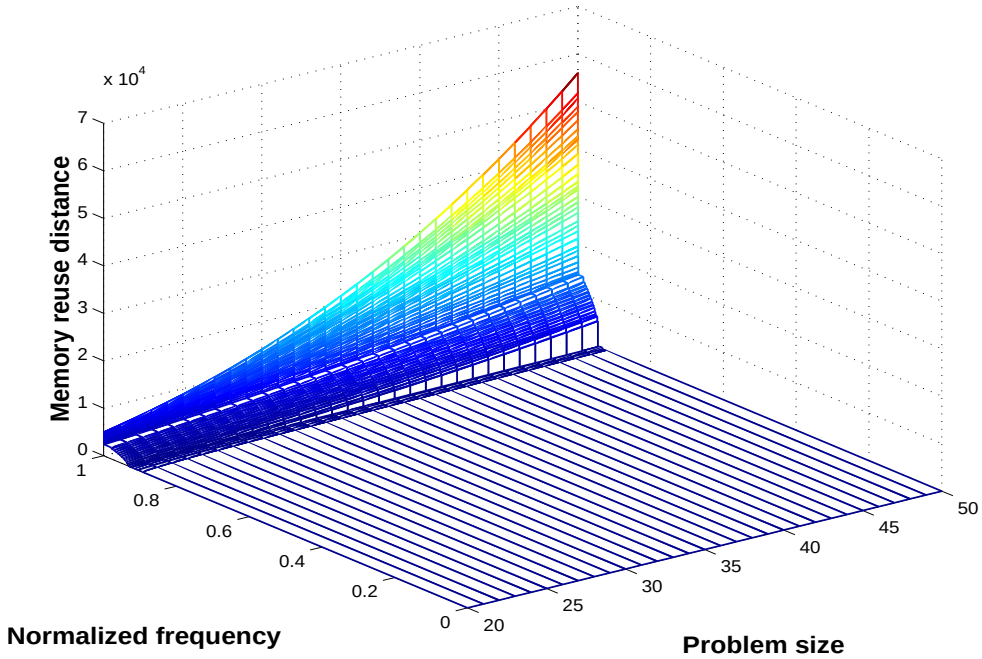


(b) Model constant distance first, and lump remaining data into one bin

Figure 5.1: Modeling MRD histograms



(c) First split of the non-constant data



(d) Final model for the data in (a)

Figure 5.1: Modeling MRD histograms (con't)

iterations. This includes spatial reuse to mesh like data structures, where the inner dimension does not cover an integer number of cache lines. Therefore, accesses to these incomplete lines experience less spatial reuse than accesses to cache lines that are completely filled. As problem size increases, the significance of the regular pattern increases asymptotically towards a constant limit which is a function of each particular code shape, and we are able to capture this behavior with our technique.

The remaining bins and their parameters are determined using a recursive algorithm. We start by computing an average distance for all references that were not modeled in the first step, and we build a model for their average distance as seen in Figure 5.1(b). Next, we recursively split the set of accesses in two and compute a model for each subset. The recursion stops when the models of the two resulting subsets are sufficiently close. We apply this algorithm to determine a partitioning of the data into an appropriate number of bins by considering the data for all problem sizes at once. At each step, we use a heuristic to determine how to partition the accesses. Its decisions influence the convergence speed, the accuracy, and the stability of the final model.

In our experiments, the partitioning heuristic that yielded the most stable and accurate results was one that selects partition boundaries such that the ratio between the number of accesses in the two partitions resulting from a split is the same across all problem sizes. With such an approach we need to model only the average reuse distance of each bin; execution frequency is easily computed by dividing the frequency of the parent bin proportionally with the splitting ratio. To compute the splitting ratio of a bin, we apply the following algorithm:

- Determine midpoint of the reuse distance range for each problem size, and compute the ratio between the number of accesses with reuse distance less than and greater than the midpoint value for each problem size. Using the reuse distance midpoint speeds up model convergence by favoring creation of narrow bins where reuse distance vary abruptly along the `y` axis, and wide bins where large fractions of accesses have similar reuse distances, with a minimal recursion depth.

- Select median ratio across all problem sizes, and use this median value as the splitting ratio for all problem sizes. We opted to use the median ratio to increase modeling stability in case the midpoint ratios for some problem sizes are significantly different.

Figure 5.1(c) presents a snapshot of the model after the first splitting step. If we look at the reuse distance histogram for problem size 50 in Figure 5.1(a), we see that the largest observed reuse distance is around 6×10^4. Then, we can approximate the midpoint reuse distance for this problem size at around 3×10^4. We notice that many more accesses have reuse distance under the midpoint value than above it. As a result, the two bins produced by the first split contain very different fractions of accesses, but they cover approximately equal ranges of reuse distance.

After partitioning, we perform a (rarely needed) coalescing step that examines adjacent bins and aggregates them together if they have similar polynomials describing their reuse distance. Our approach produces a minimal number of bins with almost no loss in accuracy. If a large fraction of accesses have comparable reuse distances

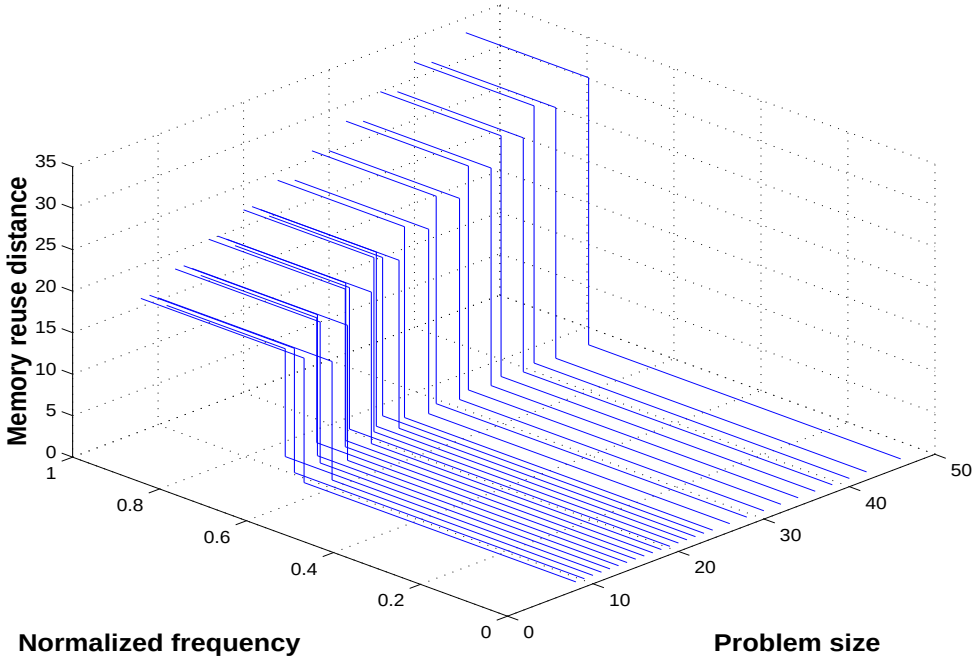
across all problem sizes, all those accesses go into one bin. However, if part of a reference's histogram is composed of many small fractions of accesses with different reuse distances, our approach produces a large number of bins for that part of the histogram and successfully captures the instruction's complex behavior. Figure 5.1(d) presents the final model computed for the data in Figure 5.1(a).

We notice that the reference depicted in Figure 5.1 has a complex behavior, with many fractions of accesses having very different reuse distances. Not all references have such complex behavior. Figure 5.2 presents the data collected and the model computed for another frequently executed reference from the Sweep3D benchmark. We can observe that its behavior can be captured using only two bins.

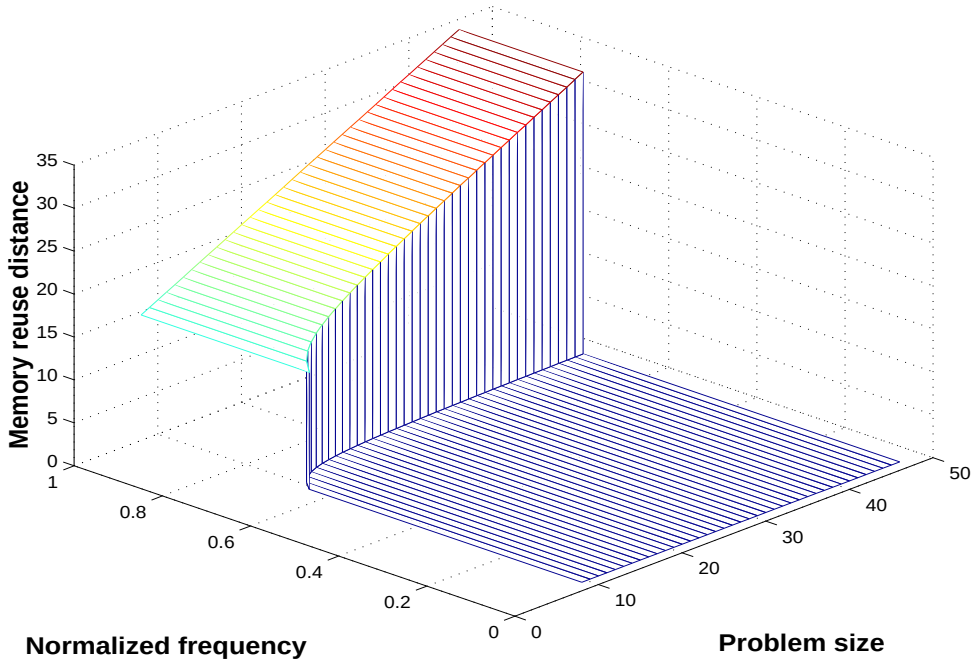
As with the models of execution frequency, our MRD histograms modeling strategy is currently implemented in Matlab using quadratic programming.

5.2.2 Considerations When Modeling MRD at Reference Level

Previously, we explained that we collect and model memory reuse data at reference level. Such an approach not only enables detailed predictions at instruction or loop level, but also enables more accurate models than if we collected aggregate reuse information for the entire program. However, modeling reuse distance data at instruction level is prone to errors due to the alignment of data in memory when the reuse information is collected for larger than unit size memory blocks. As we described in Section 4.2, to account for spatial reuse in cache lines, we collect reuse information of memory blocks, where the block size is equal to the line size of the target cache. Because spatial reuse distance uses a non-unit memory block size, it is



(a) Collected data



(b) Constructed model

Figure 5.2: Another example of MRD modeling

sensitive to data alignment and array dimensions. While the total number of cache misses at the loop level does not depend on the alignment of data, the spatial reuse distance for individual references is affected differently for distinct problem sizes and is a source of errors in the modeling step as we explain below.

Let us consider the *matrix_multiply* example from Figure 3.1 where the inner loop is unrolled once, and let us assume that array A is always aligned to the start of a cache line. Because the size of a cache line is a power of two, an even number of array elements will fit into a cache line. In our case, for an even value of N, the reference corresponding to $A[i, k+1]$ will always see a small reuse distance due to spatial reuse, because $A[i, k]$ will always perform the first access to a new cache line. However, for an odd value of N, $A[i, k+1]$ will access a new cache line first for odd rows of A, while $A[i, k]$ will access a new cache line first for even rows.

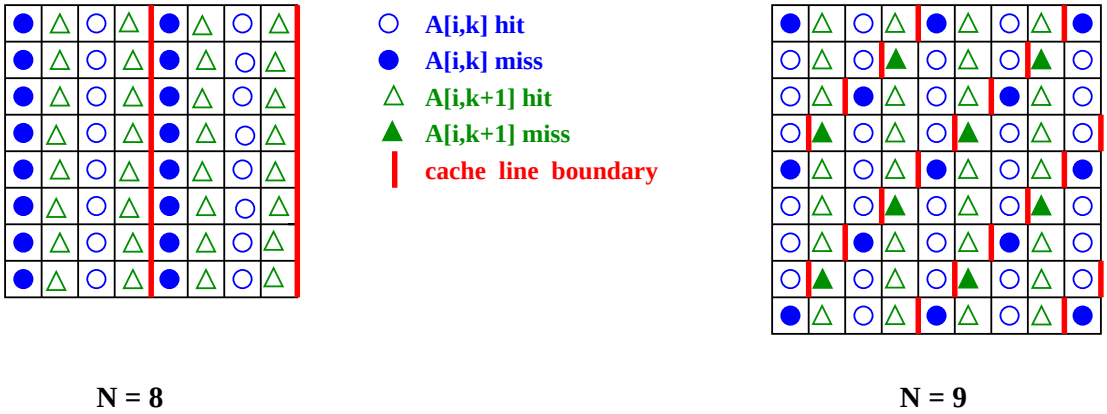
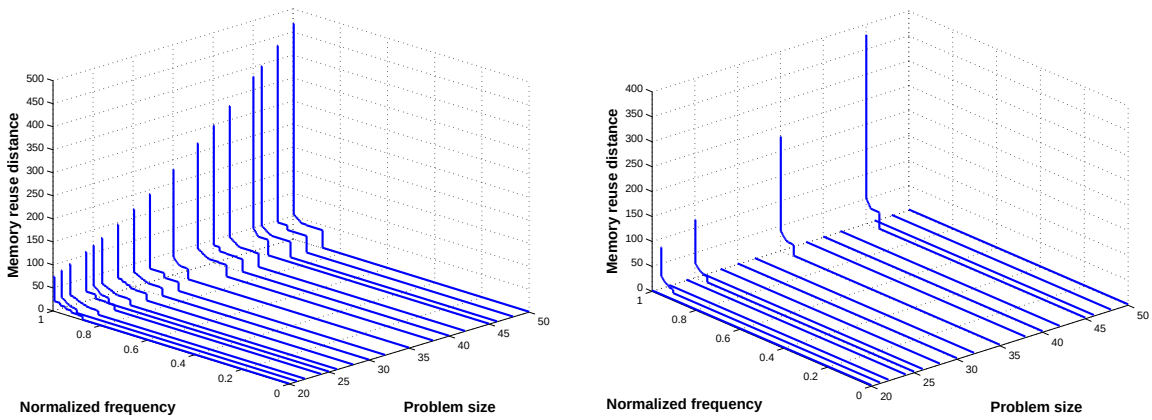


Figure 5.3: The distribution of first accesses to a new cache line for the two references to matrix A from the matrix multiply code presented in Figure 3.1, for an even problem size (N=8) and an odd problem size (N=9), assuming an architecture with a cache line that holds four double elements.

Figure 5.3 presents graphically this behavior for matrix sizes 8 and 9, assuming an architecture where the cache line size is four times the size of an array element. In

both cases, the total number of misses is approximately equal to one quarter of the number of memory accesses because only one miss occurs per four-element cache line. However, the distribution of reuse distances between the two references is different for each problem size, as seen in Figure 5.4. This problem is even more pronounced when the unrolling factor is greater and a larger number of references are affected.



(a) (b)

Figure 5.4: Memory reuse distance collected for (a) $A[i,k]$ and (b) $A[i,k+1]$, from the matrix multiply code presented in Figure 3.1.

Such inconsistencies between the reuse patterns at different problem sizes can cause large modeling errors for the affected references. However, if we consider $A[i,k]$ and $A[i,k+1]$ together, the union of their reuse distance data is consistent and predictable for every problem size, as seen in Figure 5.5.

A similar problem occurs in codes working on arrays of records when the cache line size is not a multiple of the record size. In such a case, depending on the record index, different fields can occupy the first position of a cache line. As a result, different references encounter a long reuse distance during the dynamic analysis depending on the record index. For this reason, at instrumentation time we find the sets of references

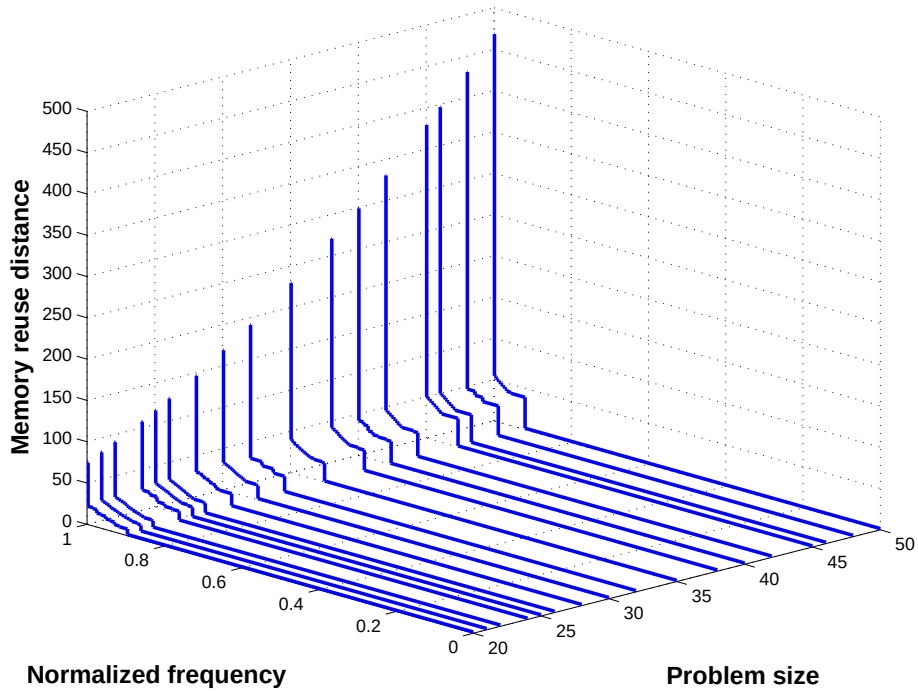


Figure 5.5: Aggregated memory reuse distance histograms of references to array A.

that have similar access patterns and insert code that collects a single reuse distance histogram for every such set.

We use the accessed data names recovered for each reference as in Section 3.2 and the symbolic formulas described in Section 3.1, to determine if two memory references are related and their data must be aggregated. Two references are aggregated only if they have similar access patterns and they both access data with the same name.

We say two references r_s, r_t located in the same loop have similar access patterns, if they have equal *stride formulas* relative to each loop containing them. In other words $Stride(r_s, L^k) = Stride(r_t, L^k)$ for every level k loop L^k containing them, $k \geq 1$. For our matrix multiply example presented in Figure 3.1(b), references $A[i, k]$ and $A[i, k+1]$, as well as references $B[k, j]$ and $B[k+1, j]$ have equal strides at each

loop level, and therefore they have similar access patterns. This is no surprise for somebody looking at the source code, but extracting such information from binaries requires the detailed static analysis we described.

After reuse distance histograms are collected, we perform additional static analysis to identify object code loops that have their origin in the same source code loop, and we perform additional aggregation between reference groups from these loops that have similar access patterns and access data with the same name. We extend the definition presented before to say that two references r_s, r_t located in different loops have similar access patterns, if they have equal *stride formulas* relative to each loop containing both of them, and their *stride formulas* relative to distinct, same level loops containing them have an integer ratio. In other words: $Stride(r_s, L^k_{st}) = Stride(r_t, L^k_{st})$ for every level k loop L^k_{st} containing both references, and $Stride(r_s, L^k_s)/Stride(r_t, L^k_t) = m/n$ for every pair of distinct level k loops L^k_s and L^k_t containing references r_s and r_t respectively, where m and n are integers and either $m = 1$ or $n = 1$.

Loop optimizations such as software pipelining and loop unrolling, split a source loop into multiple object loops: a main loop and a prolog or an epilog loop, which executes the remainder iterations. Compilers use loop unrolling aggressively. In addition, stencil computations found in scientific applications access multiple elements of an array with the same stride. As a result, there are many opportunities to aggregate references into larger sets, both inside the same loop and across adjacent loops. Figure 5.6 presents the distribution of the sizes of the memory reference groups derived for the NAS 3.0 BT benchmark. On the y axis we have the number of instructions

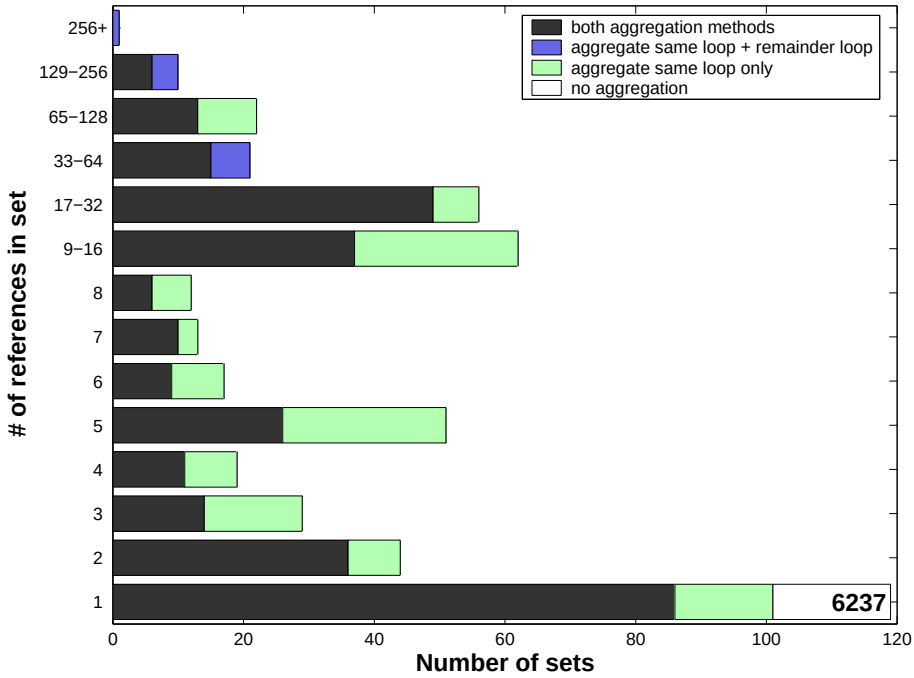


Figure 5.6: Distribution of the sizes of the instruction groups derived for benchmark NAS BT 3.0 when: (1) we perform no aggregation, (2) only references with similar patterns from the same loop are grouped together, (3) we aggregate across adjacent object code loops.

in a group, and on the x axis we see how many groups of that size were produced by aggregation. If we perform no aggregation, there are more than six thousand different groups, each with only one reference. During the instrumentation step only references with similar access patterns from the same loop are aggregated, and the number of distinct instruction groups reduces to 447. In the post-processing phase, after additional aggregation is performed across loops, only 329 groups remain.

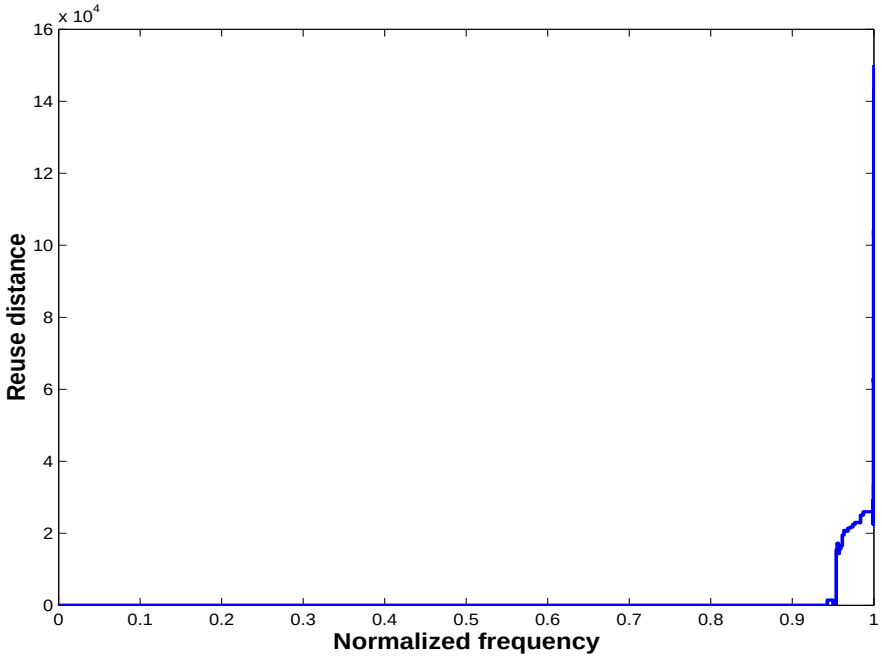
5.3 Evaluation of MRD Models

The problem of determining the ratio of hits and misses for a given cache size $\mathcal{C}$ is equivalent to determining the intersection of the model with the plane defined by

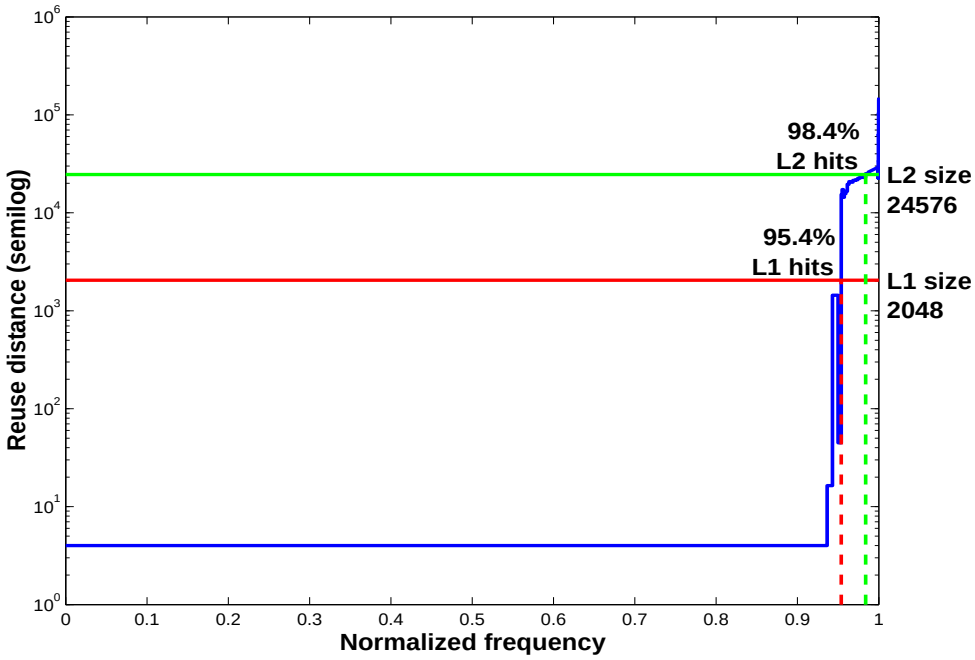
$z = \mathcal{C}$. Similarly, the problem of computing the expected behavior for one instruction at a given problem size $\mathcal{P}$ is equivalent to determining the intersection of the surface and the plane defined by $x = \mathcal{P}$. We can also determine the minimum cache size such that the hit-ratio is $\mathcal{H}$. The solution to this problem is the intersection of the model and the plane defined by $y = \mathcal{H}$. Any two of these three problems can be combined and the solution is the intersection of the surface with the corresponding two orthogonal planes.

5.3.1 Predictions for Fully-Associative Caches

For a fully-associative cache, we can use this approach to predict the ratio of misses for a given problem size and cache size. Figure 5.7(a) presents the expected behavior of the instruction modeled in Figure 5.1(d) at problem size 70. Assume that we want to predict the hit ratios for an architecture with two fully-associative levels of cache, where level one has 2048 blocks and level two has 24576 blocks. For this we must determine the number of accesses that have a reuse distance less than the specified cache sizes. Because the maximum reuse distance predicted for this reference is three orders of magnitude larger than the size of the target L1 cache, Figure 5.7(b) presents the predicted MRD histogram for problem size 70 on a logarithmic y-axis. The hit ratio is determined by the intersection of the predicted curve with the cuts corresponding to the sizes of the two cache levels. For this instruction and the considered target architecture, the model predicts a hit ratio of 95.4% for the L1 cache and 98.4% for the L2 cache.



(a)



(b)

Figure 5.7: (a) Evaluate model in Figure 5.1(d) at problem size 70; (b) Model evaluation at problem size 70 on a logarithmic y axis, and predictions for a 2048 blocks level 1 and 24576 blocks level 2 cache.

5.3.2 Predictions for Set-Associative Caches

In section 4.2 we defined memory reuse distance as the number of **distinct** memory blocks referenced between two consecutive accesses to the same memory block. If an access has reuse distance n, it means that we referenced n distinct other blocks since the previous access to the block currently accessed. For a fully-associative cache, any memory block can map to any cache block. Therefore, if the cache uses LRU replacement policy and has less than or equal to n blocks, we know that current access will be a miss because the n distinct blocks accessed since the previous access to this block have caused it to be evicted from the cache. Similarly, if the cache has more than n blocks, the current access is a hit because the accessed block was not evicted yet.

For a set-associative cache with s sets and associativity level k, a memory block can map only to one of the k blocks of a single set, where the set is uniquely determined by the block's location in memory. As a result, an access with reuse distance n is a hit if less than k out of the n accessed blocks map to this same set. The mapping of memory blocks to cache sets depends upon how data structures are laid out in memory. However, we do not collect information about the location of accessed blocks. As Hill and Smith noted in [23], we can estimate set-associative LRU distance from fully-associative LRU distance using a statistical model. This model is based on the simplifying assumption that accessed blocks are uniformly distributed in memory. In other words, the probability that two blocks map to the same set is $1/s$ and independent of where other blocks map.

With this assumption, we first compute the probability that exactly i blocks out of n distinct blocks map to a given set. We first notice that for $i > n$, the probability is zero because we cannot have more than n blocks map to a single set when there are n blocks overall. The mapping probability can be written as:

$$P_{mapping}(s, n, i) = \begin{cases} \left(\frac{1}{s}\right)^i \left(\frac{s-1}{s}\right)^{n-i} \binom{n}{i} & \text{if } i \leq n \\ 0 & \text{if } i > n \end{cases}$$

The probability formula for $i \leq n$ has three terms:

- $\left(\frac{1}{s}\right)^i$ because i blocks must map onto a specific set (the set of the currently accessed block)
- $\left(\frac{s-1}{s}\right)^{n-i}$ because the other $n - i$ blocks must map onto the other $s - 1$ sets.
- $\binom{n}{i}$ because any combination of i blocks out of the total number of n blocks can map onto our set.

The probability that an access with reuse distance n hits in a set-associative cache with s sets and associativity k can be written as:

$$P_{hit}(s, k, n) = \sum_{i=0}^{min(k-1,n)} \left(\frac{1}{s}\right)^i \left(\frac{s-1}{s}\right)^{n-i} \binom{n}{i}$$

and the probability of that access being a cache miss is 1 minus the previous formula:

$$P_{miss}(s, k, n) = 1 - \sum_{i=0}^{min(k-1,n)} \left(\frac{1}{s}\right)^i \left(\frac{s-1}{s}\right)^{n-i} \binom{n}{i}$$

This model fits very well with our MRD model, because we do not predict just an average distance for a reference, but a histogram of how many times each distance is encountered. For each bin of a reference's histogram we compute a miss probability as a function of the bin's reuse distance. The resulting probability represents the fraction of accesses in that bin that should be expected as cache misses.

In the case of a fully-associative cache we have only one set ($s = 1$) and k represents the number of blocks in the cache. If $n > k-1$, probability to hit in the cache is zero because $\left(\frac{s-1}{s}\right)^{n-i} = 0$ for any $i \leq k-1 < n$. If $n \leq k-1$, probability to hit in the cache is one because the sum reduces to a single term, $\binom{n}{i}$, where $i = n \leq k-1$. Thus, the formula is valid also in the special case of a fully-associative cache, although it is more efficient to use the direct method presented in Section 5.3.1 to compute the number of cache misses for fully-associative caches. However, we observe that while for a fully-associative cache each bin counts as either all hits or all misses, in the case of a set-associative cache a bin can have a dual behavior.

We can approximate the number of misses for a set-associative cache from the histogram of reuse distances predicted by our MRD model, with the following formula:

$$Num_{misses}(Hist, s, k) = \sum_{bin_i \in Hist} (P_{miss}(s, k, D_{bin_i}) F_{bin_i})$$

where D_{bin_i} and F_{bin_i} are the average MRD of bin_i and the execution frequency of bin_i respectively.

In the next section we are going to use this approach to predict cache miss counts for several cache configurations.

5.4 MRD Modeling Results

To validate our approach, in this section we compute cache and TLB miss predictions at the loop level for the ASCI Sweep3D benchmark and several of the NPB 2.3-serial and NPB 3.0 benchmarks, for mesh sizes ranging from 10^3 to 200^3. We compare our predictions against measurements using hardware performance counters on two different platform: an Itanium2 based machine and an Origin 2000 system based on the MIPS R12000 processor. The memory hierarchy characteristics for the two testbed machines are presented in table 5.1. On the Itanium, floating point loads

Level	# blocks/associativity/block size	
	Itanium2	R12000
L1D	256/4-way/64 B	1024/2-way/32 B
L2	2048/8-way/128 B	65536/2-way/128 B
L3	12288/6-way/128 B	–
L1 TLB[2]	32/fully/16 KB	64/fully/32 KB[3]
L2 TLB[2]	128/fully/16 KB	–

Table 5.1: Memory hierarchy characteristics for the testbed machines.

and stores bypass the small L1D cache and its associated L1 TLB. Because the benchmarks used in this test suite are all floating point intensive, the L1D cache and the L1 TLB of the Itanium2 machine have very little impact on their performance, and we do not present predictions for these two memory levels.

We build separate memory reuse distance models for each cache line size, all models being parameterized by one of the application's input parameters as described in the introduction of Section 5. To compute predictions of cache miss counts, we use

[2]A TLB behaves exactly like an LRU cache with a number of blocks equal to the number of entries in the TLB, and the size of each block equal to the size of the memory mapped by each entry.

[3]On the R12000, each TLB entry maps two consecutive pages, therefore the size of the memory mapped by an entry is 32 KB.

the size and associativity level of the target cache during evaluation as explained in Section 5.3. Nowhere in this process we make use of information such as the CPU's frequency or its number of execution units. Our cache miss predictions have nothing to do with the architecture of the CPU core. Therefore, while we consider only two platforms, we present predictions for six different cache configurations (two cache levels and one TLB level on each platform). From table 5.1 we see that the testbed machines cover a diverse set of cache configurations, including capacity, block size and associativity.

To compute the predictions, we compiled the benchmarks on a Sun UltraSPARC-II system using the Sun WorkShop 6 update 2 FORTRAN 77 5.3 compiler, and the optimizations: *-xarch=v8plus -xO4 -depend -dalign -xtypemap=real:64*. Measurements on the Itanium2 machine were performed on binaries compiled with the Intel Fortran Itanium Compiler 8.0, and the optimization flags: *-O2 -tpp2 -fno-alias*. On the Origin 2000 system we compiled the binaries with the SGI Fortran compiler version 7.3.1.3m and the optimization flags: *-O3 -r10000 -64 -LNO:opt=0*. We used the highest optimization level but we disabled high-level loop optimizations, because the sets of loop nest transformations implemented in the Sun, Intel and SGI compilers are different. Loop nest transformations change the execution order of the iterations of a loop nest, effectively altering an application's memory access patterns.

We computed predictions for the ASCI Sweep3D benchmark, a 3D Cartesian geometry neutron transport code, and several computational fluid dynamic codes, including BT, SP and LU from the NAS parallel benchmarks NPB 2.3 serial and NPB 3.0. The complete results at routine and program level of all these codes are

presented in Appendix B for the Itanium2 machine, and in Appendix C for the MIPS R12K system. In the following sections we are going to analyze the results of just a single benchmark on each of the systems, including results at loop level.

The NAS benchmarks use statically allocated data structures, with the maximum size of the working mesh specified at compile time. The benchmarks can be compiled in several standard classes named A, B and C, which have a maximum mesh size of 64, 102 and 162 respectively. We created an extra class L with a maximum mesh size of 200. We used static and dynamic analysis of the class A binaries to construct the models. The measurements on the Itanium2 and R12000 machines were performed on the binary of minimum class that accommodates that particular size.

To compute the predictions, we collected MRD data for block sizes 32, 128, 16 KB and 32 KB, for a set of problem sizes randomly selected between 20 and 50. We collected data on relatively small input problems to limit the cost of executing the instrumented binaries. Next, we built models of MRD parameterized by problem size for each of the applications, as described in section 5.2.1. Finally, to predict the cache and TLB miss counts, we evaluated the models at each problem size of interest. For each memory hierarchy level on each of the two machines, we predict a miss count for a fully-associative cache of the same size as the actual cache on the machine using only the MRD models, and a miss count that takes associativity into account using the probabilistic model described in section 5.3.2.

5.4.1 Predictions for Itanium2

Figure 5.8 presents the results for the hyper-plane 2D implementation of LU from NPB 3.0 on an Itanium2 machine. The graph on the top row presents the measurements and the predictions aggregated at the entire program level for this application. For the L2 and L3 caches we present both fully-associative and set-associative predictions as explained above. Because the TLB is fully associative, there are no set-associative predictions for it. For all graphs, the x axis represents the mesh size and the y axis represents the number of misses per cell, and per time step, where the number of cells is equal to $mesh_size^3$. This normalized view of the data enables us to understand how the application's characteristics scale with the amount of useful work, and at the same time makes the graphs more readable by bringing the counts for all mesh sizes to comparable levels. We measured reuse distance histograms for a range of problem sizes between 20 and 50, also indicated on each graph by two vertical lines. Using this data, we built scalable models of reuse distance and we predicted the cache miss counts for mesh sizes from $10 \times 10 \times 10$ to $200 \times 200 \times 200$.

While predictions are in general accurate, we notice that we under-predict the number of L2 misses for the LU benchmark at large problem sizes. The second row in figure 5.8 presents the predictions for two routines of the LU application, and Appendix B includes results for all routines that contribute at least 3% of the application's L2 cache misses. Looking at the routine level predictions, we notice that the entire L2 prediction error comes from routine *rhs*. Moreover, we have noticed that our models under predict the number of L2 misses for all implementations of the LU

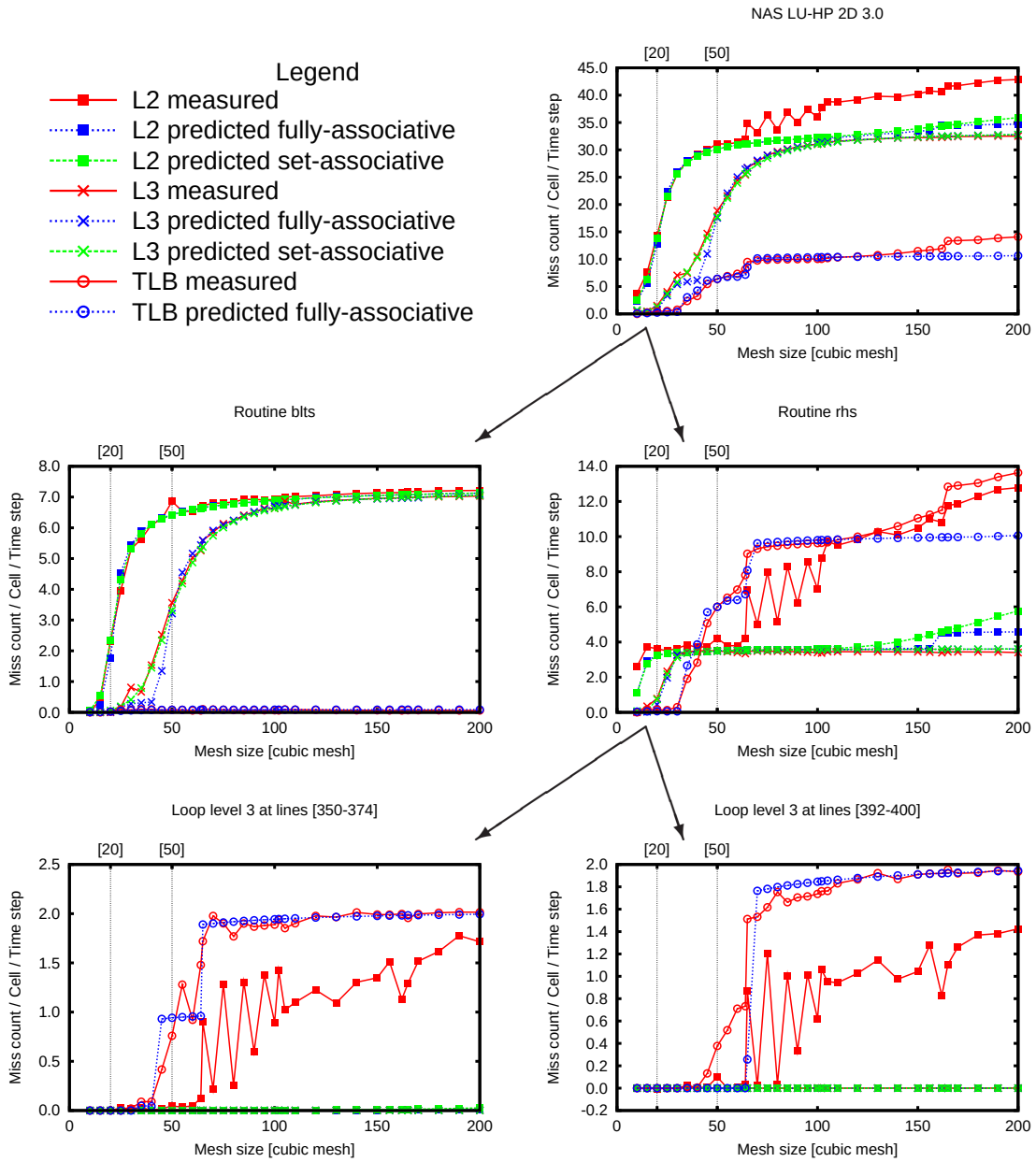


Figure 5.8: Predictions of L2, L3 and TLB misses for the LU HP 2D benchmark from NPB 3.0, on an Itanium2 based machine with a 256KB 8-way set-associative L2 cache, 1.5MB 6-way set-associative L3 cache, and 128 entries fully-associative L2 TLB. We present predictions for two of its routines and two level 3 loops.

benchmark in NPB 3.0 and in NPB 2.3, and in all cases the error was manifesting itself in routine *rhs*. In all these cases, we noticed a correlation between the higher

number of measured L2 misses and a high rate of TLB misses. As you can see from the graph for routine *rhs* in figure 5.8, almost all TLB misses measured for the LU application are produced by this routine.

On a TLB miss, the OS needs to find the entry for the offending page in the virtual page table which is stored in memory. On the Itanium, the page table is accessed through the L2 cache. This has the advantage that on a TLB miss, in addition to the offending entry being brought into the TLB, an entire cache line of page entries is brought into the L2 cache. Thus, successive TLB misses to neighboring pages are serviced much faster from the L2 cache instead of going all the way to memory. However, if an application accesses memory with a large stride, larger than the size of a memory page times the number of page entries that fit into a cache line, each TLB miss will have to go to L3 or to memory causing an L2 cache miss. This is what happens in routine *rhs* of the LU benchmark. We cannot predict these cache misses because they are not produced by the application explicitly, but are the result of an interaction between the architectural design and the application's access stride.

On the bottom row of figure 5.8, we present the predictions and the measurements for two level 3 loops from routine *rhs*. For these loops, we notice that the number of L2 misses predicted by the model is zero for all problem sizes, but the measurements on the Itanium2 show the code experiences L2 misses once it starts missing in the TLB.

5.4.2 Predictions for MIPS R12000

Figure 5.9 presents the results for the SP benchmark from NPB 3.0 on an Origin 2000 system. As with the Itanium results, each graph presents the normalized counts of cache and TLB misses. All L2 and TLB miss counts are scaled by a factor of 5 to bring them into the same range with the L1 miss counts and make the graphs easier to read. We present both the fully-associative and the set-associative predictions for the two cache levels, and only fully-associative predictions for TLB. While for the Itanium machine the difference between the fully-associative and the set-associative predictions is quite small due to the high associativity level of its L2 and L3 caches, on the R12000 with its 2-way set-associative caches we can notice a significant difference. Although based on the simplifying assumption that accessed memory blocks are uniformly distributed across sets, the set-associative predictions approximate well the measured counts. We cannot estimate precisely the conflict misses at each problem size (see the graph for Sweep3D in figure C.1), but the set-associative predictions capture the actual trend. Results for more applications at routine level are presented in Appendix C.

We'll analyze in more detail the SP benchmark from NPB 3.0. We selected this benchmark on the R12000 because of the large difference between its fully-associative and set-associative L1 predictions. Second row in figure 5.9 presents the results for two routines from benchmark SP 3.0. We selected routines that show different memory utilization profiles, to demonstrate the accuracy of the models with various memory access patterns. We notice that most of the SP's L2 miss prediction error is produced

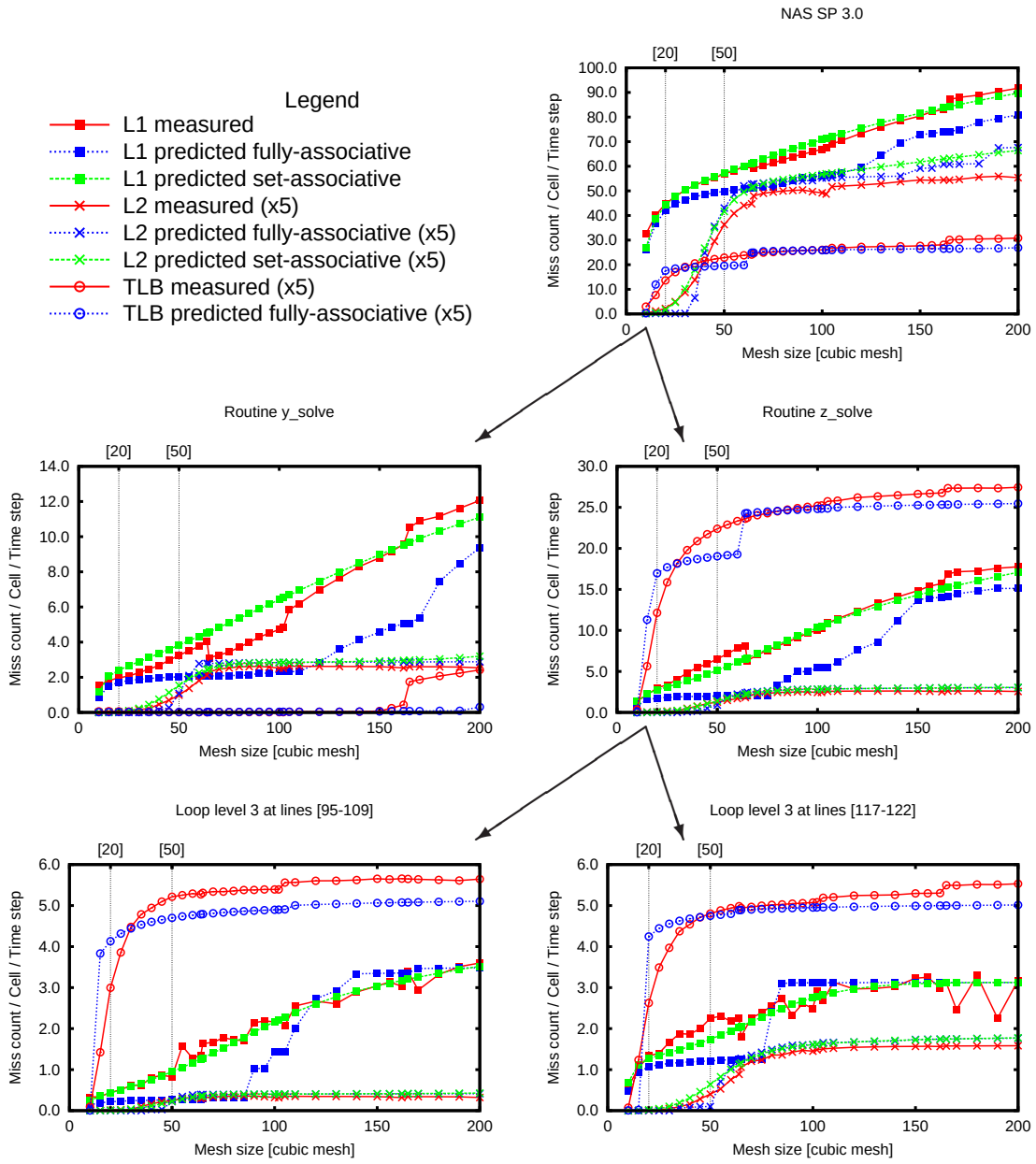


Figure 5.9: Predictions of L1, L2 and TLB misses for the SP benchmark from NPB 3.0, on a MIPS R12000 based machine with a 32KB 2-way set-associative L1 cache, 8MB 2-way set-associative L2 cache, and 64 double entries fully-associative TLB. We present predictions for two of its routines and two level 3 loops.

by routine *compute_rhs*, seen in figure C.9, and that almost all its TLB misses are produced by routine *z_solve*. If the 256 KB L2 cache on the Itanium seems too small

for caching page table entries in the presence of large stride accesses, on the R12000 with its large 8 MB L2 cache we did not notice an increase in the number of L2 misses due to a high rate of TLB misses. Last row in figure 5.9 shows two level 3 loops from routine *z_solve*. The L1 set-associative predictions approximate well the measured values for all problem sizes, and we can see the number of capacity and conflict misses at each problem size. The accuracy of the set-associative predictions validate in turn the accuracy of our MRD models which predict the fully-associative distances used by the probabilistic model.

Chapter 6

Cross-Architecture Predictions of Execution Time

To understand how an application performs on a target architecture, we combine information gathered from static analysis and dynamic measurements of execution behavior. Next, we map this information onto a model of the target architecture using a modulo instruction scheduler. The scheduler is logically organized into two modules: a front-end module that implements functionality dependent on the native binary, and a back-end module that works on an intermediate representation of the code to isolate the scheduler implementation from the underlying native architecture. At a high level, the scheduler can also be seen as a four step process.

1. Recover the executed paths through each routine. We reconstruct the control flow graph (CFG) of each routine using static analysis. The CFG is annotated with edge and basic block execution frequencies obtained either from direct dynamic measurements at scale or from evaluations of parameterized scalable models described in Chapter 5. From this representation we identify paths in the CFG and compute their associated frequencies. These paths serve as input for the other steps.

2. Translate native instructions into generic RISC instructions and compute an intermediate representation (IR) of the code. We defined a set of

generic RISC instruction classes and a module that translates machine instructions from the native binary into a representation based on generic RISC instructions. In addition, a dependence graph is constructed for each executed path. A dependence graph in which nodes correspond to generic instructions and edges represent dependences between instructions, is the intermediate representation of the native code used by the back-end module.

3. Model target architecture. We designed a machine description language (MDL) that we use to describe the resources available on the target architecture and the resources needed by each generic instruction during execution.

4. Schedule executed paths on the target architecture. We implemented a configurable modulo instruction scheduler that is instantiated with a description of the target architecture. The scheduler enables us to explore how an application would performs on different target architectures and to understand where execution cost is incurred.

Steps 1 and 2 work on the native binary and therefore are part of the scheduler's front-end module. Step 4 works with an architecture-independent representation of the code, implementing the bulk of the scheduling logic. The following sections describe these steps in more detail as follows. Section 6.1 describes the scheduler's front-end module. Section 6.2 illustrates the most important MDL constructs with snippets from our description of the Itanium2 architecture [28]. Appendix D presents the complete grammar of our MDL. Section 6.3 describes the implementation of the scheduler's back-end module. Section 6.4 provides an empirical analysis of the cost

of the scheduling algorithm. Section 6.5 presents cross-architecture prediction results for the ASCI Sweep3D code and several NPB 2.3 and NPB 3.0 benchmarks.

6.1 Scheduler Front End

The front-end module analyzes an application's binary and constructs an intermediate representation of the code to be used by the back-end module. To do this, it first recovers the executed paths through each routine converting the native machine instructions into generic RISC instructions, and constructs a dependence graph for each executed path where nodes represent generic RISC instructions and edges correspond to dependences between instructions.

To recover executed paths in loop nests, we work from inside out. A loop is replaced by a special InnerLoop instruction in its parent scope. As a result, no basic block is considered at more than one loop level. Once a loop is scheduled, we also compute information about registers that are live across its boundaries. We classify these registers as either input or output registers, depending on the loop boundary crossed by their live range. Input registers are those registers that are used before being defined inside the loop. Output registers are registers defined inside the loop that are live on the loop exit. We record the number of clock cycles from the start of the schedule to the first use of an input register and the number of clock cycles from the definition of an output register to the end of the schedule. Such information is used to compute register dependences between a loop and the instructions in its parent scope. In addition, inner loops and function calls act as fence instructions in

our schedules; they prevent instruction reordering across them.

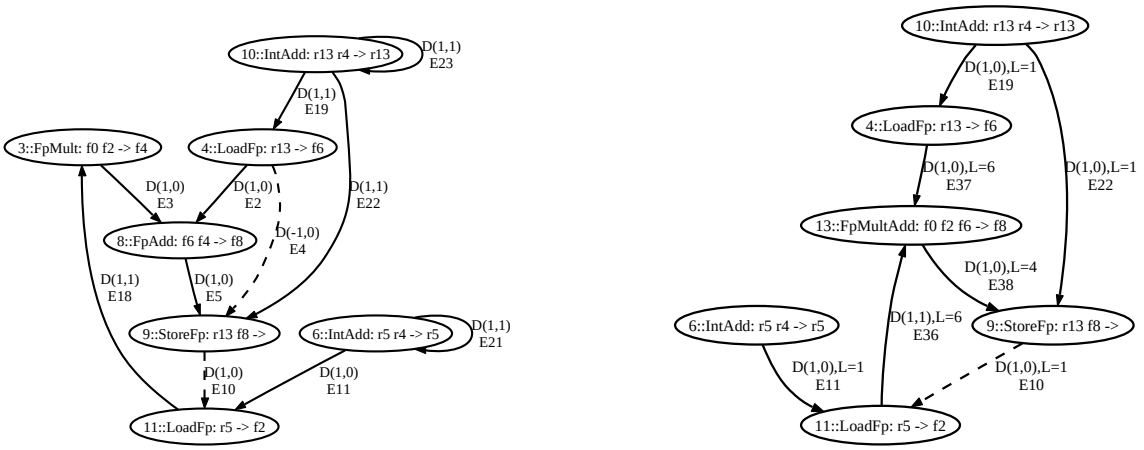
As seen in Figure 1.1, the scheduler works on an intermediate representation (IR) of the code. An appropriate format for the IR is a dependence graph, in which nodes represent generic instructions and edges represent dependences between instructions. Figure 6.1(b) shows the dependence graph for the innermost loop of routine `compute` shown in Figure 6.1(a).[1] Using such an intermediate representation has two main benefits. First, it isolates the scheduler from the binary analysis library underneath, making it portable to a different run-time system. We need to provide only a front end that translates machine code into the IR by identifying all schedule dependences among instructions. Second, it makes the scheduler more generic by separating it from the type of operations that need to be scheduled. Currently, the front-end analyzes RISC binaries and thus the scheduler and its predictions are targeted to a generic RISC instruction set. However, the scheduler is largely independent of the type of operations that must be scheduled. The scheduler can be used to analyze code sequences that include higher-level operations (i.e. FFT or dot product operations). The generic instruction set that is used affects mostly the target machine description and the front-end module. The machine description must specify execution units which can execute such operations and the architecture description model must define execution templates for each generic instruction type, while the front-end must recognize such operations and include them as nodes into the IR.

In our implementation, we defined a set of generic RISC instruction classes and

[1] To reduce clutter, the graph does not include the loop control arithmetic and the loop branch instruction.

```
void
compute(int size, double* A, double c1){
int i, j;
for (j=0 ; j<size ; ++j)
   for (i=0 ; i<size-1 ; i+=1){
      A[(i+1)*size+j] =
             A[(i+1)*size+j] +
             c1 * A[(i)*size+j];
   }
}
```

(a)



(b) (c)

Figure 6.1: (a) Sample source code; (b) IR for the inner most loop; (c) IR after replacement rules and edge latencies are computed.

our front-end translates SPARC machine instructions into the intermediate representation. The translation table for SPARC binaries is presented in Table 6.1. In addition to the generic RISC instructions presented in Table 6.1, our generic RISC instruction set includes a few more classes which do not have a direct mapping to the SPARC ISA, but are used to describe the instruction set of other architectures.

Native SPARC Instruction	Generic RISC Class
SETHI, ADD[X][cc], TADDcc[TV], SUB[X][cc], TSUBcc[TV], SAVE, RESTORE	IB_int_add
BIcc, BPcc, FBfcc, FBPfcc	IB_br_CC
BPr	IB_branch
CALL, JMPL, RETT	IB_jump
AND[N][cc], OR[N][cc], X[N]OR[cc]	IB_logical
SLL, SRL, SRA	IB_shift
MULScc, UMUL[cc], SMUL[cc]	IB_int_mult32
MULX	IB_int_mult64
UDIV[cc], SDIV[cc]	IB_int_div32
UDIVX, SDIVX	IB_int_div64
TICC	IB_trap
MOVR, MOVcc, RDCCR, RDASR, RDPSR, RDWIM, RDTBR, RDY, WRCCR, WRASR, WRPSR, WRWIM, WRTBR, WRY	IB_int_move
FADD{s,d,q}, FSUB{s,d,q}, FCMP[E]{s,d,q}	IB_fp_add
FDIVs	IB_fp_div32
FDIV{d,q}	IB_fp_div64
FMUL{s,d,q}, FdMULq, FsMULd	IB_fp_mult
FMOV{s,d,q}, FABS{s,d,q}, FNEG{s,d,q} FMOVR{s,d,q}, FMOVcc{s,d,q}	IB_fp_move
FSQRTs	IB_fp_sqrt32
FSQRT{d,q}	IB_fp_sqrt64
F{s,d,q,i,x}TO{s,d,q,i,x}	IB_fp_cvt
LDSB, LDSH, LDUB, LDUH, LD, LDD, LDSBA, LDSHA LDUBA, LDUHA, LDA, LDDA, LDX, LDFSR, LDCSR	IB_load_gp
LDF, LDDF, LDC, LDDC	IB_load_fp
LDSTUB, LDSTUBA, SWAP, SWAPA	IB_load_atomic
STB, STBA, STH, STHA, ST, STA, STD, STDA, STX, STFSR, STCSR	IB_store_gp
STF, STDF, STDFQ, STC, STDC, STDCQ	IB_store_fp
PREFETCH, PREFETCHA	IB_prefetch

Table 6.1: Classification of SPARC native instructions into generic RISC classes.

6.2 Machine Description Language

This section presents the main language constructs of the Machine Description Language that we use to describe the characteristics of a target architecture. Appendix D presents the full language grammar using railroad diagrams. At a minimum, the machine description language must specify the resources available on the target architecture and list the resources needed by each instruction type during execution. The following sections describe each of the language constructs in turn.

6.2.1 Execution Units

Figure 6.2 presents the language construct for defining the list of execution units (top) and an optional construct for defining special restrictions between units (bottom). When defining the available execution units, an optional multiplicity factor for each unit class can be included if there are multiple units of the same type. Duplicate units can also be distinctively enumerated using multiple names instead of using the optional multiplicity operator. However, in this case alternative instruction execution templates must be defined for each instance of the unit. Using the multiplicity operator simplifies the declaration of both the list of available units and the instruction execution templates. In addition, it provides a single point of control when playing with alternative machine designs that have different number of units of a given type.

The number and type of units defined is at the discretion of the machine designer. The only restriction is that each unit class must have a different name. Additional units can be declared just to simplify the definition of restrictions between different

List of execution units (EU):

```
CpuUnits = U_Alu*6, U_Int*2, U_IShift,
           U_Mem*4, U_PAlu*6, U_PSMU*2,
           U_PMult, U_PopCnt, U_FMAC*2,
           U_FMisc*2, U_Br*3,
           I_M*4, I_I*2, I_F*2, I_B*3;
```

Special restrictions between EUs:

```
Maximum 1 from U_PMult, U_PopCnt;
Maximum 6 from I_M, I_I, I_F, I_B;
```

Figure 6.2: MDL constructs for defining the execution units and restrictions between units.

instruction execution templates within the constraints of the language. Notice in Figure 6.2 that for the Itanium2 model we included also a list of issue ports (names starting will letter `I`) in addition to the execution units (whose names start with letter `U`). Using the convention that each instruction template must make use of one issue port of the proper type in addition to one or more execution units when the instruction is issued, we can restrict the number and type of instructions that can be issued in the same cycle. For example, to model the six issue width of the Itanium2 processor, the second restriction rule in Figure 6.2 specifies that at most six issue ports can be used in any given cycle.

6.2.2 Instruction Execution Templates

Figure 6.3 presents examples of instruction execution templates. An instruction template defines the latency of an instruction, and the type and number of execution units used in each clock cycle. The first instruction template in Figure 6.3 represents the most common format of template declaration, thus the shortest. It applies to

Instruction execution templates:

```
Instruction LoadFp template =
       I_M + U_Mem, NOTHING*5;
Instruction LoadGp template =
       U_Mem[0:1](1)+I_M[0:1](1);
Instruction StoreFp template =
       U_Mem[2:3](1)+I_M[2:3](1);
Instruction StoreGp template =
       U_Mem[2:3](1)+I_M[2:3](1);
```

Figure 6.3: MDL constructs for declaring instruction execution templates.

instructions that execute on fully pipelined symmetric execution units. On Itanium2, floating-point loads can be issued to any of the four memory units, and have a minimum latency of six cycles when data is found in the L2 cache. Thus, one `LoadFp` instruction is declared to need one issue port of type `I_M` and one execution unit of type `U_Mem` in the first clock cycle, plus five additional clock cycles in which it does not conflict with the issue of any instruction. *NOTHING* is a keyword which specifies that no execution unit is used. Instruction templates can make use of the multiplicity operator to specify consecutive clock cycles that require the same type and number of resources.

The other templates in Figure 6.3 show the extended form of declaring an execution template which is needed in case of asymmetric execution units. While floating-point loads can execute on any of the four memory type units, fix-point loads can be issued only to one of the first two memory units, and stores can execute only on the last two units. Thus, these templates use an optional range operator in square brackets to specify a subset of units of a given type, and the count operator between round parentheses to specify how many units of that type are needed.

The latency of a fix-point load instruction from the L1D cache is only one cycle, thus the `LoadGp` template has length one. Both store instruction types are declared with unit length templates because the units are fully pipelined and these instructions do not produce any value; thus, their latency is irrelevant because they cannot be sources of data flow dependences. Latency of memory and control dependences can be specified using bypass rules that we will present later. Instructions can have associated multiple execution templates, possibly with different lengths. For example, on the Itanium, many fixed-point operations can issue on either the `I_I` or the `I_M` ports, and they can execute on either the `U_Alu` or the `U_Int` units.

6.2.3 Instruction Replacement Rules

Instruction replacement rules, shown in Figure 6.4, are an important type of language construct used to translate sequences of instructions from the instruction set of the input architecture, into functionally equivalent sequences of instructions found on the target architecture. We introduced the replacement construct to account for slight variations in the instruction set of different architectures. For example, the SPARC architecture does not have a multiply-add instruction while Itanium2 does. Moving data between general-purpose and floating-point registers is accomplished on SPARC by a save followed by a load from the same stack location using registers of different types. Itanium2 provides two instructions for transferring the content of a floating-point register to a general-purpose register and vice-versa. In addition, the IA-64 instruction set does not include the following type of instructions: integer multiply, integer divide, floating-point divide, and floating-point square root. Floating-point

Instruction replacement rules:

```
Replace FpMult $fX, $fY -> $fZ +
        FpAdd $fZ, $fT -> $fD with
   FpMultAdd $fX, $fY, $fT -> $fD;

Replace StoreFp $fX -> [$rY] +
        LoadGp [$rY] -> $rZ with
   GetF $fX -> $rZ;

Replace IntMult32 $rX, $rY -> $rZ with
   SetF $rX -> $f1 +
   SetF $rY -> $f2 +
   FpMultAdd $f1, $f2 -> $f3 +
   GetF $f3 -> $rZ;
```

Figure 6.4: MDL constructs for declaring instruction replacement rules.

divide and square root operations are executed in software using a sequence of fully-pipelined instructions. The integer multiply and divide operations are executed by translating the operands to floating-point format, executing the equivalent floating-point operations, and finally transferring the result back into a fixed-point register. Our Itanium2 architecture description provides replacement rules for all these types of instructions.

A replacement rule consists of an input instruction pattern and an output instruction pattern. When the machine description file is parsed, all input replacement patterns are converted to our intermediate representation, a dependence graph. Registers in the input pattern that are used before being defined are classified as input parameters. Registers that are defined and then used in a later instruction create data flow dependences. The syntax allows for both register and memory type dependences. Control dependences are not supported as part of the replacement patterns.

Registers that are defined but not used by a following instruction are considered output parameters. Applying pattern matching on the intermediate representation of the code ensures that sequences of non-consecutive instructions can be matched by the input pattern if they are connected with the correct type of dependences. The replacement algorithm executes four main steps:

1. Search the IR code for a subgraph that matches an input pattern. If no match is found then we are done. Otherwise go to step 2.

2. If a match is found, associate actual nodes from the code with each node of the input pattern and identify incoming and outgoing dependences that correspond to the input and output parameters of the input pattern.

3. Convert the output pattern to our intermediate representation and insert the necessary nodes and edges into our code's IR. Connect these new nodes to the original code by creating dependences for the identified input and output parameters.

4. Remove the original nodes matched by the input pattern, unless any non-final node has additional outgoing edges not covered by the input pattern. Repeat from step 1 until no more matches are found.

To simplify the search algorithm and because we did not find the need for generalized pattern matching, currently we restrict input patterns to list-like graphs, while the output patterns can be arbitrary graphs. Figure 6.1(c) presents the dependence graph for loop `i` of the code shown in Figure 6.1(a) after the replacement rules were applied.

Bypass rules:

```
Bypass latency 1 for ANY_INSTRUCTION
       -> [control] InnerLoop;
Bypass latency 0 for ANY_INSTRUCTION
       -> [control] CondBranch |
                    UncondBranch |
                    Jump;
```

Figure 6.5: Example of MDL constructs for defining bypass latency rules.

One multiply instruction and one add instruction were replaced with a single multiply-add.

6.2.4 Bypass Latencies

Bypass latency rules, shown in Figure 6.5, are used to specify different latencies than what would normally result from the instruction execution templates for certain combinations of source instruction type, dependence type, and sink instruction type. Some combinations can be covered by more than one bypass rule. The first matching rule is always used by the scheduler. Therefore, one should write the exception rules first, and the more general rules last. However, all bypass rules have precedence over the instruction execution templates when computing dependence latencies. As a result, bypass rules and instruction execution templates can be interleaved in any order. The two bypass rules shown in Figure 6.5 refer to control dependences. For example, the first rule specifies that instructions within an inner loop and instructions within the parent scope that precede the inner loop cannot issue in the same clock cycle. The second rule specifies that a branch or function call instruction can be issued in the same cycle as an instruction that precedes it if there are no other types

List the Memory Hierarchy Levels (MHL):

```
MemoryHierarchy =
      L1D  [256, 64, 4, 32, L2D, 4],
      L2D  [2048, 128, 8, 32, L3D, 8],
      L3D  [12288, 128, 6, 6, DRAM, 110],
      DRAM [*, 16384, *, 0.04, DISK, 10000],
      TLB  [128, 8, *, 8, L2D, 25];
```

Figure 6.6: Example of MDL constructs for describing the memory hierarchy levels.

of dependences between them, even if the source instruction normally has a long latency.

6.2.5 Memory Hierarchy Characteristics

The MDL construct in Figure 6.6 defines the characteristics of the memory hierarchy. For each memory level, the parameters are: number of blocks, block size (bytes), associativity, bandwidth from a lower level on a miss at this level (bytes/clock), memory level accessed on a miss at this level, penalty in cycles for going to that level. The value of some attributes can be omitted and then a default value is used, depending upon the attribute type. Not all memory hierarchy information is used by the scheduler at this time. The number of misses at each level of the memory hierarchy are computed by an external program from models of memory reuse distance and these values are passed as input to the scheduler. The scheduler uses the miss counts and the latency information for each memory level to estimate the contribution of memory accesses to the application's execution time. In addition, the scheduler uses the block size information and the number of cache misses to infer the program's bandwidth requirements.

6.3 Scheduler Implementation

We implemented an architecture generic, critical-path driven, bidirectional modulo scheduler. It is close in concept to Huff's bidirectional scheduler [26], although we do not consider register pressure among the scheduling priorities at this time. The scheduler starts by pruning the dependence graph of edges that create trivial self-cycles, and edges that are determined to be redundant once the transitive property of dependences is taken into account. Next, the graph is transformed by applying the replacement operations specified in the machine description file, and all edges of the new graph are assigned a latency value based on the bypass latency rules and the instruction execution templates. Once the latencies are computed, the graph is pruned one more time, using the latency information to identify and remove trivial edges.

Once all dependences between instructions and their associated latencies are computed, the scheduler proceeds to compute the instruction schedule based on the following steps:

1. Identify opportunities for simplifying the dependence graph. Dependence graphs for large outer loops may contain a very large number of recurrences that differ in only a few edges. Computing all distinct recurrences, a mandatory step of the scheduling algorithm that we use, can be very expensive for such large loops. We look for opportunities to lower the complexity of dependence graphs by identifying *super-structures*, sub-graphs with unique entry and exit nodes. A *super-structure* can be logically replaced by an edge between its entry and exit nodes.

2. Compute minimum initiation interval (MII). Schedule length is bounded below by two factors, resource contention and dependence cycles. We compute the minimum schedule length that is theoretically achievable by considering each factor in isolation. The minimum initiation interval is given by the maximum of the two lower bounds.

3. Compute actual schedule length. In practice, most loops can be scheduled with a schedule length equal to the MII. In some cases, accommodating both dependences and resource constraints may yield a schedule length greater than the MII. We use an iterative algorithm to compute the feasible schedule length, k. We start with a schedule length of $k = MII$. We attempt to schedule all instructions from one loop iteration in k clock cycles, incrementing k each time the scheduling algorithm fails.

We describe each of these steps in more detail in the following sections. Sections 6.3.1 and 6.3.2 describe the process of computing the lower bounds due to resource contention and recurrences, respectively. Section 6.3.3 describes some preliminary priority metrics that are used during the scheduling step to determine the order in which nodes are selected for scheduling. Section 6.3.4 provides motivation for implementing a hybrid iterative-recursive scheduling algorithm. Section 6.3.5 describes the order in which nodes are selected for scheduling. Section 6.3.6 lists the properties of *super-structures* and proves a new set of dominance-based properties for nodes that belong to *super-structures*. Section 6.3.7 describes an algorithm that uses dominance information to find all *super-structures* in an arbitrary dependence graph.

6.3.1 Minimum Schedule Length Due to Resource Contention

An execution unit can be in use by at most one instruction in any given clock cycle. The lower bound due to resource contention, LB_{Res}, is determined by how tightly we can map all instructions from one loop iteration onto the machine execution units if we assume no dependences between instructions. The machine description may specify asymmetric execution units with multiplicity larger than one. To distinguish between different instances of asymmetric execution units, all units are explicitly represented in the scheduler's data-structures.

The machine description language described in Section 6.2 enables alternative instruction execution templates of possibly different lengths for the same instruction class. Therefore, we use a greedy algorithm to compute the lower bound due to resource contention. We first sort the loop's instructions in ascending order by the number of distinct execution templates associated with them. Then, we map each instruction onto available resources. When multiple choices are possible we select the execution template that makes use of the least utilized resources. At each step we consider the optional unit restriction rules when determining the execution units with the lowest utilization factor. After all instructions are mapped, the minimum schedule length due to resource contention is equal to the utilization factor of the resource in most demand.

$$LB_{Res} = \max_{u \in \mathcal{U}} (\text{uses}(u)),$$

where $\mathcal{U}$ is the set of available execution units and $uses(u)$ represents the number of clock cycles unit u is busy serving instructions from one loop iteration.

6.3.2 Minimum Schedule Length Due to Recurrences

Separately, we compute a lower bound due to recurrences, LB_{Dep}. For this, we assume a machine with unlimited number of resources and the bound is determined by the longest dependence cycle in the code. All graph edges have associated length and distance information. The length is given by the latency computed in a previous step. The distance is computed by the scheduler's front-end as part of its dependency analysis phase. Dependences can be either *loop independent* or *loop carried* [2]. Loop-independent dependences have both their ends in the same iteration and their distance is said to be zero, $D = 0$. For loop-carried dependences, the sink instruction depends on an instance of the source instruction from $d > 0$ iterations earlier, and the distance in this case is said to be d, $D = d$. For the example in Figure 6.1(c), edge *E36* from the `LoadFp` instruction to the `FpMultAdd` instruction is the only loop-carried dependence and has a distance of 1. All the other dependences in that graph are loop independent.

For each dependence cycle c, we compute the sum of latencies $L(c)$ and the sum of distances $T(c)$ over all its edges. Every recurrence must contain at least one carried dependence. As a result $T(c)$ is guaranteed to be strictly positive. If an instruction is part of a recurrence with total length $L(c)$ and total distance $T(c)$, then it can start executing no earlier than $L(c)$ clock cycles after its instance from $T(c)$ iterations earlier executed. Thus, each recurrence creates a lower bound on schedule length equal to $\lceil L(c)/T(c) \rceil$, and the lower bound due to application dependences is:

$$LB_{Dep} = \max_{c \in \mathcal{C}} \left\lceil \frac{L(c)}{T(c)} \right\rceil,$$

where $\mathcal{C}$ is the set of dependence cycles.

The minimum initiation interval becomes MII $= \max(LB_{Res}, LB_{Dep})$. In practice, most loops can be scheduled with a length equal to this lower bound. However, for some loops, accommodating both dependences and resource constraints increases the feasible schedule length. To find the schedule that can be achieved, we start with a schedule length k equal to MII and increase it until we can successfully map all instructions onto the available resources in k clock cycles.

6.3.3 Preliminary Priority Metrics

Because our scheduler works on a dependence graph in which a node stands for an instruction and an edge represents a dependency between instructions, I use interchangeably the terms *node* and *instruction*, as well as *edge* and *dependency* in the description of the scheduling algorithm.

Once the minimum initiation interval is computed, we compute additional metrics for each graph node and edge, information that is used to compute scheduling priorities. For each node in the graph we compute the longest path from a root node and the longest path to a leaf node. For this, we perform two DFS traversals of the dependence graph, starting from the root nodes and from the leaf nodes respectively. This step has complexity $O(N + E)$.

Each node receives also a resource contention score. For this, we use the contention information derived for each execution unit while computing LB_{Res} (see Section 6.3.1). The resource contention score of an instruction is computed as a dot product between the contention score of each unit and the square of the weighted average utilization

of each unit across all execution templates associated with this instruction. Note that the weighted utilization factor of each unit depends on the instruction type and is a value between 0 and 1. An unit with an utilization factor of 1 means that it is required in each execution template associated with an instruction. This unit utilization score is independent of the instruction mix inside a loop. Therefore, this information is associated with the machine state and it is computed only once for each instruction type, independent of how many paths we analyze. The resource contention score associated with each unit depends upon the mix of instructions in the code and it must be computed separately for each path. This process is explained in Section 6.3.1 and has complexity $O((N + I) \times (T + E))$, where N is the number of instructions in the path, I represents the number of generic instruction types, T is the maximum number of templates associated with an instruction, and E represents the number of execution units on the target architecture. Except for N, all factors are machine-dependent constants. Thus, the complexity of this step is $O(N)$.

For nodes and edges that are part of recurrences, we determine also the longest dependence cycle of which they are a part, and the list of distinct cycles to which they belong. This information is collected at the time we compute the dependence cycles. Many program loops have none, or just a few recurrences. Thus the overhead of tracking all cycles associated with one node or edge is fairly small. However, for some loops, particularly outer loops that contain many other inner loops and function calls, the number of different cycles can be extremely large. We mentioned before that function calls and inner loops act as fence instructions, which prevent reordering of instructions across them. This behavior is implemented by creating

control dependences from prior instructions to the fence instruction, and from the fence instruction to the instructions after it. The effect is that all dependence paths join at these fence instructions, producing nodes with large fan-in and fan-out values. As a result, the number of cycles can grow exponentially with the number of fence instructions in the graph.

To reduce the overhead, we make the realistic assumption that no dependency between two regular instructions can have a latency greater than the execution cost of an inner loop or of a function call. We can make this simplification because we use the resulting instruction schedule to evaluate performance, and thus, we do not have to be as conservative as an actual compiler. Based on this assumption, we do not create any dependences that cross a fence instruction. All dependence paths pass through the fence instructions instead. This assumption enables us to schedule the code between two fence instructions independently of the other instructions in the code. In other words, a loop that contains two or more inner loops and function calls, can be seen as a series of *segments* for scheduling purposes. A *segment* is a subgraph bounded by two fence nodes. Because *segments* have only one entry node and one exit node[2], they can be replaced in the original graph by *super edges* of length equal to the *segments*' schedule length.

We generalized the idea of *segments* to any subgraph that has a single input node and a single output node, not necessarily barrier nodes. We call such subgraphs *super-structures* and we can have hierarchical *super-structures*. Figure 6.7 presents a

[2]There are no dependences between instructions within a *segment* and instructions outside the segment, because such dependences would have to bypass a fence instruction.

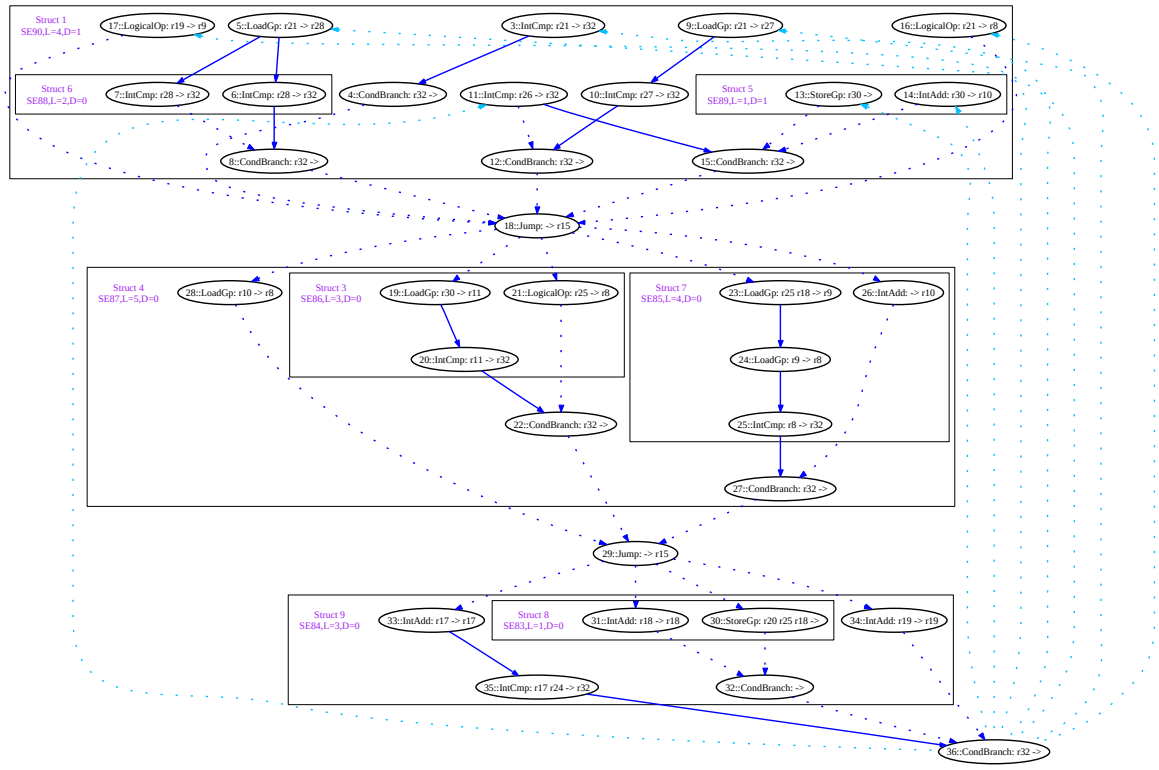


Figure 6.7: Sample dependence graph with hierarchical *super-structures*.

sample dependence graph and the *super-structures* present in the graph are marked with rectangles. Each rectangle includes the inner nodes of a *super-structure* and hierarchical structures are readily apparent. We developed an efficient algorithm to find all subgraphs that have single input and output nodes. This algorithm is presented in section 6.3.7.

Once *super-structures* are computed, we think of each *super-structure* as a single edge of length equal to the longest path through the *super-structure* for the purpose of computing longest paths from root nodes and to leaf nodes, and for computing dependence cycles, as explained in the first part of this section. With these changes, we maintain information about all distinct paths through each *super-structure* in addition to information about all distinct dependence cycle.

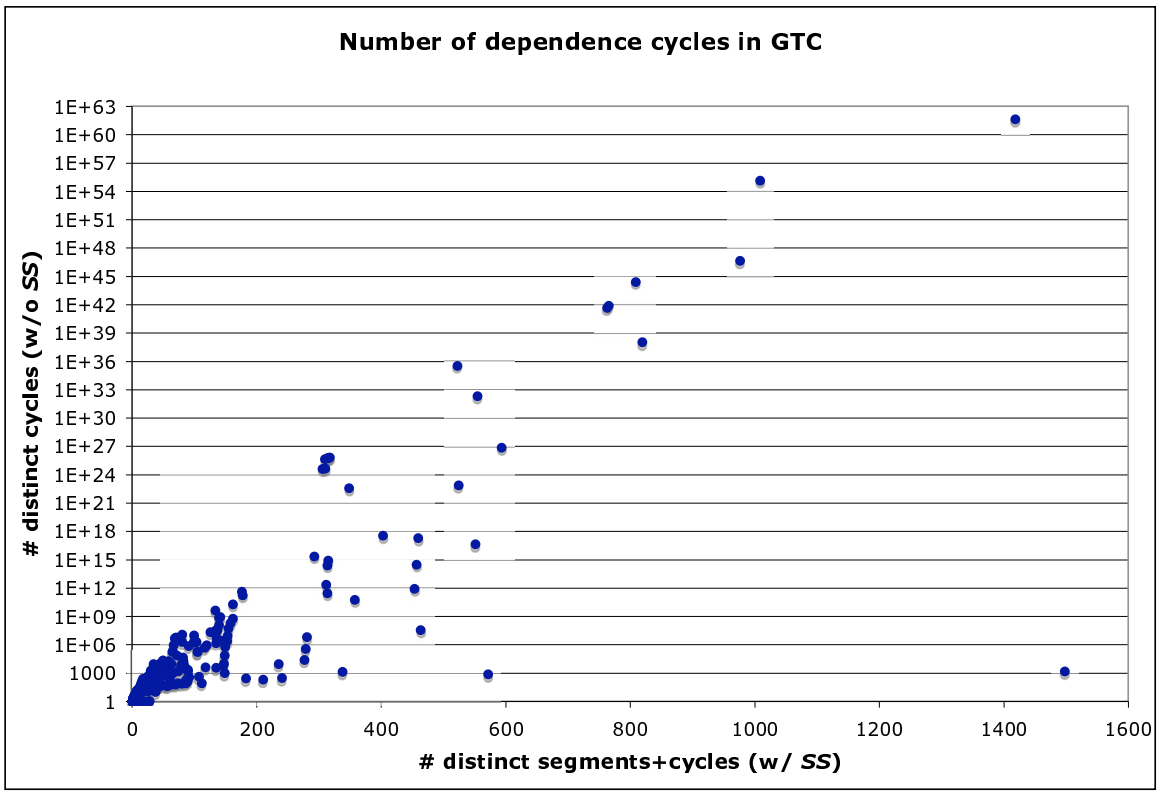


Figure 6.8: Distribution of the number of dependence cycles in GTC, using *super-structures* on the x axis, and without *super-structures* on the y axis (logarithmic scale).

To understand how much more efficient the scheduling algorithm is when using the *super-structure* information, Figure 6.16(a) presents a scatter plot of the number of distinct dependence cycles across all program scopes in the Gyrokinetic Toroidal Code (GTC) [34], with and without *super-structures*. The y axis presents on a logarithmic scale the number of distinct recurrences in the original dependence graphs. The x axis presents the number of distinct recurrences plus the number of distinct paths through *super-structures* of the same dependence graphs when we compute and use *super-structure* information. The figure includes data for all executed program scopes in GTC, with the exception of two outliers with coordinates $(12205, 46719)$ and $(8410, 1.71E+08)$, which if included would reduce the available resolution for

the other points on the x axis.

We notice that without the *super-structure* information, the number of distinct cycles can be as high as 10^{62} for a single scope. While these numbers may seem extremely high when compared with results reported in [57, 26], we must note that previous studies have focused only on inner loops of limited size and with no function calls. On the other hand, we use the modulo scheduling algorithm to compute schedule latencies for all program scopes, where inner loops and function calls are considered fence instructions, but we can still reorder instructions that are after the last fence instruction with instructions from the next iteration that are executed before the first fence instruction. In fact, the data points that have very large numbers of distinct dependence cycles when not using *super-structures*, correspond to scopes that contain multiple inner loops and function calls.

For each program scope, we compute separate schedules for each path taken through the control flow graph. For loops, we should have at least two distinct paths. One corresponds to the path that takes the loop back-edge, and the other corresponds to the exit path. We schedule the exit path separately because we try to model the effects of the wind-up and wind-down code segments usually associated with software pipelining. Thus, for the exit path, even though we use the exact same scheduling algorithm, we prevent software pipelining by adding control dependences from the loop control branch instruction to all root nodes of the dependence graph. This has the effect of producing a schedule with no overlap between instructions from different iterations. The dependence graph displayed in Figure 6.7 corresponds to such an exit path.

6.3.4 A Hybrid Iterative-Recursive Scheduling Algorithm

We map instructions one by one in an order determined by a dynamic priority function that tracks how much of each recurrence is still not scheduled. We use limited backtracking and unscheduling of operations already scheduled when the algorithm cannot continue.

To implement limited backtracking and at the same time preserve the higher efficiency of an iterative algorithm, we implemented a dual mode algorithm that can function either iteratively, recursively, or a mix of the two. Most instructions are scheduled in an iterative fashion. When the scheduler reaches a state that is determined to have a high chance of producing suboptimal code, a recursive call inside a try / catch C++ control structure is used to invoke the scheduler which then resumes its iterative behavior. Before each recursive call, the scheduler saves its state into the current stack frame. The state information is sufficient to understand the condition that led to the decision to go recursive, and it also includes the value of a global logical time clock at the time of the recursive call. The scheduler maintains a logical timer that is incremented on each scheduled instruction. Each scheduled instruction gets assigned the time stamp at the time it was scheduled. The time stamps enable us to keep track of the order in which the instructions were scheduled, and to roll back the scheduler to any previous state without using additional memory.

When the scheduler reaches a situation in which it cannot find any suitable issue slot for an instruction, it first attempts to unschedule some nodes upstream or downstream of current node based on an heuristic that takes into account the lengths of

chains of instructions scheduled in each direction, and then it searches for an issue slot again. If it still cannot find any issue slot, it tries to backtrack from one of the previously saved states. For this, it throws an exception which encapsulates information about the failure and the minimum time stamp among the already scheduled instructions that restrict the placement of the current instruction.

We limit the amount of backtracking both globally and at each recursive call level. The try / catch mechanism of the most recent recursive call will always catch the exception. The scheduler first checks if it exceeded the number of global retries. If yes, it will re-throw the exception after it sets the time stamp inside the exception to zero. Otherwise, it checks if it exceeds the number of local retries, and it re-throws the unmodified exception if it does. Finally, if the exception doesn't exceed any of retry limits, it checks that the time stamp associated with the recursive call is greater or equal than the time stamp specified in the exception, and that the failure information is compatible with the condition that caused the scheduler to go recursive. If any of these tests fails, the scheduler re-throws the exception.

If the exception can be processed at this level, the scheduler unschedules all instructions that were scheduled subsequent to this recursive call, as well as certain priorly scheduled instructions depending on the failure condition. It then tries to reschedule the node that caused the recursive call using a different issue slot, after which it resumes the normal scheduling process.

6.3.5 Selecting the Next Node to Be Scheduled

In the process of computing recurrences, we determine the strongly connected components (SCC) in the graph and we build a forest of directed DAGs of all SCCs, where an edge represents the presence of a dependence path from one SCC to another. Note that we cannot have cycles in the SCC DAGs. Otherwise the SCCs involved in the cycle would form a single strongly connected component.

Each node has associated multiple metrics, including the length of the longest cycle to which it belongs, the longest distance to the root and leaf nodes, and a resource utilization score as explained in section 6.3.3. The order in which these metrics are considered depends on the loop being resource or dependence constrained. Thus, if the loop has recurrences, the longest cycle metric is the most significant. Otherwise, the resource utilization metric is the first sorting criterion.

Assuming a dependence graph with multiple strongly connected components, we select first the node with the highest priority which must belong to the longest recurrence, and then assign it to an issue slot. We determine the SCC to which it belongs and we mark that SCC as constrained. Once we schedule one node of an SCC, the freedom of where to issue all the other nodes in that SCC is limited by the chain of dependences to the already scheduled node. Working on the DAG of SCCs, we also mark as constrained all the SCCs upstream and downstream of the current SCC if we follow only incoming or only outgoing edges respectively. Figure 6.9 presents a sample DAG and the constraining effect of scheduling a node part of $scc2$, or a node part of $scc1$.

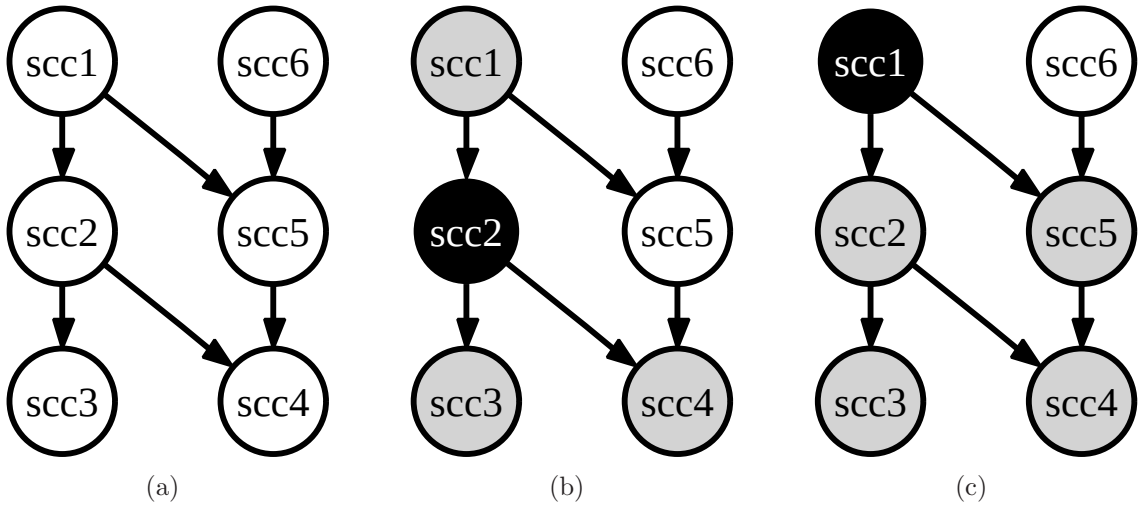


Figure 6.9: (a) Sample DAG of SCCs; (b) constrained SCCs when a node in scc2 is scheduled; (c) constrained SCCs when a node in scc1 is scheduled.

As long as there are unconstrained SCCs, we keep selecting and scheduling the highest priority node belonging to one of the unconstrained SCCs and we determine the new set of constrained SCCs at each step. Once there are no more unconstrained SCCs, the first phase of the scheduling process is completed. If the dependence graph did not contain any recurrences in the first place, then in the first phase we schedule only the highest priority node.

For all nodes already scheduled, we keep a priority queue of incident edges which we use in the second phase of the algorithm. An edge can be part of a cycle, or part of a dependence path connecting two SCCs, or just a simple edge. All edges contain a metric specifying how much is left to schedule which is computed as follows: for edges part of cycles this metric is computed as the maximum over all cycles to which it belongs, of the amount still not scheduled from each cycle; edges part of a path between two SCCs have associated pre-computed metrics which represent how far

away are the cycles upstream and downstream; finally, edges that are part of neither a cycle, nor a path between cycles, have associated pre-computed values specifying the longest distances to a root node and to a leaf node. Edges part of cycles have higher priority than edges that are part of paths between SCCs, which in turn have higher priority than simple edges.

In the second phase of the algorithm, we select edges based on the priorities described above. Once an edge is selected, we use modulo arithmetic to compute lower and upper bounds on the clock cycles in which the not yet scheduled end of the edge can be issued. We look only at a node's incoming and outgoing dependences to compute these bounds, we do not consider further neighbors. Thus, the time complexity of this step over all nodes is $O(E)$ because we look at each edge at most twice, not taking into account the backtracking which causes some nodes to be scheduled more than once, but which is rather an exceptional phenomenon.

Once a node is scheduled, we must insert all its incident edges whose other ends are still not scheduled, into the edges priority queue. However, before we can include an edge into the priority queue, we must compute the metric specifying how much remains to be scheduled, based on the edge type. For simple edges and for edges part of paths between SCCs this is a straightforward process; we use the pre-computed metrics described in Section 6.3.3 that specify the longest distances to root/leaf nodes or to upstream/downstream cycles, respectively. The choice between root and leaf nodes, or between upstream and downstream cycles is made based on which end of an edge has not yet been scheduled.

For edges that belong to recurrences we must traverse all cycles to which they

belong and update the amount left to be scheduled from each cycle, selecting the maximum value over all cycles. Each cycle is processed in $O(1)$; however, an edge may be part of many cycles. Computing the amount left to be scheduled for edges part of cycles is potentially the most expensive part of the algorithm. Its time complexity is $O(E\ cpe)$, where *cpe* is the average number of cycles per edge. The average number of cycles per edge is a function of both the number of cycles in the graph and their length. Some cycles may contain only two edges. At the same time, other cycles may be much longer. However, the maximum number of edges in a cycle is N because a node can be included at most once in each cycle. We will consider the length of a cycle to be a function of N. It is probable that a large graph contains longer cycles than a smaller graph. With these considerations, $cpe = C\,N/E$, where C is the number of distinct cycles in the graph, and the time complexity of this step becomes $O(C\,N)$. It is difficult if not impossible to derive the number of distinct dependence cycles in a graph analytically and the worst case scenario is total overkill. In Section 6.4 we analyze empirically the number of distinct recurrences and the observed complexity of the scheduling step, using measurements from three full applications.

The described scheduling policy has the following effects. First, a number of nodes with the highest priority are selected and scheduled until all strongly connected components are potentially constrained. Next, we schedule nodes bidirectionally along recurrences, where the priority function represents how much is left to schedule from each recurrence. Once all starting SCCs are scheduled, we start scheduling nodes along the paths that lead to other SCCs if there are still recurrences that are not completely scheduled. The fact that we schedule some nodes that are not part of

cycles before nodes that belong to recurrences can be seen as priority inversion. That is why we consider the cases where we reach a new recurrence following a path of edges that are not part of any cycles as states that have a high potential of producing suboptimal code. We invoke the scheduling algorithm recursively in these situations. When we have to backtrack from such a point, we rollback the scheduler to the saved state, but we also unschedule all nodes that are not part of cycles. Unscheduling all nodes that are not part of recurrences makes it more likely for the newly reached recurrence to be successfully scheduled. Once all recurrences are scheduled, the rest of the paths that do not lead to any new cycles are laid out in a similar fashion.

6.3.6 Properties of Super-Structures

While finding recurrences in general dependence graphs for large outer loops, we realized the number of distinct recurrences can grow exponentially with the number of inner loops and function calls. This is due to the large number of control dependences added to prevent reordering across fence instructions such as inner-loops and function calls. However, by disallowing any dependences to bypass the fence instructions, we can see the dependence graphs as a series of disjoint subgraphs connected by the fence instructions. This observation greatly reduces the complexity of the scheduling for outer loops that have at least two inner loops and/or function calls.

We encountered cases, however, where large dependence graphs with a single fence instruction contained a very large number of distinct recurrences. In one such case, the dependence graph of an outer loop from a weather modeling code contained almost 5 million distinct recurrences with just a single fence instruction. We noticed the

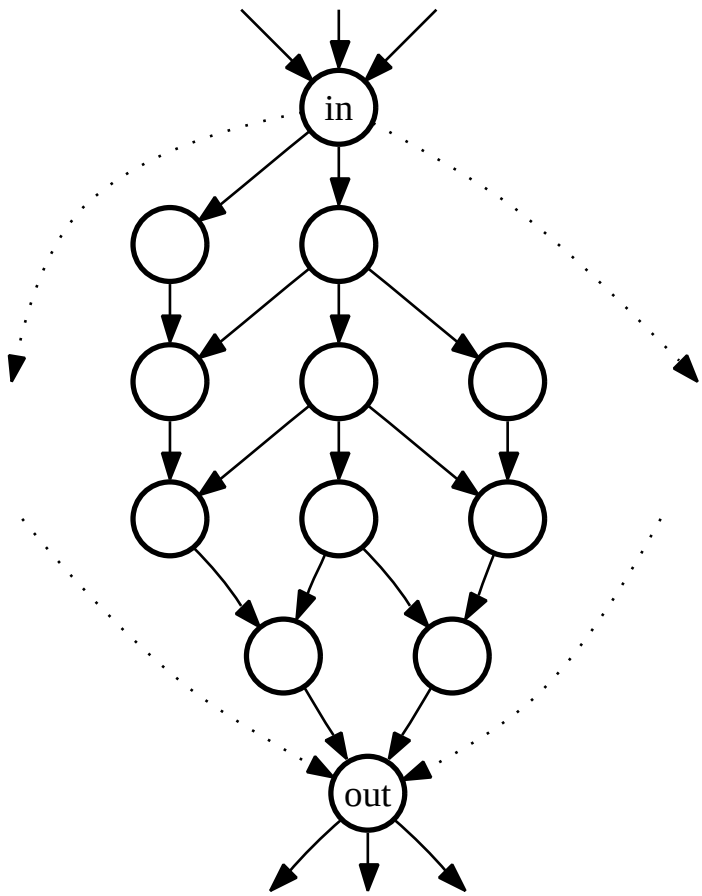


Figure 6.10: Example of diamond-like structures.

presence of many diamond like structures like the one in Figure 6.10, which produced recurrences that differ in only a few edges. As a result, we decided to generalize the idea that we used for fence instructions to arbitrary instructions. For this, we need an efficient and systematic algorithm to find all subgraphs $\mathcal{S}$ of a graph $G(N, E)$, that have a single input node r and a single output node s. We say r is the entry node or the root node of $\mathcal{S}$, s is the exit node or the sink node of $\mathcal{S}$, and $\hat{\mathcal{S}} = \mathcal{S} \setminus \{r, s\} \neq \emptyset$ is the set of internal nodes of $\mathcal{S}$. The nodes of a *super-structure* have the following

properties:

$$\forall v \in \hat{\mathcal{S}}, (u,v) \in E \Longrightarrow u \equiv r \ \vee \ u \in \hat{\mathcal{S}}$$

$$\forall v \in \hat{\mathcal{S}}, (v,u) \in E \Longrightarrow u \equiv s \ \vee \ u \in \hat{\mathcal{S}}$$

$$\forall v \in \hat{\mathcal{S}}, \exists u_0, \ldots, u_n \textit{ such that } r = u_0 \wedge u_n = v \wedge (u_i, u_{i+1}) \in E \textit{ for } i = 0, \ldots, n-1$$

$$\forall v \in \hat{\mathcal{S}}, \exists u_0, \ldots, u_n \textit{ such that } v = u_0 \wedge u_n = s \wedge (u_i, u_{i+1}) \in E \textit{ for } i = 0, \ldots, n-1$$

The first rule specifies that any predecessor in G of an internal node of a *super-structure*, is either an internal node of $\mathcal{S}$ as well, or it is the entry node of $\mathcal{S}$. The second rule is symmetric to the first one and specifies that all successor nodes of an internal node are either internal nodes or the exit node of $\mathcal{S}$. In other words, except for the entry and exit nodes, all nodes in $\mathcal{S}$ have all their input and output dependences in $\mathcal{S}$ as well. The last two properties specify that any internal node v is reachable from r following edges in G, and that the exit node is reachable from any of the internal nodes.

We want to find all nonempty sets of nodes that have the above properties, and their unique entry and exit nodes. Figure 6.10 presents such a subgraph for which the entry node is marked with "in" and the exit node is marked with "out." We notice that the entry and exit nodes can have any number of additional edges which are not part of $\mathcal{S}$. However, internal nodes cannot have any other dependences to nodes outside of $\mathcal{S}$.

We start by listing a few properties of the nodes that are part of a *super-structure* with entry node r and exit node s.

LEMMA 6.1 *All nodes on a path from r to an internal node $v \in \hat{\mathcal{S}}$ are also in $\mathcal{S}$. Second, all nodes on a path from an internal node v to the exit node s are also in $\mathcal{S}$.*

PROOF. Let $r = u_0, \ldots, u_n = v$ be a path from the entry node to the internal node v. By the first property, any predecessor of an internal node is either an internal node as well, or it is the entry node. By applying the first rule recursively, it results that $u_{n-1}, \ldots, u_1$ are all internal nodes of $\mathcal{S}$. Similarly, we can prove that all nodes on a path from an internal node to the exit node are also part of $\mathcal{S}$.

LEMMA 6.2 *Any internal node $v \in \hat{\mathcal{S}}$ is dominated by r and post-dominated by s.*

PROOF. The notion of dominance was introduced by Prosser in [55]. We assume the reader is familiar with the concept of dominators. More detail about dominators and algorithms for computing them can be found elsewhere [15, 35].

The proof for this lemma follows directly from the properties of a *super-structure*. Let v be an internal node of $\mathcal{S}$. We want to show that all paths from a root node of G to v pass through r. Assume there is a path $u_0, \ldots, u_n = v$ such that u_0 is a root node of G and $r \notin \{u_0, \ldots, u_n\}$. If $u_0 \in \mathcal{S}$, it means $u_0 \neq r$ and thus u_0 is an internal node of $\mathcal{S}$. However, by the third property of *super-structures*, u_0 must be reachable from r and thus u_0 cannot be a root node of G. Else, $u_0 \notin \mathcal{S}$. There must be a node u_i, $i = 0, \ldots, n-1$, such that $u_i \notin \mathcal{S}$ and $u_{i+1} \in \hat{\mathcal{S}}$. By the first property, all predecessors of an internal node of $\mathcal{S}$ are also in $\mathcal{S}$. Thus, u_i is either an internal node as well, or it is the entry node of $\mathcal{S}$. We have shown by contradiction that all paths from a root node of G to v must pass through r. As a result, r is a dominator of v. Similarly, we can prove that s is a post-dominator of v.

LEMMA 6.3 *For any internal node $v \in \hat{\mathcal{S}}$, all nodes on the path from r to v in the dominator tree, are also part of $\mathcal{S}$. In addition, all nodes on the path from s to v in the post-dominator tree, are also part of $\mathcal{S}$.*

PROOF. The proof for this lemma follows from the previous two lemmas. If r is not an immediate dominator for v, then any node u on the path from r to v in the dominator tree, is also on any path from r to v in the directed graph G. By the first lemma, $u \in \mathcal{S}$. Similarly, we can prove that all nodes on the path from s to v in the post-dominator tree, are also members of $\mathcal{S}$.

Based on these lemmas, we can derive the following properties for the subgraphs of interest:

- All nodes of the subgraph have a common dominator r called the entry node of the subgraph; r may not be the immediate dominator of all nodes.
- All nodes of the subgraph have a common post-dominator s called the exit node of the subgraph; s may not be the immediate post-dominator of all nodes.
- All nodes on the dominator tree path from a node v of the subgraph to the common dominator node r, are also part of the subgraph.
- All nodes on the post-dominator tree path from a node v of the subgraph to the common post-dominator node s, are also part of the subgraph.

6.3.7 An Algorithm for Computing Super-Structures

Based on the properties derived in the previous section, we can design an algorithm that uses the dominator and post-dominator trees to find all *super-structures*

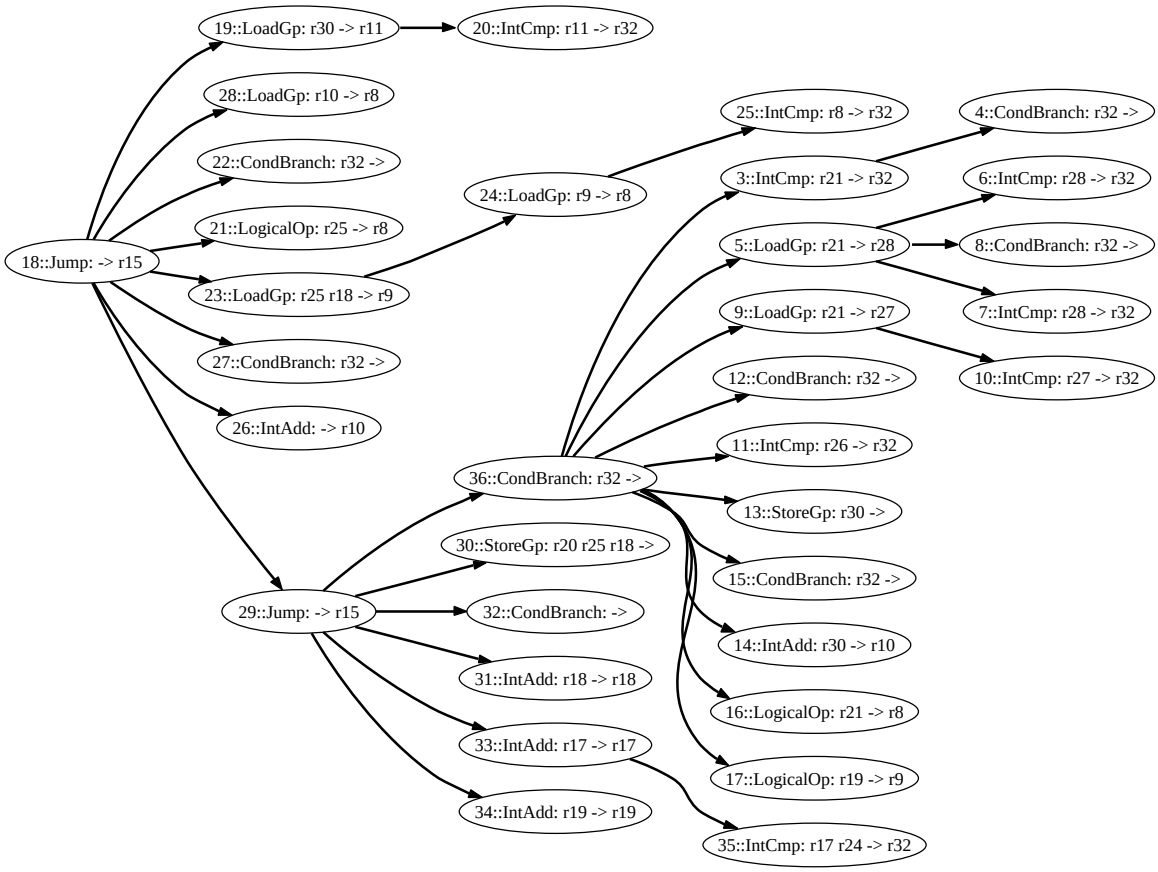


Figure 6.11: Dominator tree of the dependence graph presented in Figure 6.7.

in an arbitrary dependence graph. While dependence graphs can be arbitrarily complex, dominator trees are much more structured and easier to analyze. We use the sophisticated version of the Lengauer-Tarjan algorithm [35] to compute dominator and post-dominator trees. This algorithm has complexity $O(E\,\alpha(E, N))$ where E is the number of edges in the input graph, N is the number of nodes, and $\alpha(E, N)$ is a functional inverse of Ackermann's function.

Dependence graphs are in general complex and not well structured. For example, there may be graphs with no true root nodes, *i.e.*, nodes with no predecessor. If all nodes are part of recurrences or depend on nodes part of recurrences, we need

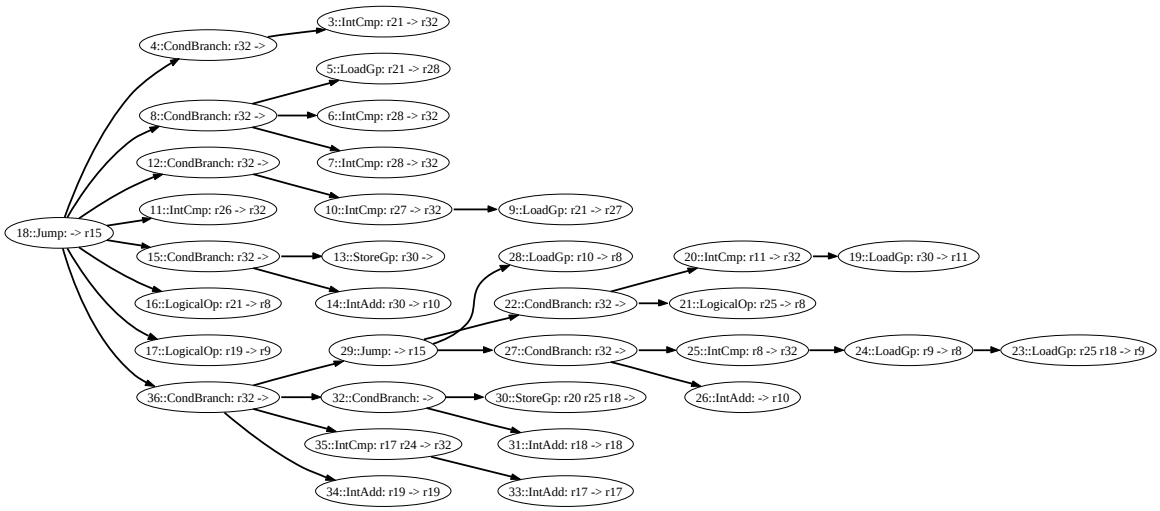


Figure 6.12: Post-dominator tree of the dependence graph presented in Figure 6.7.

to break some dependence cycles by dropping some edges, to create root nodes. All nodes in the graph must be reachable from at least one root node. When we do have multiple graph root node, the dominator tree will be a forest. The root nodes do not have any dominators; as a result, they cannot be part of any *super-structure*. Because the choice of edges to be dropped is not unique, the set of root nodes is not unique in cases when we have to break dependence cycles. The algorithm for finding *super-structures* is dependent upon which nodes are selected as root nodes, because the dominator algorithm depends on this information. It is preferable to select root nodes that give us a minimum section through the graph. Therefore, if we have any fence instructions, we select the first fence node as our unique root node, because all graph nodes are part of recurrences that include all the fence nodes.

Similar things can be said about the leaf nodes and the post-dominator tree. If we have any fence instructions, we select the root of our post-dominator tree to be the same fence instruction that we used as root for the dominator tree. Such a

selection maximizes the number of *super-structures* that are found. If we have no fence instructions and we need to break some recurrences, the leaf nodes are selected to be consistent with our selection of root nodes, *i.e.*, using the same set of dropped edges. Figures 6.11 and 6.12 show the dominator and the post-dominator trees of the dependence graph presented in Figure 6.7. Because that dependence graph contains several fence instructions, we selected the first one as the root of our both dominator and post-dominator trees.

The algorithm for computing *super-structures* starts with an educated guess about the entry and exit nodes of the *super-structures* containing each graph node. Over several steps this information is refined until when, in the end, we have the correct entry and exit information for all nodes. In a final step, we traverse all nodes and group together those nodes that have the same entry and exit nodes, and we remove the trivial *super-structures* that contain only a single path.

In the initialization phase, lines 1–5 of Algorithm 1, we assign a *minEntry* node and a *minExit* node to each graph node v. The *minEntry*$[v]$ and *minExit*$[v]$ information represents an initial guess of the entry and exit nodes of the *super-structure* containing v. Based on Lemma 6.2, we initialize *minEntry*$[v]$ with the immediate dominator of v. If v is a graph root node, and thus does not have an immediate dominator, *minEntry*$[v]$ receives a special value $\emptyset$. Similarly, *minExit*$[v]$ is initialized with the immediate post-dominator of v, or with $\emptyset$ if v is a leaf node.

The main step of the algorithm, shown in Algorithm 1 on page 123, iterates over all graph nodes. For each node v it builds a set of neighboring nodes that should be part of the same *super-structure* as v. Then, it attempts to find their entry and

exit nodes using an iterative approach. Line 7 of Algorithm 1 computes the initial set of nodes that are identified as belonging to the same *super-structure* as v. The pseudo-code for this sub-step is shown in Algorithm 2 on page 124. To understand if two nodes are part of the same *super-structure*, we use the property that all neighbors of an internal node of a *super-structure* are either internal nodes as well, or are one of its entry/exit nodes. Algorithm 2 iterates over all successors u of v and includes them in the initial set of nodes, unless u is the current exit node for v, or v is the current entry node for u. Separately, the program tests if v is an exit node for its entry node r, or an entry node for its exit node s. If v is an exit node for its entry node r, it means v post-dominates r while r dominates v. As a result v cannot belong to a *super-structure* whose entry node is r. Moreover, v and r are guaranteed to be part of the same *super-structure* and we include r into the set of nodes that are part of the same *super-structure* as v. Similarly we include v's exit node s into this initial set if v is an entry node for s.

Lines 8–18 of the main step execute an iterative algorithm that computes a closure of nodes that have common dominator and post-dominator nodes. We start by identifying the least common dominator and post-dominator of the nodes in the initial set. According to Lemma 6.3, all nodes on the paths to the common dominator and post-dominator nodes in the dominator and post-dominator trees respectively, are also part of the *super-structure*. Thus, in the process of computing the common dominator and post-dominator, we add new nodes to the set of nodes for which we try to find a common ancestor in the dominator trees. The problem reduces to a fixed point algorithm of finding the least common ancestor (LCA) of a set of nodes

Algorithm 1: Main step of computing *super-structures*.

```
input  : The dependence graph G
input  : The dominator and post-dominator trees predom and postdom
output: Disjoint-set data-structures preDS and postDS of size n
output: The minEntry and minExit arrays of nodes of length n

// Initialize the temporary and output data-structures
1  preDS ← DisjointSet(n)
2  postDS ← DisjointSet(n)
3  foreach node v ∈ G do
4      minEntry[v] ← predom.IDom[v]
5      minExit[v] ← postdom.IDom[v]

// Iterate over all nodes
6  foreach node v ∈ G do
7      compute domset and pdomset, the initial sets of nodes that belong to the
       same super-structure as v; initialize r and s as the entry and exit nodes of v
8      if domset.notEmpty() OR r ≠ ∅ OR s ≠ ∅ then
9          first ← true
10         repeat
11             if domset.notEmpty() OR first then
12                 predom.FindLCA(v,domset,pdomset,preDS,postDS,minEntry)
13                 domset.Clear()
14             if pdomset.notEmpty() OR first then
15                 postdom.FindLCA(v,pdomset,domset,postDS,preDS,minExit)
16                 pdomset.Clear()
17             first ← false
18         until domset.notEmpty()        // pdomset has just been cleared
```

in both the dominator and post-dominator trees, at each step the set of nodes being expanded with all the nodes that we traverse on the paths to the two LCAs. The least common ancestor of a set of nodes is computed by procedure **FindLCA** shown on page 125. The last four arguments to **FindLCA** are input/output parameters.

We use two disjoint-set data-structures [16], *preDS* and *postDS*, to union the nodes that are part of the same *super-structures*. We make use of two such data-structures to enable us to work independently on the dominator and post-dominator

Algorithm 2: Initialize *domset* and *pdomset* for node v.

```
1  domset ← ∅, pdomset ← ∅;                    // temporary sets of nodes
2  vPreId ← preDS.Find(v)
3  vPostId ← postDS.Find(v)
4  r ← minEntry[vPreId]
5  if r ≠ ∅ then
6      rPostId ← postDS.Find(r)
7      if rPostId ≠ vPostId AND minExit[rPostId] ≡ v then
8          domset.Insert(r)
9          pdomset.Insert(r)
10 s ← minExit[vPostId]
11 if s ≠ ∅ then
12     sPreId ← preDS.Find(s)
13     if sPreId ≠ vPreId AND minEntry[sPreId] ≡ v then
14         domset.Insert(s)
15         pdomset.Insert(s)
16 foreach successor node u of v do
17     uPreId ← preDS.Find(u)
18     if uPreId ≠ vPreId AND minEntry[uPreId] ≠ v AND s ≠ u then
19         domset.Insert(u)
20         pdomset.Insert(u)
```

trees. The two disjoint-set data structures are used by procedure **FindLCA** to union v with both the initial set of nodes identified as belonging to the same *super-structure* and the additional nodes encountered on the paths towards the common dominator and post-dominator nodes. We implemented routine **FindLCA** as a method of the *Dominator* class. Thus, it has access to all data structures that were initialized in the process of computing the dominance information, including the DFS indices of all nodes and a *vertex* array which performs the translation from a DFS index to the corresponding node. The **FindLCA** routine receives as input arguments the node v and a set of nodes, *thisset*, for which we want to find the least common ancestor

Procedure `FindLCA`(v,*thisset*,*otherset*,*thisDS*,*otherDS*,*minLCA*)

input : Array dfsIndex contains the DFS index of each node
input : Array vertex translates from DFS index to node
input : Array numKids specifies the number of children of each node
output: Array minLCA stores the updated dominance information

```
vId ← thisDS.Find(v)
pv ← minLCA [vId]
if pv ≠ ∅ then idx1 ← dfsIndex[pv]
else idx1 ← 0
foreach node u ∈ thisset do
    tempset ← ∅          // temporary set to store the traversed nodes
    tempset.Insert(u)                          // insert u into tempset
    uId ← thisDS.Find(u)
    pu ← minLCA [uId]
    if pu ≠ ∅ then idx2 ← dfsIndex[pu]
    else idx2 ← 0
    while idx1 ≠ idx2 do
        if idx1 < idx2 then
            newIdx ← idx2
            firstIdx ← false
        else
            newIdx ← idx1
            firstIdx ← true
        t ← vertex[newIdx]
        tempset.Insert(t)
        pu ← minLCA[thisDS.Find(t)]
        if pu ≠ ∅ then newIdx ← dfsIndex[pu]
        else newIdx ← 0
        if firstIdx then idx1 ← newIdx
        else idx2 ← newIdx
        otherset.Insert(t)                     // add t to otherset as well
    // LCA found; union v with all the nodes in tempset
    foreach node t ∈ tempset do
        tId ← thisDS.Find(t)
        if vId ≠ tId then
            thisDS.Union(vId, tId)
            vId ← thisDS.Find(v)
    minLCA [vId] ← vertex [idx1]
```

Procedure `FindLCA(`v,$thisset$,$otherset$,$thisDS$,$otherDS$,$minLCA$`)` (con't)

```
   /* idx1 is the DFS index of the found LCA. We want to make sure
      the LCA has at least two immediate children, which indicate it
      has more than one outgoing dependences.  Keep traversing the
      dominator tree upwards, and union all the traversed nodes    */
33 while idx1 ≠ 0 AND numKids[idx1] < 2 do
34     t ← vertex[idx1]
35     otherset.Insert(t)   // add t to otherset as well tId ← thisDS.Find(t)
36     pt ← minLCA[t]
37     if pt ≠ ∅ then idx1 ← dfsIndex[pt]
38     else idx1 ← 0
39     thisDS.Union(vId, tId)
40     vId ← thisDS.Find(v)
41     minLCA [vId] ← pt
```

(LCA). On lines 12–26 we compute the LCA of two nodes at a time using the DFS indexing information to accomplish this task efficiently. After each intermediate LCA is computed, on lines 27–31 we union v with the nodes that were encountered on the path to the intermediate LCA. Unioning the nodes has the effect of path compression on the dominator tree. In each successive step, we compute the ancestor of the previously found LCA and of a new node from *thisset*. We also add all intermediary nodes traversed on the paths to the LCA into a separate set of nodes, *otherset*, as seen on line 26. The set of nodes *otherset* will be used as input for finding the LCA in the other dominator tree.

Once the LCA is identified, **FindLCA** performs a final step, shown on lines 33–41, in which it ensures that the found LCA has more than one child in the dominator tree. If the LCA has only one child, we keep going up in the dominator tree until we find a dominator with at least two children. We perform this additional step to eliminate trivial *super-structures* that have only one distinct path through them. To

get the number of children of a node in the dominator tree, we use the array *numKids* that is computed when building the dominator tree.

We use the union by rank with path compression implementation of the disjoint-set data-structure, whose time complexity is $O(p\,\alpha(p,q))$ for p disjoint-set operations on q elements [16]. At each step, we union the nodes that are determined to be part of the same *super-structure*, and we use path compression on the dominator and post-dominator trees. As a result, each dominator tree edge is traversed at most two times when processing the nodes. Since the number of edges in the dominator tree/forest is at most $n-1$, the asymptotic complexity of the main step of the algorithm is $O(E\,\alpha(E,N))$, where N is the number of nodes and E is the number of edges in the dependence graph. At the end of this step, *minEntry* and *minExit* contain the updated entry and exit information for each node.

After the main step we use a refinement step, presented in Algorithm 5, which traverses each *super-structure* S, and tests the dominance relationship between its entry and exit nodes. Because a *super-structure* has one entry node and one exit node, at a high level it can be thought of as there being an edge from its entry node r to its exit node s. However, this connectivity might not be considered in the main step of the algorithm if all paths from r to s are part of *super-structures*. When we test all successors u of a node v in the sub-step presented in Algorithm 2, we do not include u in the initial set of nodes to be unioned with v if u is an exit node for v, or if v is an entry node for u. The purpose of this refinement step is to account for this connectivity in the graph where all *super-structures* were replaced by super-edges from the entry nodes to their corresponding exit nodes. Thus, if r dominates s and

Algorithm 5: The refinement step of computing *super-structures*.

```
   // Iterate over all structures
1  foreach super-structure S do
2      r ← entry(S)
3      s ← exit(S)
4      rDomS ← predom.Dominates(r,s)
5      sPdomR ← postdom.Dominates(s,r)
6      if (rDomS AND sPdomR) OR (NOT rDomS AND NOT sPdomR)
       then
7          domset.Insert(s)
8          pdomset.Insert(s)
9          repeat
10             if domset.notEmpty() then
11                 predom.FindLCA(r,domset,pdomset,preDS,postDS,minEntry)
12                 domset.Clear()
13             if pdomset.notEmpty() then
14                 postdom.FindLCA(r,pdomset,domset,postDS,preDS,minExit)
15                 pdomset.Clear()
16         until domset.notEmpty()        // pdomset has just been cleared
```

s post-dominates r, or if neither of the dominance relationships holds, then r and s must be part of the same *super-structure* and we union them using an iterative process similar to the one in the main step. However, if only r dominates s but s does not post-dominate r, or vice-versa, then r can be an entry node for s or s can be an exit node for r respectively, and we do not union them.

Note that the iterative step of the refinement algorithm is executed more than once only if additional nodes are unioned with r and s in the last call to the **FindLCA** routine. However, the total number of union operations possible in the main step and the refinement step combined is bounded above by the number of nodes in the graph. The asymptotic complexity of the conditional step at lines 4–6, is more difficult to compute. Note that the dominance relationship of two nodes can be computed in

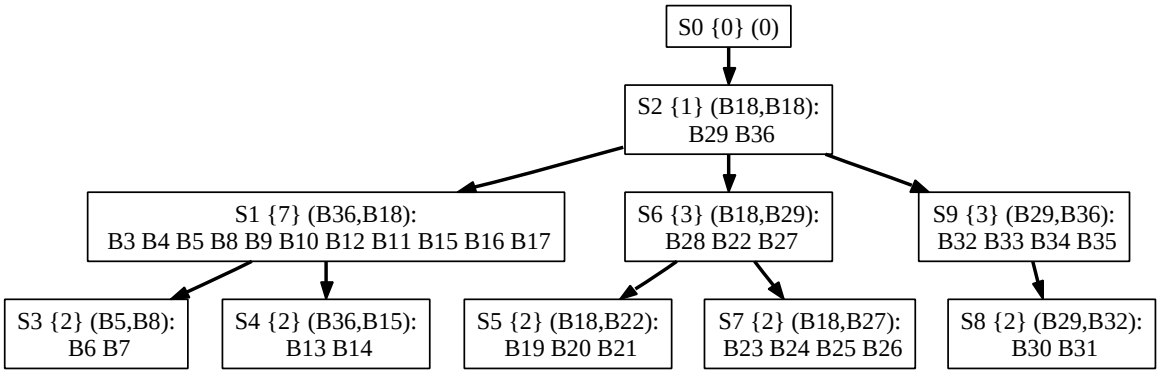


Figure 6.13: Hierarchy of super-structures for the dependence graph in Figure 6.7.

$O(h)$ where h is the height of the dominator tree. In the worst case, when the graph degenerates to a list, h is equal to $N - 1$. However, we cannot have any *super-structures* in that case. Remember that the final step in **FindLCA** ensures that only nodes that dominate multiple nodes directly can be entry nodes, and only nodes that post-dominate multiple nodes directly can be exit nodes. In general the dominator trees are very shallow and the cost of this step is very small; therefore, a loose upper bound for this step is $O(|S|\, N)$.

At this point we have our *super-structures*. However, some of them can be trivial, that is they contain only one path, even if the entry node dominates multiple nodes. Figure 6.13 presents the tree of *super-structures* computed for our sample dependence graph presented in Figure 6.7. Each node represents a *super-structure* where the first line specifies the entry and the exit nodes, and the number of top edges (edges from the entry node to an internal node). The second line of each rectangle specifies the internal nodes of the *super-structure*. The *super-structures* are organized as a tree in which edges represent the enclosing relationship and the root of the tree represents an imaginary *super-structure* that encompasses the entire graph. A final step is needed

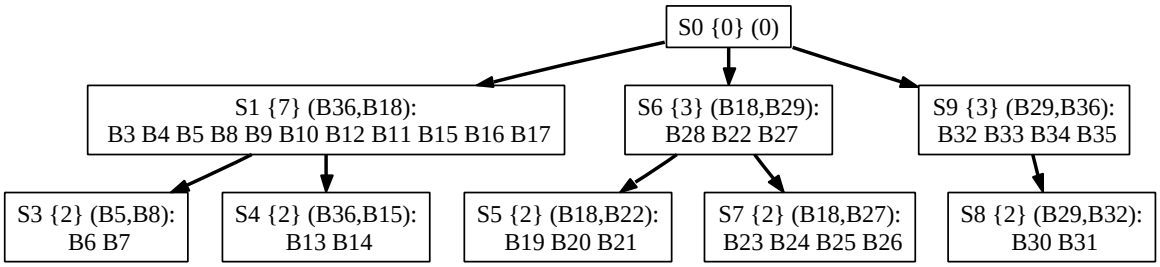


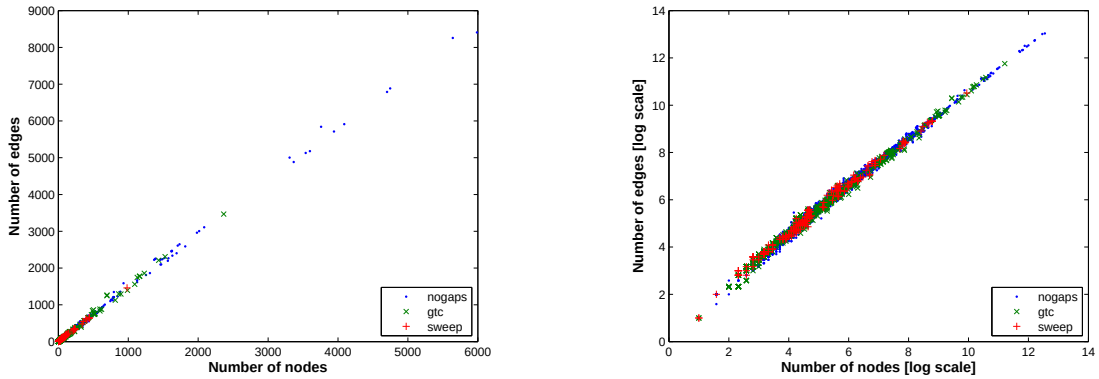
Figure 6.14: Final hierarchy of super-structures for the dependence graph in Figure 6.7.

to merge the trivial *super-structures* within their closest enclosing *super-structures* that have more than one path. This step has complexity $O(E)$ since all edges of the graph are traversed to test if they are top edges for a *super-structure* and the merging step is $O(N)$. Figure 6.14 presents the final hierarchy of *super-structures* for our sample dependence graph.

6.4 Empirical Analysis of the Scheduling Cost

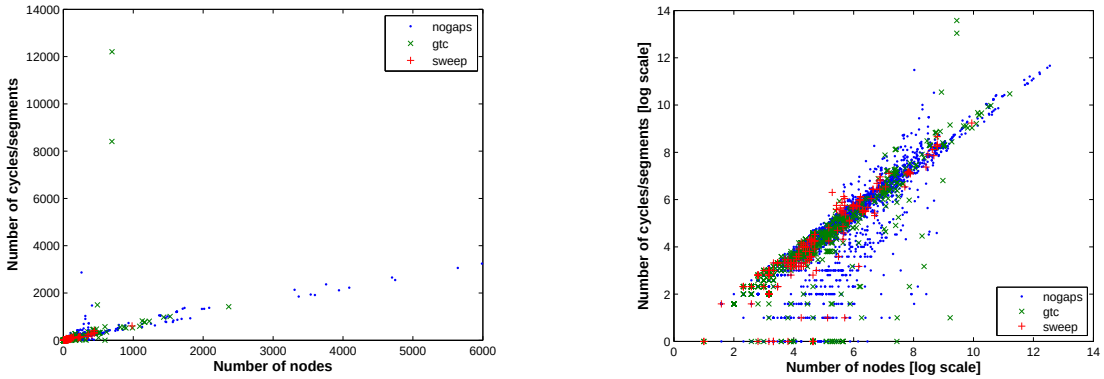
Previous sections provided some asymptotic bounds on the execution cost of the various steps of our instruction scheduler. In this section we present measured statistics from analyzing three full applications: the ASCI Sweep3D benchmark, the Gyrokinetic Toroidal Code (GTC), and Nogaps, a weather modeling code. We collected data about the number of nodes and edges in the dependence graphs corresponding to all executed program paths of the three applications, the number of dependence cycles in each of the graphs, and the execution cost of the various steps involved in computing cross-architecture predictions.

Figure 6.15 presents a scatter plot of the number of edges vs. the number of nodes for all program paths in the three applications. The left graph presents the data on



(a) (b)

Figure 6.15: The number of edges as a function of the number of nodes for all executed paths in Sweep3D, GTC and Nogaps; (a) linear scale; (b) logarithmic scale



(a) (b)

Figure 6.16: The number of recurrences as a function of the number of nodes for all executed paths in Sweep3D, GTC and Nogaps; (a) linear scale; (b) logarithmic scale

a linear scale, while the right graph uses a logarithmic scale for both axes. The data shows a strong correlation between the number of nodes and the number of edges in the dependence graphs for these applications, with about 1.47 edges for each node.

Figure 6.16(a) shows the number of recurrences as a function of the number of nodes, on both a linear scale (left) and a logarithmic scale (right). The data points are more spread-out than in the graphs presenting the number of edges. However, we

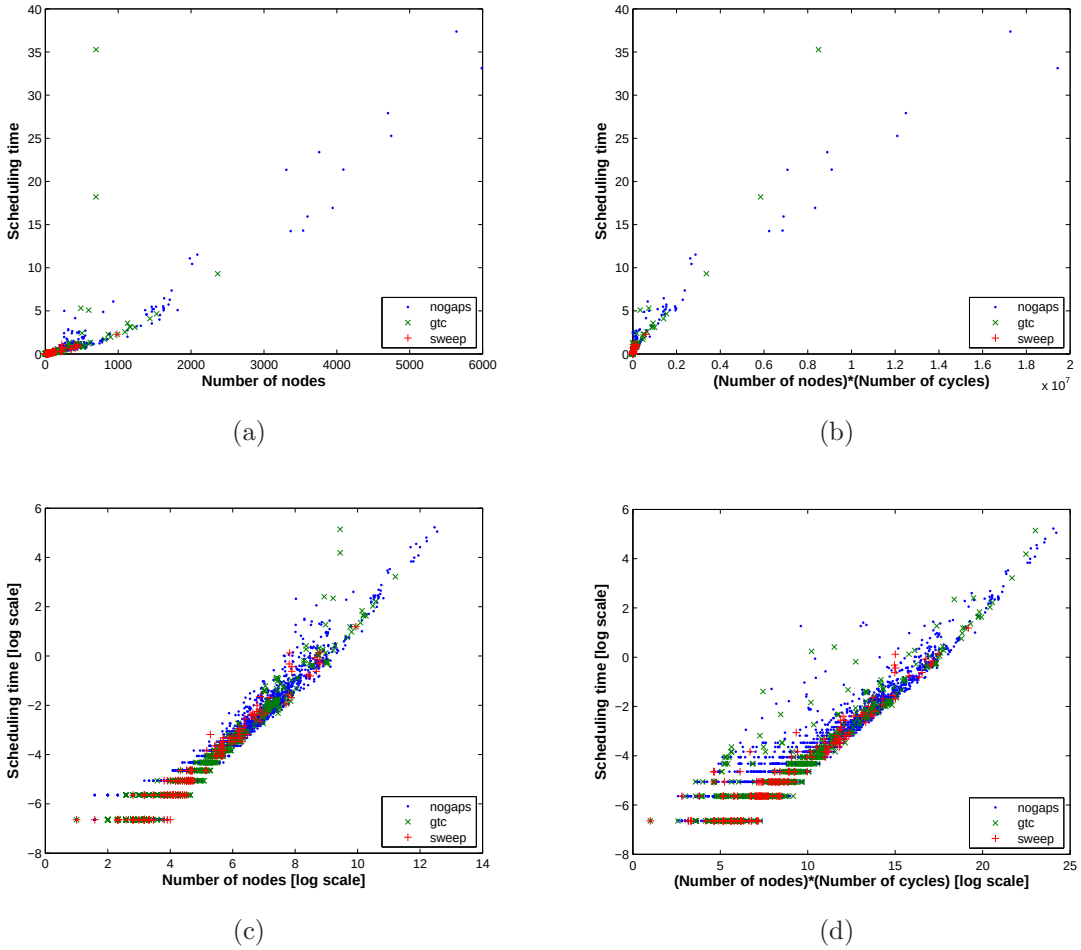


Figure 6.17: Scatter plot of scheduling cost vs. (a) the number of nodes; and (b) the number of nodes times the number of recurrences. Subfigures (c) and (d) show the same data on a logarithmic scale.

notice a concentration of points with a strong correlation between the number of nodes and the number of recurrences+*super-structure* paths. For these three applications there are roughly 0.63 dependence cycles + *super-structure* paths for every node. Note also that most of the analyzed program scopes have less than 500 instructions. The data points with a large number of nodes/edges/cycles correspond to outer level loops containing multiple inner loops and function calls.

Figure 6.17(a) presents the cost of instruction scheduling as a function of the number of nodes. This cost includes the execution of all steps from applying replacement rules on the input graph (see section 6.2.3), pruning the graph, computing the lower bounds on the schedule length, to finding the *super-structures* in the graph, computing the dependence cycles and computing the actual instruction schedule and schedule length. We notice that the schedule cost is super-linear. In a previous section we mentioned that the step of computing priority metrics for edges that are inserted into the edges priority queue is the most expensive operation, with an asymptotic complexity of $O(N\,C)$, where N is the number of nodes and C is the number of recurrences. For this reason, we also plotted the schedule time vs. $N \times C$ (see Figure 6.17(b)). This plot confirms our theoretical result, showing that the schedule time is $O(N\,C)$. Since C is a linear function of N, at least for these three applications, we could also say that empirically, the scheduling cost is $O(N^2)$.

Figures 6.17(c) and 6.17(d) present the same data on a logarithmic scale, such that the values corresponding to the smaller dependence graphs are easier to see. The time plateaus seen at the bottom of these two graphs are an artifact of the relatively large granularity timer used for these measurements.

6.5 Cross-Architecture Prediction Results

In this section, we apply our methodology to predict the cross-architecture instruction schedule time and the overall execution time for several programs. We use six benchmarks in this study, including: the ASCI Sweep3D benchmark [19], BT and SP from NPB 2.3-serial [4] , and BT, LU and SP from NPB 3.0.

We compare our predictions against measurements performed using hardware performance counters on an Origin 2000 system and an Itanium2 based machine. The Itanium2 system used for validation measurements has a 900MHz CPU, a 16KB level 1 data cache (L1D), a 256KB 8-way set-associative unified level 2 cache and a 1.5MB 6-way set-associative unified level 3 cache. The level 1 cache has a line size of 64 bytes, while both level 2 and 3 caches have a line size of 128 bytes. The Origin 2000 system used for validation measurements is a NUMA machine with 16 R12000 CPUs at 300MHz and 10GB of RAM. Each CPU has a 2-way set-associative 32KB L1 data cache, and a TLB with 64 entries, where each TLB entry maps two consecutive 16KB pages. Each pair of CPUs shares a large 8MB 2-way set-associative unified L2 cache. The L1 cache uses a line size of 32 bytes, and the L2 cache has a line size of 128 bytes.

To compute the predictions, we compiled the benchmark applications on a Sun UltraSPARC-II system using the Sun WorkShop 6 update 2 FORTRAN 77 5.3 compiler, and the optimization flags: *-xarch=v8plus -xO4 -depend -dalign -xtypemap=real:64*. Measurements on the Origin 2000 system were performed on binaries compiled with the SGI Fortran compiler Version 7.3.1.3m and the optimization flags: *-O3 -r10000 -64 -LNO:opt=0*, while on the Itanium2 system we compiled the binaries using the

Intel Fortran Itanium Compiler 8.0, and the optimization flags: *-O2 -tpp2 -fno-alias*. We used the highest optimization level but we disabled the high level loop nest optimizations because the sets of loop nest transformations implemented in the Sun, SGI and Intel compilers are different. Loop nest optimizations are tailored to each target architecture and they interfere with our ability to make cross-architecture and cross-compiler predictions.

To estimate execution time for an application on a target architecture, we must consider both the cost of computation and the exposed memory hierarchy latency for the application on the specified machine. How well the instruction-level parallelism in the most frequently executed loops of an application matches the number and the type of execution units available on a target architecture determines the compactness of the instruction schedule and thus the computation cost.

To characterize the execution frequency of basic blocks and loops in a program for arbitrary problem sizes, we construct models of basic block execution frequency. For each of the six applications considered in this study, we used our tool to collect edge counter execution histograms for a range of problem sizes between 20 and 50. Next, we derived models of the execution frequency for each edge counter and evaluated the models at each problem size of interest.

6.5.1 Predictions for Itanium2

We used our scheduler, initialized with a description of the Itanium2 architecture, to estimate execution time for each application when all memory references hit in the appropriate cache closest to the CPU. For the Itanium2 system, we considered

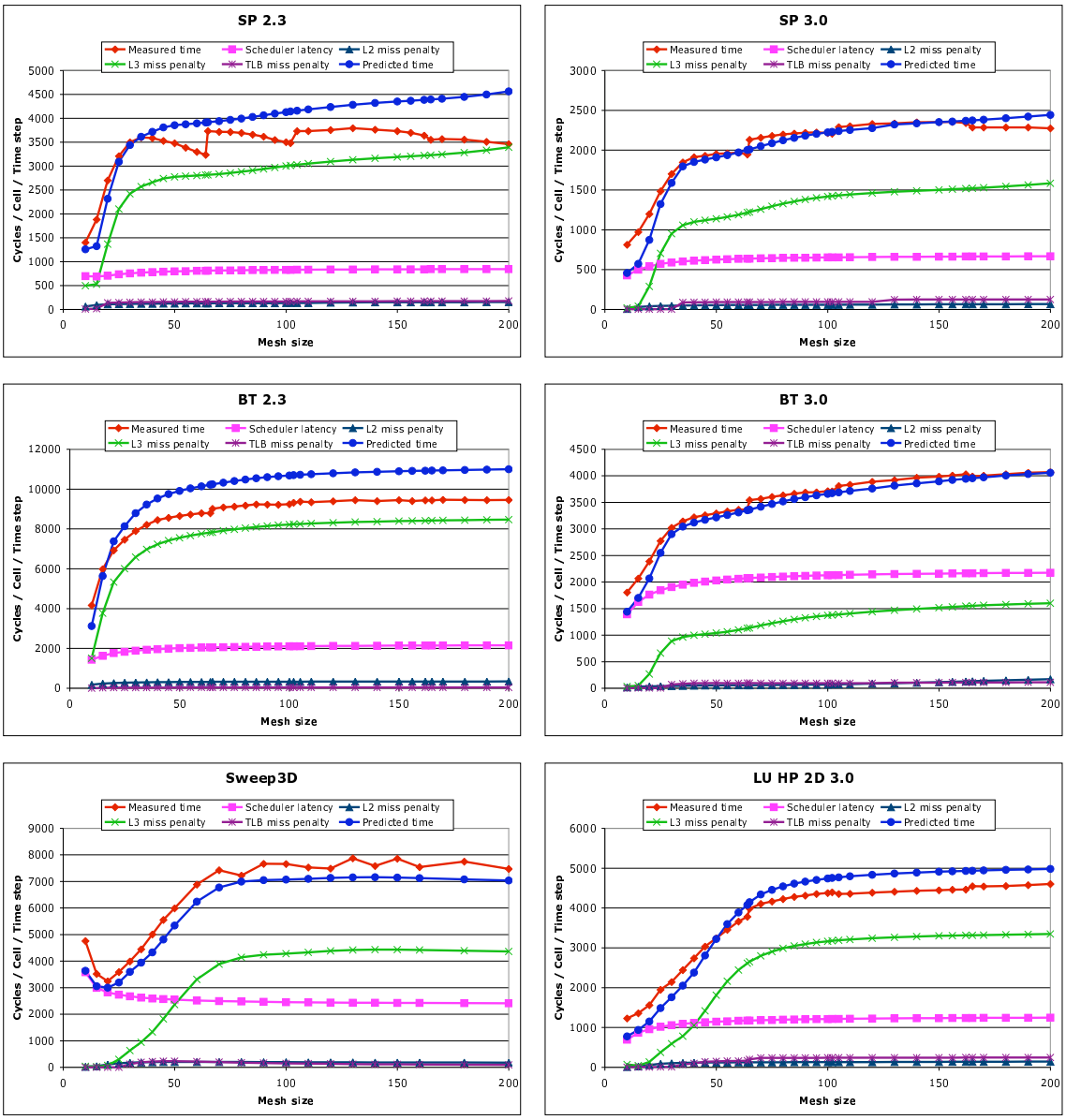


Figure 6.18: Itanium2 execution time predictions for 5 NAS benchmarks and Sweep3D. Each graph presents: (1) the measured time, (2) the instruction execution time predicted by the scheduler when all memory accesses are cache hits, (3) the predicted L2 miss penalty, (4) the predicted L3 miss penalty, (5) the predicted TLB penalty, and (6) the predicted execution time which includes both the instruction execution cost and the exposed memory hierarchy latency.

that all integer references hit in L1 cache and all floating point loads and stores hit in L2 cache. On Itanium, floating-point values bypass the L1D cache. The

"Scheduler latency" curves in Figure 6.18 represent the predicted execution times when all memory accesses are cache hits. All graphs present the execution time per time step normalized to the number of mesh cells. The x axis represents the mesh size, and the y axis represents the number of CPU cycles per cell.

How to translate the cache miss counts implied by our models of memory reuse distance into an expected latency for each memory reference, on an arbitrary architecture, is still an open problem. On an out-of-order superscalar machine, the processor can execute instructions in an order different than that specified by the compiler. On such machines, the hardware logic can hide some of the cache miss penalty; its ability to do so is limited by the size of the instruction window and by the number of instructions without unresolved dependences available in the window at any given time. On an in-order or VLIW machine[3], the processor always executes the instructions in the order determined by the compiler. In the case of a cache miss, the hardware continues to issue instructions until it reaches an instruction with unresolved dependences, such as an instruction that needs the data returned by a reference that missed in cache. At this point, the execution pipeline stalls until the dependences are resolved.

It is the compiler's job to order instructions in a way that minimizes execution time. Compilers can rearrange the instructions in a loop to group together loads to data that cannot be in the same cache line such that if more than one access misses in the cache, the latency of fetching the data from memory for every cache miss after the first one, is partially hidden by the latency associated with the first miss [56].

[3]Most modern machines have non-blocking caches, therefore we consider only such cases. Predictions for machines with blocking caches are much easier.

Such an optimization is limited by the number of parallel, large-stride loads available in a loop and by the maximum number of outstanding memory references allowed in the system. On Itanium2, 6 writes to memory and 16 loads to L3 or memory can be outstanding at any given time. With such a large parallelism in the architecture, the instruction schedule in the most frequently executed loops is typically the factor that determines the memory system parallelism realized.

We define the memory hierarchy penalty for an application as the sum of the penalties incurred by each cache miss reference. The Itanium2 machine used to validate the predictions has the following memory access latencies for a floating point reference: minimum access time to L2 cache is 6 cycles, minimum access time to L3 is 14 cycles, and the access time to main memory is about 116 cycles. Therefore, the penalty for each L2 cache miss is 8 cycles, and the penalty for each L3 cache miss is 102 cycles. For TLB misses we used a penalty of 23 cycles.

We compiled the six applications under study with prefetching disabled and we determined empirically that there was very little overlap between memory accesses for these applications. As a result, to compute our predictions we assumed the full penalty of an L3 miss and 50% of the penalty of an L2 miss are exposed. This empirical level of memory parallelism is an approximative value we use to predict the memory hierarchy penalty from cache miss data for all the applications in this study. The actual exposed memory penalty must be determined for each loop of a program using static analysis.

In addition to the instruction execution cost computed by the scheduler, Figure 6.18 presents the actual execution time measured with hardware counters, the

predicted penalty time due to L2, L3 and TLB misses, and the end-to-end predicted execution time which includes both the time computed by the scheduler and the predicted memory hierarchy penalty. We notice that BT 3.0, which was optimized to improve cache reuse since version 2.3, is the only application whose execution time is dominated by the instruction scheduling cost. For the other applications, the L3 miss penalty is the dominant factor. The predictions of L3 cache misses, shown in Appendix B, are fairly accurate, hence the profiles for the curves of measured and predicted execution time are very similar. The errors observed for some of the applications are due to either a less accurate prediction of L3 miss counts (application SP 2.3), or our assumption that the full L3 penalty is exposed (application BT 2.3).

6.5.2 Predictions for MIPS R12000

Similarly, we used our scheduler to estimate the execution time of each application on a MIPS R12000 processor. On the R12000 we used the latency of the L1 cache for all memory references. Figure 6.19 presents the predicted instruction execution time when all memory accesses are cache hits, the memory penalty at each memory level, and the total execution time predicted by combining the schedule latency and the L1, L2 and TLB penalties. For validation, all graphs include the actual execution time measured using hardware performance counters. As before, all graphs present values normalized to the number of cells and time steps.

For the R12000-based Origin system, we used the following memory access latencies for a floating point reference: minimum access time to L1 cache is 3 cycles, minimum access time to L2 is 11 cycles, the access time to main memory ranges from

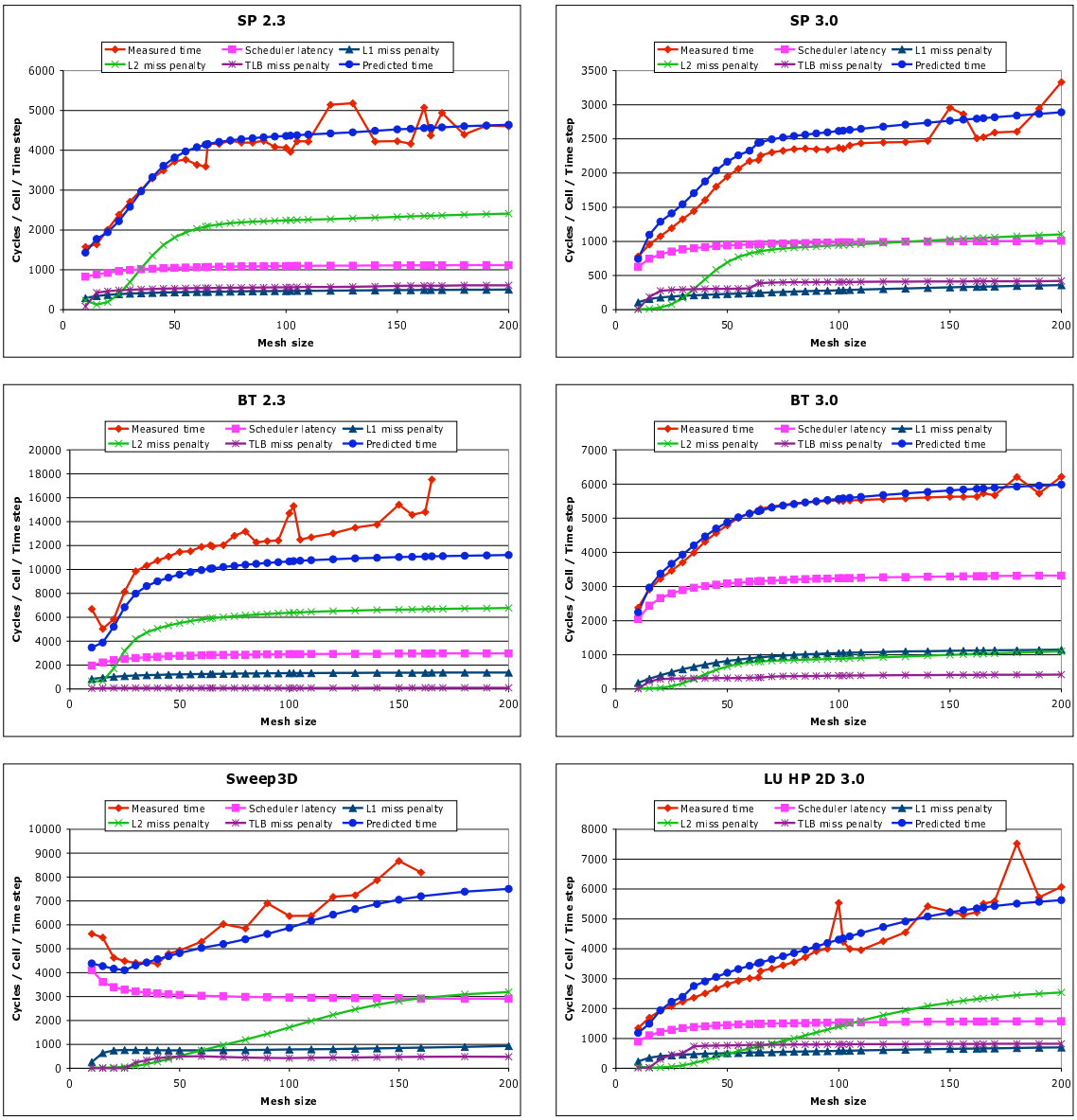


Figure 6.19: MIPS R12000 execution time predictions for 5 NAS benchmarks and Sweep3D. Each graph presents: (1) the measured time, (2) the instruction execution time predicted by the scheduler when all memory accesses are cache hits, (3) the predicted L1 miss penalty, (4) the predicted L2 miss penalty, (5) the predicted TLB penalty, and (6) the predicted execution time which includes both the instruction execution cost and the exposed memory hierarchy latency.

92 cycles (restart latency) to 116 cycles (back-to-back latency) [25, 54], and the TLB miss penalty is 78 cycles.

We used the same assumptions about the fraction of the memory penalty exposed to the applications as we did for our Itanium2 predictions. The figures show that with these assumptions, the execution time predictions match well the measured values for four of the applications. The exceptions are BT 2.3 and Sweep3D for which we underpredict by about 20%. The spikes in measured execution time that can be observed for some problem sizes, are a result of competing with other running jobs for access to memory.

For Sweep3D, part of the execution time prediction error is due to the variation in the number of L2 conflict misses predicted by our probabilistic model and the actual number of misses on the R12000 (see Appendix C). Figure 6.20 presents the L2 miss penalty and the predicted execution time (the **A** curves in Figure 6.20) when we consider the measured number of L2 misses instead of the predicted counts in our execution time formula. With the measured number of L2 misses, the predicted and measured execution time curves have a similar shape, but we underpredict the execution cost. The fact that the gap between the predicted and the measured execution time increases with problem size, combined with the observation that the scheduler latency, L1 and TLB penalties are constant or slightly decreasing when problem size grows (see Figure 6.19(e)), suggests that the difference is due to a larger memory penalty incurred by L2 misses, the only component of the predicted execution time that increases with problem size.

Curves **B** in Figure 6.20 represent the L2 miss penalty and the predicted execution time when we use a 135 cycle memory latency in our formula for predicted execution time, while all the other parameters are left unchanged. The restart memory latency

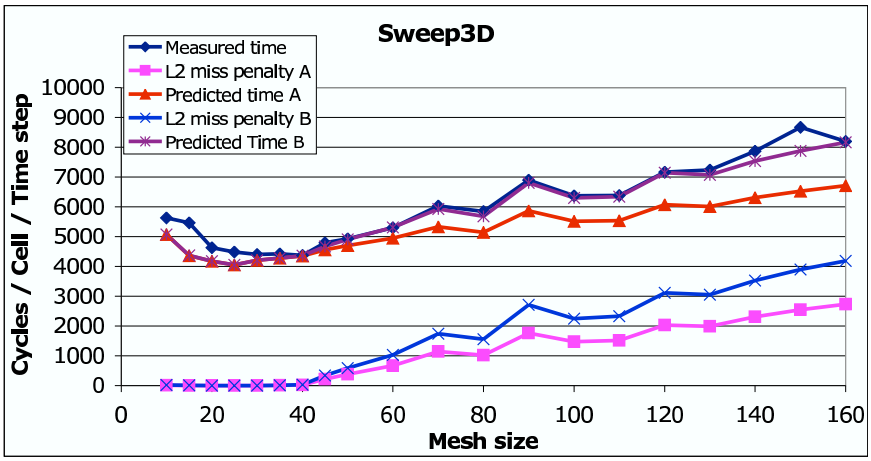


Figure 6.20: L2 miss penalty and predicted execution time using (A) the measured L2 miss counts, plus (B) assuming a 135 cycle memory latency.

of 92 cycles presented at the beginning of this section is micro-benchmarked for read operations when all cache lines are clean. In a real application, some of the cache lines will be dirty. If a cache miss causes a dirty line to be replaced, the dirty line must be written back to memory before the new line can be fetched; this may result in a higher memory latency.

We acknowledge that the formula for predicted execution time is ad-hoc, and that we have to understand better how to automatically translate our accurate predictions of cache miss counts into an expected memory hierarchy delay. Empirically, we could determine a more accurate value for the fraction of the memory penalty that is exposed to each application using an approach similar to the one presented in [38]. However, we presented this last step to validate the ideas that are at the base of our performance prediction method. The accurate predictions accross a large set of problem sizes on both an Itanium2 machine and an Origin 2000 system validate our general approach for computing cross-architecture predictions.

Chapter 7

Understanding Performance Bottlenecks

Over the past two decades, advances in semiconductor technologies and micro-architecture design have produced a dramatic increase in the peak performance of microprocessors. This speed increase has come at the cost of a substantial increase in architecture complexity. As a result, it has become increasingly difficult for applications to sustain significant fractions of peak performance. This is due in part to compilers and application developers not being able to harness the potential of the architectures and in part due to an imbalance between the resources offered by super-scalar architectures and the actual needs of applications.

In this chapter, we describe extensions to our toolkit's analysis capabilities to leverage its detailed insight into performance of applications in order to pinpoint performance bottlenecks and determine the potential for improvement from correcting them. We focus on performance bottlenecks due to insufficient instruction-level parallelism, inefficiencies due to a mismatch between the number and type of resources provided on the target machine vs. the type of resources required by the most frequently executed application loops, or inefficiencies due to poor data locality.

7.1 Understanding Instruction Schedule Inefficiencies

In chapter 6, we have described the steps of a fairly standard modulo-scheduling algorithm. Our implementation has its strengths and weaknesses. A strength of the scheduler is its configurability. The instruction types and the target machine model are both user definable. A weakness of the scheduler is that it ignores some architectural details, such as register pressure or branch miss prediction. It is adequate for our purposes because our goals are to predict a lower bound (though not too loose) on achievable performance and to understand what sections of code may benefit from transformations or from additional execution units. A modulo-scheduler offers us this insight because we can attribute each clock cycle of the predicted schedule time to a particular cause.

When we compute the lower bounds on schedule length (see sections 6.3.1 and 6.3.2), if the lower bound due to recurrences, LB_{Dep}, is greater or equal than the lower bound due to resource contention, LB_{Res}, we consider that LB_{Dep} clock cycles of each loop iteration are due to *application dependences*. If, on the other hand, the bound due to resource contention is greater, we know also which unit was determined to have the highest contention factor[1]. In case multiple units have the same contention factor, the tie is resolved by the order in which units are defined in the machine description file. In such cases we say LB_{Res} clock cycles of each iteration are due to a *resource bottleneck*, and we refine this cost further by the type of unit that is causing the most contention.

[1]Because the machine model may contain optional restriction rules between units, if one of the rules is determined to cause the most contention, then the cost is associated with that rule.

In the next step of the scheduling algorithm, we try to find an actual feasible schedule length that takes into account both instruction dependences and resource contention. Every time we increase the schedule length, we determine what resource type, either execution unit or restriction rule, prevented the scheduler from continuing. The way this scheduling step is implemented, the algorithm does not try to schedule an instruction in a clock cycle that breaks dependences. Therefore, the scheduler fails when there is no execution template that does not conflict with resources already allocated or with one of the optional restriction rules for any of the valid issue clock cycles. This additional scheduling cost for each iteration is counted separately as *scheduling extra cost*. Again, we refine this cost further by the unit type that was the source of contention. As is the case with LB_{Res}, there may be multiple execution units that conflict with an instruction's execution template(s), more so as multiple clock cycles and multiple templates may be involved. The unit that is selected is the first unit found to conflict with the last probed execution template, in the last valid issue cycle.

Optionally, the scheduler can be invoked with cache miss count predictions at loop level, predicted from memory reuse distance models (see section 5.2), for each level of the memory hierarchy. If miss count predictions are available, the scheduler computes also a *memory penalty time* for each level of the memory hierarchy, as the product between the number of accesses that miss into a memory level and the penalty of a miss taken from the machine description file. This memory penalty time does not account for possible overlap between memory accesses and computation, or between multiple outstanding memory requests. We found that in practice there is

little memory parallelism when applications are not compiled with prefetching and our estimates are quite accurate (see section 6.5), but this is definitely not true when prefetching is used.

Using miss count predictions and the computation time predicted by the scheduler, we compute the bandwidth consumed at each memory level assuming ideal prefetching (no miss penalty time). This metric provides an estimate for the minimum bandwidth needed to avoid starving the execution units if good data prefetching can be achieved. Alternatively, this metric can be used to determine if an application runs inefficiently because of lack of bandwidth. This metric is similar to the loop balance metric introduced in [12], except we consider the balance between memory bandwidth and the actual instruction schedule cost of a loop, not just the floating-point peak performance. We can compare this value directly with either the peak or the sustainable bandwidth of a machine to determine if bandwidth is a limiting factor.

We compute the execution costs in a bottom-up fashion, from the innermost loops to routines and to the entire program, aggregating costs for each of the categories described above. At the end of this process, we have not only a prediction of instruction schedule time for the entire program, each routine and each loop, but also the attribution of execution cost to the factors that contribute to that cost:

- **Execution time**
  - **memory penalty**
    - **penalty at each level**
  - **bandwidth delays**
    - **bandwidth delays at each level**
  - **schedule time**
    - **application dependence time**
    - **bottleneck on resources time**
      - **bottleneck per each resource**
    - **scheduling extra time**
      - **extra time due to each resource**

Performance monitoring hardware on most modern architectures can provide insight into resource utilization on current platforms. Our performance tool can provide such insight for future architectures, at a much lower cost than cycle accurate simulators. However, we realized that what is lacking in current performance tools, is a way to point the application developer, or an automatic tuning system for that matter, to those sections of code that can benefit the most from program transformations or from additional machine resources. Just because a loop accounts for a significant fraction of execution time, it may not be wasting any issue slots. It may actually have good instruction and memory balance with respect to the target architecture with little room for improvement. We need to focus on loops that are frequently executed but also use resources inefficiently.

7.1.1 Assessing the Potential for Improving the Instruction Schedule

One of the steps in the scheduling algorithm is the computation of the minimum initiation interval. For this, two lower bounds on the schedule length are computed.

LB_{Res} represents the lower bound due to resource contention, and is computed assuming there are no schedule dependences between instruction. Let $\mathcal{C}$ be the computation cost computed by the scheduler when both instruction dependences and resource contention are considered. We define the metric *maximum gain achievable from increased ILP* as $MaxGain_{ILP} = \mathcal{C} - LB_{Res}$. This metric represents exactly what its name implies. If total computation cost is $\mathcal{C}$, and the cost achievable using the same set of machine resources if we removed all data dependences in loops is LB_{Res}, then the maximum we can expect to gain from transforming the code to increase ILP, is $\mathcal{C} - LB_{Res}$. If the code performance is limited by the number and type of machine resources, that is if $\mathcal{C} = LB_{Res}$, there is nothing that can be gained from transforming the code, unless we rewrite the code using different instructions that require different resources.

LB_{Dep} represents the lower bound due to dependence cycles, and is computed assuming an unlimited number of machine resources. We define the metric *maximum gain achievable from additional resources* as $MaxGain_{Res} = \mathcal{C} - LB_{Dep}$. The name of this metric is self explanatory. No matter how many execution units we add to a machine, the execution cost of the code cannot be lower than LB_{Dep} unless we also apply code transformations. For loops without recurrences, LB_{Dep} of N iterations is equal to the execution cost of one iteration from start to finish, independent of N. With an unlimited number of machine resources, and with no carried dependences, all iterations can be executed in parallel in the time taken by a single iteration. However, we do not apply the same idea to outer loops or loops with function calls. As we explained at the start of chapter 6, inner loops and function calls act as fence

instructions, therefore they create at least a recurrence on themselves.

Each of these two metrics provides an estimate of the performance increase possible by modifying only one variable of the equation in isolation. At an extreme, if we removed all dependences and we assumed an infinite number of resources, we could execute a program in one cycle. However, we consider such a metric too unrealistic to be of any use in practice. As with other performance data we compute, we aggregate these two metrics in a bottom-up fashion up to the entire program level.

7.2 Understanding the Unfulfilled Memory Reuse Potential

In the previous section, we focused on understanding execution inefficiencies due to instruction schedule dependences, contention on execution units, or inadequate memory bandwidth relative to the machine issue width. The potential for improving the performance of data-intensive scientific programs by enhancing data reuse in cache is even more substantial because CPUs are significantly faster than memory. For data intensive applications, it is widely accepted that memory latency and bandwidth are the factors that most limit node performance on microprocessor-based systems.

Traditional performance tools typically collect or simulate cache miss counts and rates and attribute them at the function level. While such information identifies the program scopes that suffer from poor data locality, it is often insufficient to diagnose the causes for poor data locality and identify what program transformations would improve memory hierarchy utilization.

To understand why a particular loop experiences a large number of cache misses, it

helps to think of a non-compulsory cache miss as reuse of data that has been accessed too far in the past to still be in cache. This is in fact the idea behind memory reuse distance. It is an architecture independent metric that tells us the number of distinct memory blocks accessed by a program between pairs of accesses to the same block.

To understand if a memory access is a hit or miss in a fully-associative cache using LRU replacement, one can simply compare the distance to its previous use with the size of the cache. For set-associative caches, we have described in section 5.3.2 a reuse distance based probabilistic model that yields accurate predictions in practice [42].

However, up to this point we used only part of the information that we can obtain through memory reuse simulation. In particular, each data reuse can be thought of as an arc from one access to a block of data to the next access to that block. Associating reuse distance data with only one end of a reuse arc fails to capture the correlation between references that access the same data.

7.2.1 Identifying Reuse Patterns With Poor Data Locality

We already need to store some bits of information about each memory block that is accessed when we collect the memory reuse distance data (see section 4.2). By extending the information for a block to include also the identity of the most recent access, we can associate a reuse distance with a (*source*, *destination*) pair of scopes where the two endpoints of the reuse arc reside.

This approach enables us to collect reuse distance histograms separately for each pair of scopes that reuse the same data. Once reuse distance histograms are translated into cache miss predictions for a target cache architecture, this approach enables us

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

(a)

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

(b)

Figure 7.1: (a) Example of data reuse carried by an outer loop; (b) transformed example using loop interchange.

to understand not only where we experience a large fraction of cache misses, but also where that data has been previously accessed before it was evicted from cache. If we can transform the program to bring the two accesses closer, for example by fusing their source and destination loops, we may be able to shorten the reuse distance so that the data can be reused before it is evicted from cache.

We found that in many cases a single scope was both the source and the destination of a reuse arc. While this provides the insight that data is accessed repeatedly in the same loop without being touched in a different program scope, we found such information insufficient to understand how to correct the problem. What was missing, was a way to tell which outer loop was carrying the reuse, or in other words, which loop was causing the program to access the same data again on different iterations. If we know which loop is causing the reuse and if the distance of that reuse is too large for our cache size, then it may be possible to shorten the reuse distance by either interchanging the loop carrying the reuse inwards, or by blocking the loop inside it and moving the resulting loop that iterates over blocks, outside the loop carrying the reuse.

Loop interchange and blocking are well studied compiler transformations. A more

thorough discussion of these transformations can be found in [2]. Figure 7.1(a) presents a simple loop nest written in Fortran. Although Fortran stores arrays in column major order, the inner loop here iterates over rows. There is no reuse carried by the `J` loop, since each element of a row is in a different cache line. However, for non-unit size cache lines, there is spatial reuse carried by the outer `I` loop. By interchanging the loops as shown in Figure 7.1(b), we move the loop carrying spatial reuse inwards, which reduces the reuse distance for the accesses.

While it is reasonably easy to understand reuse patterns for simple loop nests, for complex applications understanding reuse is a daunting task. To capture the carrying scope of a reuse automatically, we extended our reuse distance data collection infrastructure, which was described in section 4.2, in several ways.

- We add instrumentation to monitor entry and exit of routines and loops. For loops, we instrument only loop entry and exit edges in the control flow so that instrumentation code is not executed on every iteration.

- We maintain a logical *access clock* that is incremented at each memory access.

- We maintain a dynamic stack of scopes in the shared library. When a scope is entered, we push a record containing the scope id and the value of the access clock onto the stack. On exit, we pop the entry off the scope stack. The stack stores the active routines and loops, in the order in which they were entered.

- On a memory access, in addition to the steps presented in section 4.2, we traverse the dynamic stack of scopes starting from the top, looking for $\mathcal{S}$—the

shallowest entry whose access clock is less than the access clock value associated with the previous access to current memory block. Because the access clock is incremented on each memory access, $\mathcal{S}$ is the most recent active scope that was entered before our previous access to current memory block. $\mathcal{S}$ is the least common ancestor in the dynamic scope stack for both ends of the reuse arc and we say it is the carrying scope of the reuse.

- For a reference, we collect separate histograms of reuse distances for each combination of (*source scope*, *carrying scope*) of the reuse arcs for which the reference is the sink.

- The dynamic scope stack enables us to collect footprint information as well. On a scope exit event, we use the logical time stamp stored in the scope's stack entry and the balanced binary tree of memory blocks, to compute the number of distinct memory blocks accessed since we entered that scope. For each scope we collect a histogram of distinct footprint values observed at run-time.

These extensions increase the resolution at which we collect memory reuse distance data. For one reference, or group of related references, we store multiple reuse histograms, one for each distinct combination of source scope and carrying scope of the reuse arcs. While this may seem to cause a large increase in the amount of data collected, in practice, the space increase is reasonable and completely worth it for the additional insight it provides. First, during execution applications access data in some well defined patterns. A load or store instruction is associated with a program variable that is accessed in a finite number of scopes that are executed in a pre-determined

order. Thus, there is not an explosion in the number of histograms collected for each reference. Second, reuse distances seen by an instruction at run-time vary depending on the source and carrying scopes of the reuse arcs. The effect is that while in our initial implementation we had fewer histograms, they had a greater number of bins to capture the different distance values encountered. In the new version, we have more but smaller histograms.

The new data is still completely reuse distance based. It differs just by the granularity at which it is collected. Therefore, the MRD modeling algorithm described in section 5.2 can be directly applied to model the scaling of individual reuse patterns, and predict their behavior for different inputs. In addition, the fact that the different patterns are collected and modeled separately, has the secondary positive effect of increasing the accuracy of the scalable models, especially for regular applications. For irregular or adaptive applications, aggregating the data may result in more accurate models in some cases.

Our new data enables us to compute cache miss predictions for an architecture separately for each reuse pattern. Thus, when we investigate performance bottlenecks due to poor data locality, we can highlight the principal memory reuse patterns that contribute to cache misses and suggest a set of possible transformations that would improve reuse. Not only does this information provide insight into the transformations that might improve a particular reuse pattern, but it also can pinpoint cache misses that are caused by reuse patterns intrinsic to an application, such as reuse of data across different time steps of an algorithm or reuse across function calls, which would require global code transformations to improve.

We compute several metrics based on our memory reuse analysis. For each scope, we compute traditional cache miss information; we use this to identify loops responsible for a high fraction of cache misses. However, we also associate cache miss counts with the scope that accessed data last before it was evicted, with the scope that is carrying these long data reuses, and a combination of these two factors. To guide tuning, we also compute the number of cache misses carried by each scope. We break down carried miss counts by the source or/and destination scopes of the reuse. These final metrics pinpoint opportunities for loop fusion and provide insight into reuse patterns that are difficult or impossible to eliminate, such as reuse across time steps or function calls. To focus tuning efforts effectively, it is important to know which cache misses can be potentially eliminated and which cannot; this helps focus tuning on cases that can provide a big payoff relative to the tuning effort. In section 8, we describe how we use this information to guide the tuning of two scientific applications.

7.2.2 Understanding Fragmentation in Cache Lines

The previous section described techniques for identifying opportunities to improve memory hierarchy utilization by shortening temporal and spatial distance. This section describes a strategy for diagnosing poor spatial locality caused by data layout. Caches are organized as blocks (lines) that typically contain multiple words. The benefit of using non-unit size cache lines is that when any word of a block is accessed, the whole block is loaded into the cache and further accesses to any word in the block will hit in cache until the block is evicted. Once a block has been fetched into cache, having accesses to other words in the block hit in cache is called spatial reuse. To

```
DO I = 1, N, 4
   DO J = 1, M
      A(I+1,J,K) = A(I,J,K) + B(I,J) - B(I+1,J)
      A(I+3,J,L) = A(I+2,J,L) + B(I+2,J) - B(I+3,J)
   ENDDO
ENDDO
```

Figure 7.2: Cache line fragmentation example.

exploit spatial reuse, we need to pack data that is accessed together into the same block.

We call the fraction of data in a memory block that is not accessed the *fragmentation factor*. We compute fragmentation factors for each array reference and each loop nest in the program. To identify where fragmentation occurs, we use static analysis. In section 5.2.2, we explain that we aggregate related references to improve the accuracy of the modeling step. At instrumentation time, we group together references of a loop that access data with the same name and the same symbolic stride, and we call them *related references*. To analyze the fragmentation of cache lines, we work on groups of related references and use the static symbolic formulas described in section 3.1, to compute reuse groups and the *fragmentation factor* of each reuse group. Computing the fragmentation factor for each group of references[2], consists of a three step process.

Step 1. Find the enclosing loop for which this group of references experiences the smallest non-zero constant stride. When the reference group is enclosed in a loop nest, traverse the loops from the inside out, and terminate the search if a loop is encountered

[2]Note that all references in a group have equal strides with respect to all enclosing loops. It suffices to consider the strides of only one reference in the group during analysis.

for which the references have an irregular stride. If a loop with a constant non-zero stride is not present, then we do not compute any *fragmentation factor* for that group of references. Otherwise, let s be the smallest constant stride that we find and go to **step 2**.

For the Fortran loop shown in Figure 7.2, the arrays are in column-major order, all four accesses to `A` are part of a single group of related references, and all four accesses to `B` are part of a second group of related references. For both groups, the loop with the smallest non-zero constant stride is the outer loop `I`, and the stride is 32 if we assume that the elements of the two arrays are double precision floating point values.

Step 2. Split a group of related references into reuse groups based on their *first location* symbolic formulas. Let F_1 and F_2 be the formulas describing the *first location* accessed by two references of a group. As computed in **step 1**, their smallest non-zero constant stride is s. If the two *first location* formulas differ only by a constant value, we compute how many iterations it would take for one formula to access a location within s bytes of the *first location* accessed by the other formula. If the necessary number of iterations is less than the average number of iterations executed by that loop (identified using data from the dynamic analysis), then the two references are part of the same *reuse group*. Otherwise, the two references are part of distinct *reuse groups*.

For our example in Figure 7.2, the group of references to array `A` is split into two *reuse groups*. One *reuse group* contains references `A(I,J,K)` and `A(I+1,J,K)`, and

the second *reuse group* contains references `A(I+2,J,L)` and `A(I+3,J,L)`. The four references have been separated into two *reuse groups* because they access memory locations far apart, due to different indices in their third dimension. In contrast, all four references to array `B` are part of a single *reuse group*.

Step 3. Compute the hot foot-print information for each *reuse group* derived from a group of related references. For this, we use modular arithmetic to map the locations accessed by all references of a *reuse group* to a block of memory of size s, effectively computing the coverage of the block, *i.e.*, the number of distinct bytes accessed in the block. For a group of related references we select the maximum coverage, c, over all its *reuse groups*, and the fragmentation factor is $f = 1 - c/s$.

Returning to our example, both *reuse groups* corresponding to the set of references to array `A` have a coverage of 16 bytes, and thus the fragmentation factor for array `A` is 0.5. The single *reuse group* for the four references to array `B` has coverage 32, and thus a fragmentation factor of 0.

While it is possible to have non unit stride accesses to arrays of simple data types, the main culprits of data fragmentation are arrays of records, where only some of the record fields are accessed in a particular loop. The problem can be solved by replacing the array of records with a collection of arrays, one array for each individual record field. A loop working with only a few fields of the original record needs to load into cache only the arrays corresponding to those fields. If the original loop was incurring cache misses, this transformation will reduce the number of misses, which will reduce both the data bandwidth and memory delays for the loop. This transformation has

the secondary effect of increasing the number of parallel data streams in loops that work with multiple record fields. While additional streams can improve performance by increasing memory parallelism [52], they can hurt performance on architectures with small TLBs and architectures that use hardware prefetching but can handle only a limited number of data streams (see section 8.3.2).

For our example in Figure 7.2, and based only on the code in that loop, array `A` is better written as two separate arrays, each containing every other group of two elements of its inner dimension.

Using the fragmentation factors derived for each group of related references, we compute metrics which specify how many cache misses at each memory level are due to fragmentation effects. The number of cache misses due to cache line fragmentation is computed separately for each memory reuse pattern; we report this information at the level of individual loops and data arrays. Similarly, we compute the number of cache misses due to irregular reuse patterns. A reuse pattern is considered irregular if its carrying scope produces an irregular or indirect symbolic stride, as explained in section 3.1, for the references at its destination end.

7.3 Interpreting the Performance Data

To identify performance problems and opportunities for tuning, we output all metrics described in the previous sections in XML format, and we use the `hpcviewer` user interface [45] that is part of HPCToolkit [48] to explore the data. The viewer enables us to sort the data by any metric and to associate metrics with the program

source code. We output three databases which represent three different views of the computed performance data. The main database presents the inclusive metrics aggregated at all levels of a program scope tree. It can be browsed in a top-down fashion to find regions of code that account for a significant fraction of any performance metric.

Not all metrics can be sensibly aggregated based on the static program scope tree. For example, aggregating the number of misses carried by scopes based on their static program hierarchy is meaningless. The carried number of misses is rather a measure representative of the dynamic tree of scopes observed at run-time. This information could be presented hierarchically along the edges of a calling context tree [3] that includes also loop scopes. A reuse pattern already specifies the source, the destination and the carrying scopes of a reuse arc; aggregating the number of misses carried by scopes does not seem to provide any additional insight into reuse patterns. While for some applications the distribution of reuse distances corresponding to a reuse pattern may be different depending on the calling context, for most scientific programs separating the data based on the calling context may dilute the significance of some important reuse patterns. At this point we do not collect data about the memory reuse patterns separately for each context tree node to avoid the additional complexity and run-time overhead. If needed, the data collection infrastructure can be extended to include calling context as well.

The second database presents the exclusive values of all metrics in a flat view, such that all program scopes can be compared as equal peers. This is the recommended view to identify program scopes that carry the highest number of long data reuses, but it can be used to understand the most significant scopes with respect to other

metrics as well.

The third database includes only data about the memory reuse patterns. Once again, this is a flat database. However, its entries correspond not to individual scopes, but to pairs of scopes that represent the source and destination scopes of reuse patterns. Its purpose is to quickly identify the reuse patterns contributing the highest number of cache misses at each memory level.

Identifying the performance bottlenecks is only part of the work, albeit a very important part. We need to understand what code transformations work best in each situation. In this section we provide suggestions of code transformations for the most important inefficiencies identified by our toolkit.

For loop nests with recurrences in the inner most loops, the most important transformations for increasing instruction-level parallelism are: loop interchange, unroll & jam, recognition of reductions, loop skewing and loop fusion. These transformations are described in great detail in [2]. Note that these transformations are not legal in all cases and their profitability vary case by case. In general there is a tradeoff between high ILP and short memory reuse distances. This is particularly true when trying to optimize for register reuse, since the number of registers is generally small. When optimizing for cache reuse, we have more leeway because caches are larger and we do not incur a penalty until the data starts to be evicted from the cache.

For example, while loop interchange can significantly boost the amount of ILP if we find an outer loop without recurrences that can be legally interchanged to the innermost position, it also increases the reuse distances for the references that are part of the recurrence. This may be desirable for vector machines that benefit the

most from instruction-level parallelism if the compiler can hide the memory latency. For super-scalar machines which have limited parallelism, we are better off taking the middle ground. We can balance the amount of available ILP and the distances of data reuses by strip-mining the loop without recurrences, moving the loop that iterates over a stripe in the innermost position and then unrolling it. This transformation is equivalent to the unroll & jam transformation which consists of unrolling an outer loop a number of times, and then fusing the resulting inner loops. By controlling the size of one stripe or the unrolling factor we can determine a balance between ILP and memory locality. We do not get any additional benefit if we increase the amount of available ILP beyond the width of the target machine. This is why these transformations are in general machine dependent.

Scenario	Transformations & comments
large number of fragmentation misses due to one array	data transformation: split the original array into multiple arrays
large number of irregular misses and $S \equiv D$	apply data or computation reordering
large number of misses and $S \equiv D$, C is an outer loop of same loop nest	carrying scope iterates over the array's inner dimension; apply loop interchange or dimension interchange on the affected array; if multiple arrays with different dimension orderings, loop blocking may work best
$S \not\equiv D$, C is inside same routine as S and D	fuse S and D
as the previous case, but S or D are in a different routine invoked from C	strip-mine S and D with the same stripe and promote the loops over stripes outside of C, fusing them in the process
C is a time step loop or a main loop of the program	apply time skewing if possible; alternatively, do not focus on these hard or impossible to remove misses

Table 7.1: Recommended transformations for improving memory reuse.

Table 7.1 summarizes recommended transformations for improving memory reuse, based on the type of reuse pattern that is producing cache misses. We use S, D and C to denote the source, the destination and the carrying scopes of a reuse pattern. These recommendations are just that, general guidelines to use in each situation. Determining whether a transformation is legal is left for the application developer. In some instances, enabling transformations such as loop skewing or loop alignment may be necessary before we can apply the transformations listed in table 7.1.

Chapter 8

Case Studies

In this chapter, we briefly illustrate how to analyze and tune an application using the performance metrics described in section 7. We study three applications. Sweep3D [19] is a 3D Cartesian geometry neutron transport code benchmark from the DOE's Accelerated Strategic Computing Initiative. As a procurement benchmark, this code has been carefully tuned already. The Parallel Ocean Program (POP) [36] is a climate modeling application developed at Los Alamos National Laboratory. This is a more complex code with the execution cost spread over a large number of routines. The Gyrokinetic Toroidal Code (GTC) [34] is a particle-in-cell code that simulates turbulent transport of particles and energy in burning plasma. We compiled the three codes on a Sun UltraSPARC-II system using the Sun WorkShop 6 update 2 FORTRAN 77 5.3 compiler, and the optimizations: *-xarch=v8plus -xO4 -depend -dalign -xtypemap=real:64*.

8.1 Analysis and Tuning of Sweep3D

Sweep3D performs a series of diagonal sweeps across a 3D Cartesian mesh, which is distributed across the processors of a parallel job. Figure 8.1 presents a schematic

diagram of the computational kernel of Sweep3D. The `idiag` loop is the main computational loop on each node. It performs a sweep from one corner of the local mesh to the opposing corner. In each iteration of the `idiag` loop, one diagonal plane of cells is processed by the `jkm` loop. Before and after the `idiag` loop there is MPI communication to exchange data with the neighboring processors. Finally, the outer `iq` loop iterates over all octants, starting a sweep from each corner of the global mesh.

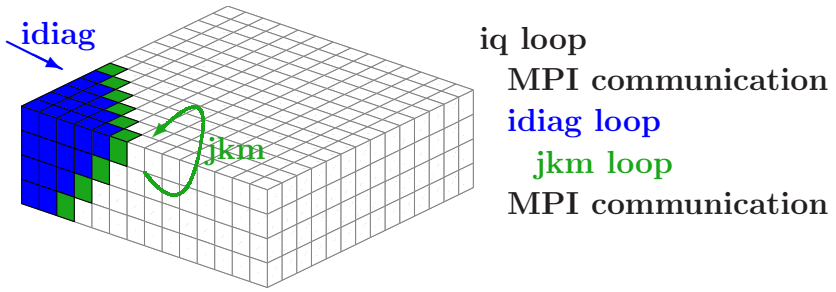


Figure 8.1: Main computation loops in Sweep3D.

For Sweep3D, we used our toolkit to collect edge frequency data and memory reuse distance information for a cubic mesh size of $50 \times 50 \times 50$ and 6 time steps without fix-up code. We used the reuse distance data to compute the number of L2, L3, and TLB misses for an Itanium2 processor with a 256KB 8-way set-associative L2 cache, 1.5MB 6-way set-associative L3 cache, and a 128-entry fully-associative TLB. We processed all data using our modulo-scheduler instantiated with a description of the Itanium2 architecture to generate the three XML databases with our performance metrics.

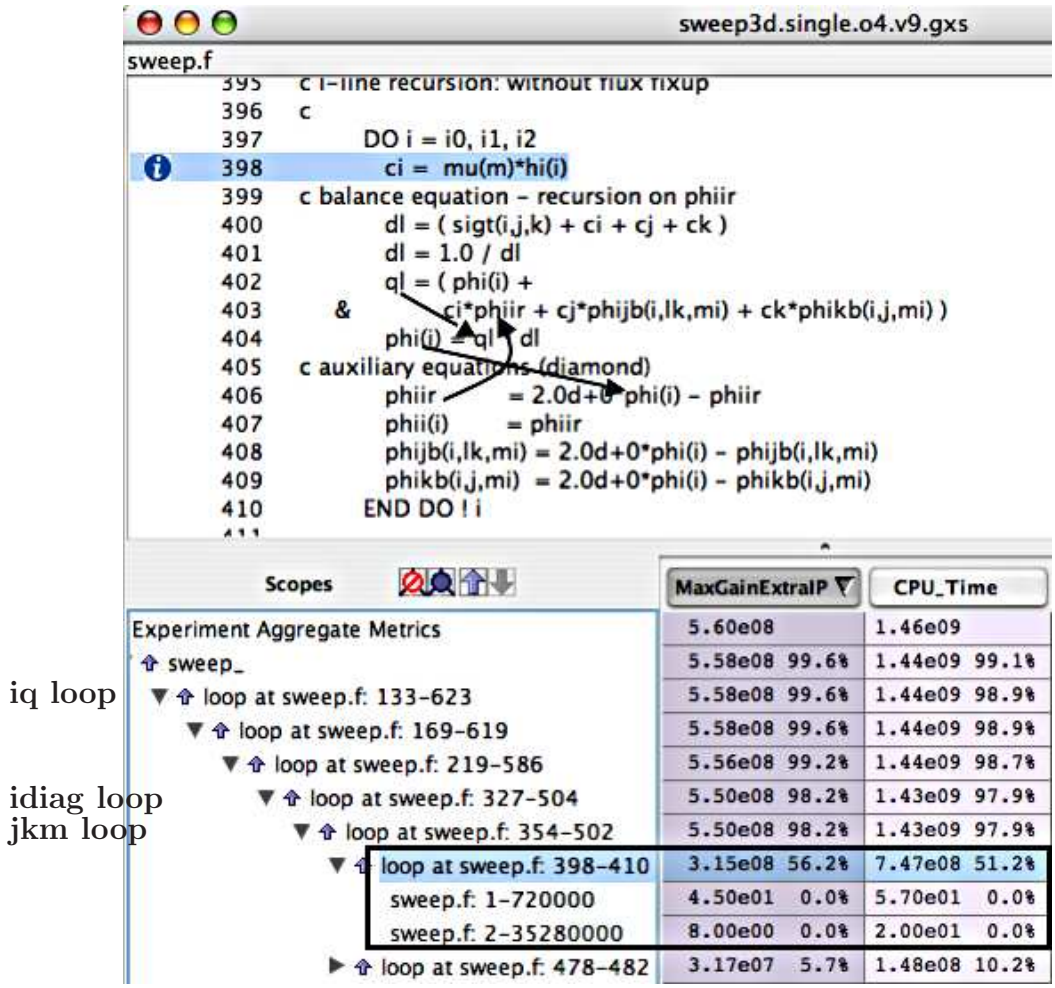


Figure 8.2: Sweep3D: understanding potential for improvement from increased ILP.

8.1.1 Identifying Performance Inefficiencies Due to Insufficient ILP

Figure 8.2 shows a snapshot of the `hpcviewer` interface browsing the performance database sorted by the $MaxGain_{ILP}$ metric (in the bottom right pane). Due to limited horizontal space in a paper, only two metrics are shown: i) maximum gain expected from increased ILP, and ii) predicted computation time. We expanded six levels of scopes that are shown to account for over 98% of the potential for improvement according to this metric. While 98% of this potential is contained within the

level five loop at lines [354–502] (the `jkm` loop), the I-line recursion loop without flux fixup at lines [398–410] accounts for 56% of the entire potential and the next most significant level six loop accounts for only 5.7% of this potential. It is clear that we have to focus our attention on the I-line recursion loop. By expanding the scope of this loop, we expose the performance data for the two paths through this loop. While for loops we present metric totals, for paths we show metric values per iteration.

When we reconstruct the paths taken through a loop, we consider the paths that follow the loop back-edge and the exit paths separately. Back-edge paths are scheduled using software pipelining, while for exit paths software pipelining is disabled. For this loop, first path is the exit path and we notice the program enters this loop 720K times, and second path is the back-edge path which is executed over 35 million times. Looking at predicted computation time if we had an infinite number of execution units, metric not included in the figure, we notice that even if we had an infinitely wide machine the time per iteration would still be 20 clock cycles, while if we could remove the dependences, the time per iteration would drop by 40% to 12 clock cycles. This is the clear sign of a recurrence of 20 clock cycles in the code. We spotted two short recurrences, but the longest recurrence is the one marked in the source pane of Figure 8.2.

By manual inspection, we realized that loop `jkm` at lines [354–502] has no carried dependencies. If we unroll the `jkm` loop and then fuse all instances of its inner loops, we can execute multiple I-line recursions in parallel, effectively increasing the ILP. We decided to unroll the `jkm` loop only once since we already fill 60% of the issue cycles with one iteration. After transforming the code, we ran it again through our

tool. The predicted overall computation time[1] dropped by 20% from 1.46e09 down to 1.17e09, and the total potential for improvement from additional ILP has dropped by more than a factor of four, from 5.60e08 down to 1.35e08. This potential, however, is due only to loop exit paths and outer loops that cannot be effectively pipelined. For the I-line recursion loop the value of this metric dropped by a factor of 25.

All numbers presented so far are predictions from our tool. To see if these predicted improvements can be observed on a real Itanium2 machine as well, we compiled both the original and the transformed Sweep3D codes on a machine with an Itanium2 CPU running at 900MHz. We compiled the codes with the Intel Fortran Itanium Compiler 9.0, and the optimization flags: *-O2 -tpp2 -fno-alias*.[2] Using hardware performance counters we measured the execution time, the number of L2, L3 and TLB misses, and the number of instructions and NOPs retired, for both binaries and for mesh sizes from $10 \times 10 \times 10$ to $200 \times 200 \times 200$. Figure 8.3 presents the performance of the transformed code relative to the performance of the original code for all input sizes. We notice the transformed program is consistently faster by 13-18% with an average reduction of the execution time of 15.6% across these input sizes. The number of cache and TLB misses in the two versions of the code differ by only 2-3%, thus they cannot account for the performance increase. The spikes for L3 and TLB misses at small problem sizes are just an artifact of the very small miss rate at those input sizes. For larger problem sizes the differences are negligible. However, we see the number of retired instructions dropped by 16.3% and the number of retired NOPs dropped by

[1] This metric does not include memory penalty.
[2] We tried -O3 as well, but -O2 yielded higher performance.

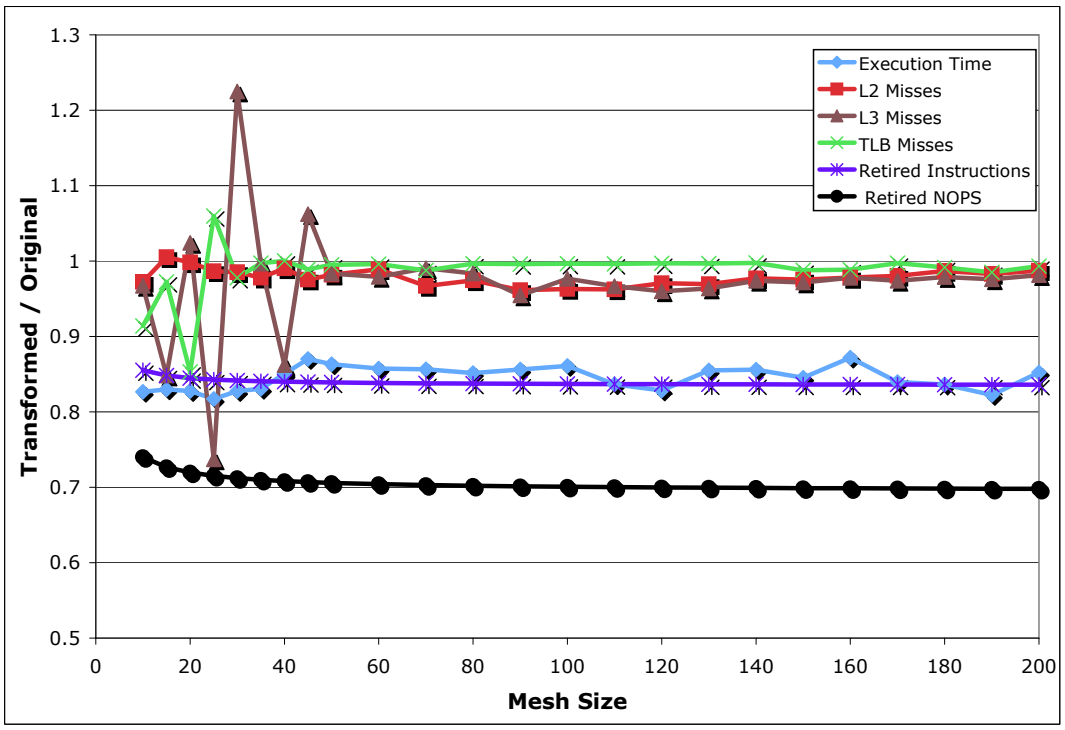


Figure 8.3: Performance of the transformed Sweep3D code relative to the original version on an Itanium2 (lower is better).

30%, a sign that issue bundles are filled with more useful instructions. We observed also an increase in memory parallelism in the transformed program, which lowers the exposed memory penalty and may account for part of the observed speedup.

Similar recurrences are in the I-line recursion loop with flux fixup at lines [416–469]. Which version of the loop is executed is controlled from the input file. The transformed code improves the execution time of that loop by a similar factor, and our measurements on Itanium2 confirmed this result.

We should mention that the performance increase obtained on Itanium2 might not be observed on every architecture. The I-line recursion loop contains a floating point divide operation (see Figure 8.2). If the throughput of the divider unit is low, or if the machine issue width is much lower, combined with possibly increased contention on

other units, then the loop might be resource limited even in the original form, or the improvement could be only modest. However, our tool will predict correctly the lack of potential gains if the machine model is accurate. Itanium2 has a large issue width and floating point division is executed in software with a sequence of fully-pipelined instructions. Thus, while the latency of one divide operation from start to finish may be longer than what could be obtained in hardware, this approach is efficient when many divide operations need to be executed.

Figure 8.4 presents the effect of issue width on the relative performance improvement of Sweep3D. We used the Itanium2 machine model, and we varied the issue width between 3 and 8, everything else being unchanged. The predicted performance

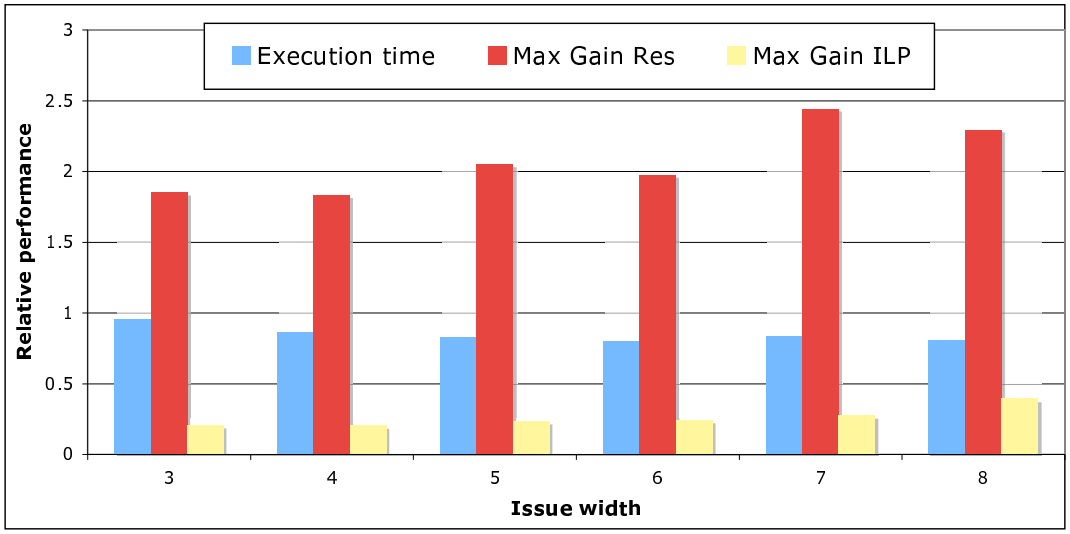


Figure 8.4: Performance of the transformed code relative to the original version, for an Itanium2 like machine with issue width from 3 to 8.

improvement is only 5% when issue width is cut to 3. For issue widths larger than 5, the relative performance of the codes varies very little, with the transformed version being between 17% and 20% faster. The transformed program is much more limited

by issue width, running 40% slower at issue width 3 when compared to issue width 6. At the same time, the original program is less sensitive, running only 20% slower when at most 3 instructions can be issued per cycle. For issue widths larger than 5, there is little improvement observed in both codes, other resources becoming the limiting factor. As we expected, the potential gain from increased ILP is much lower in the transformed code across all issue widths. The situation is reversed in the case of the improvement potential from additional machine resources.

8.1.2 Identifying Opportunities for Improving Data Reuse

In this section we use our memory reuse based metrics to identify the reuse patterns that account for the highest number of cache and TLB misses. Performance improvement from better data reuse can be even more substantial than what we saw in the previous section. Figure 8.5 shows a snapshot from our user interface of the predicted number of carried misses for the L2 and L3 caches and for the TLB. We notice that approximately 75% of all L2 cache misses and about 68% of all L3 cache misses are carried by the `idiag` loop, while the `iq` loop carries 10.5% and 22% of the L2 and L3 cache misses respectively. The situation is different with the TLB misses. The `jkm` loop carries 79% and the `idiag` loop carries 20% of all the TLB misses.

Scopes	Carried_L2D		Carried_L3D		Carried_TLB	
Experiment Aggregate Metrics	3.85e07	100.0	1.73e07	100.0	8.78e06	100.0
idiag loop ▶ loop at sweep.f: 327-491	2.87e07	74.7%	1.18e07	68.3%	1.79e06	20.4%
iq loop ▶ loop at sweep.f: 133-491	4.04e06	10.5%	3.84e06	22.2%	3.10e04	0.4%
jkm loop ▶ loop at sweep.f: 354-491	3.43e06	8.9%	6.81e03	0.0%	6.94e06	79.0%

Figure 8.5: Number of carried misses in Sweep3D

We focus on the L2 and L3 cache misses. The fact that such a high fraction of

all cache misses are carried by the `idiag` loop is a good thing from a tuning point of view, because we can focus our attention on this loop. While the `iq` loop carries the second most significant number of misses, it contains also calls to communication functions. Thus, it may require more complex transformations to improve, in case it is possible at all. Table 8.1 summarizes the main reuse patterns contributing the

Array name	In scope	Reuse source	Carrying scope	% misses
src	loop 384–391	self	ALL	26.7
			idiag	20.4
			iq	3.3
			jkm	2.9
flux	loop 474–482	self	ALL	26.9
			idiag	20.4
			iq	3.4
			jkm	3.0
face	loop 486–493	self	ALL	19.7
			idiag	15.5
			iq	2.4
			jkm	1.9
sigt phikb phijb	loop 397–410	self + others	ALL	18.4

Table 8.1: Breakdown of L2 misses in Sweep3D.

highest number of L2 cache misses in Sweep3D. We notice that four loop nests inside the `jkm` loop account for the majority of the L2 cache misses. For three of these loop nests, only accesses to one data array in each of them result in cache misses. Since the `idiag` loop carries the majority of these cache misses, we can focus our attention on understanding how the array indices are computed with respect to this loop.

Figure 8.6 shows the Fortran source code for the first two loop nests that access arrays `src` and `flux` respectively. We notice that both the `src` and the `flux` arrays

```
384  do i = 1, it
385    phi(i) = src(i,j,k,1)
386  end do
387  do n = 2, nm
388    do i = 1, it
389      phi(i) = phi(i) + pn(m,n,iq)*src(i,j,k,n)
390    end do
391  end do
...
474  do i = 1, it
475    flux(i,j,k,1) = flux(i,j,k,1) + w(m)*phi(i)
476  end do
477  do n = 2, nm
478    do i = 1, it
479      flux(i,j,k,n) = flux(i,j,k,n)
480 &       + pn(m,n,iq)*w(m)*phi(i)
481    end do
482  end do
```

Figure 8.6: Accesses to `src` and `flux`.

are four dimensional arrays and that both of them are accessed in a similar fashion. In Fortran, arrays are stored in column-major order. Thus, the first index represents the innermost dimension and the last index is the outer most one. We notice that for both `src` and `flux`, the innermost loop matches the innermost dimension. However, the next outer loop, `n`, accesses the arrays on their outermost dimension. We return to this observation later. For now, we want to understand how the `j` and `k` indices are computed.

We mentioned that in each iteration of the `idiag` loop, the `jkm` loop traverses one diagonal plane of cells as seen in Figure 8.1. Each cell of the 3D mesh is defined by unique coordinates `j`, `k` and `mi`, as seen in Figure 8.7. Notice that all cells of a 3D diagonal plane have different `j` and `k` coordinates. Thus, there is no temporal reuse of `src` and `flux` carried by the `jkm` loop. The small amount of reuse observed in

Table 8.1 is spatial reuse due to the sharing of some cache lines between neighboring cells. However, even this reuse is long enough that it results in cache misses, because the cells in a plane are not necessarily accessed in the order in which they are stored.

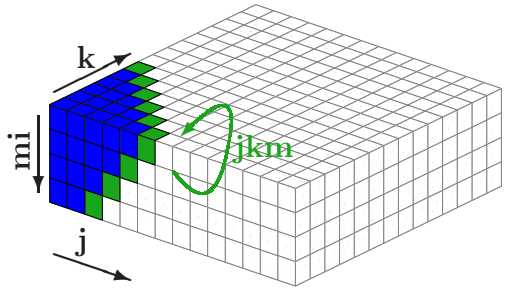


Figure 8.7: Sweep3D: `jkm` iteration space

Consecutive `idiag` iterations access adjacent diagonal planes of cells. When we project these 3D diagonal planes onto the (`j`,`k`) plane, we notice there is a great deal of overlap between two consecutive iterations of the `idiag` loop. This explains the observed reuse carried by the `idiag` loop. However, the reuse distance is too large for the data to be still in cache on the next iteration of the `idiag` loop. Finally, the reuse carried by the `iq` loop is explained by the fact that we traverse again all cells of the mesh on a new sweep that starts from a different corner.

Notice that arrays `src` and `flux` (and `face` as well) are not indexed by the `mi` coordinate of a cell. Thus, references to the three arrays corresponding to cells on different diagonal planes that differ only in the `mi` coordinate, but with equal `j` and `k` coordinates access identical memory locations. To improve data reuse for these arrays, we need to process closer together mesh cells that differ only in the `mi` coordinate.

For this, we apply tiling to the `jkm` loop on the `mi` coordinate.The transformed sweep iteration space is represented graphically in Figure 8.8, for a blocking factor of

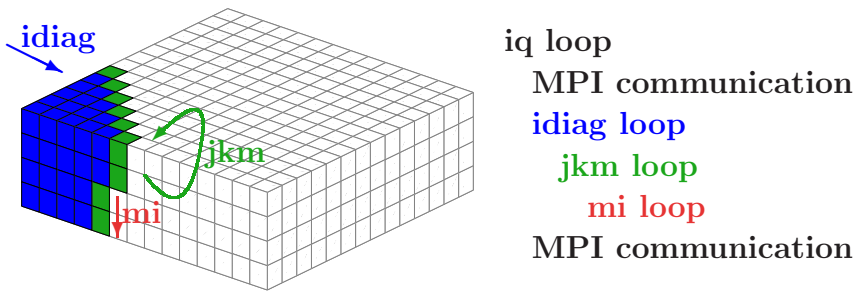


Figure 8.8: Diagram of Sweep3D after blocking on `mi`.

two. Note that `mi` is not a physical dimension of the 3D mesh; rather, it represents different angles at which the neutron movements are simulated. The third physical coordinate is `i` which is contained within each cell. Thus, by simulating multiple angles at once, we achieve better data reuse. The number of angles specified in our input file was six. Therefore, we measured the performance of the transformed code on an Itanium2 system using blocking factors of one, two, three and six.

Figures 8.9(a),(b) and (c) present the number of L2, L3 and TLB misses for the original code and for the transformed code with the four different blocking factors. All figures present the performance metrics normalized to the number of cells and time steps so that the results for different problem sizes can be easily shown on a single graph. The figures show that the original code and the code with a blocking factor of one have identical memory behavior. As the blocking factor increases, less and less accesses miss in cache. The last curve in each figure represents the performance of the transformed code with a blocking factor of six plus a dimensional interchange for several arrays to better reflect the way in which they are traversed. For the `src` and `flux` arrays we moved the `n` dimension into the second position. These transformations reduce cache and TLB misses by integer factors.

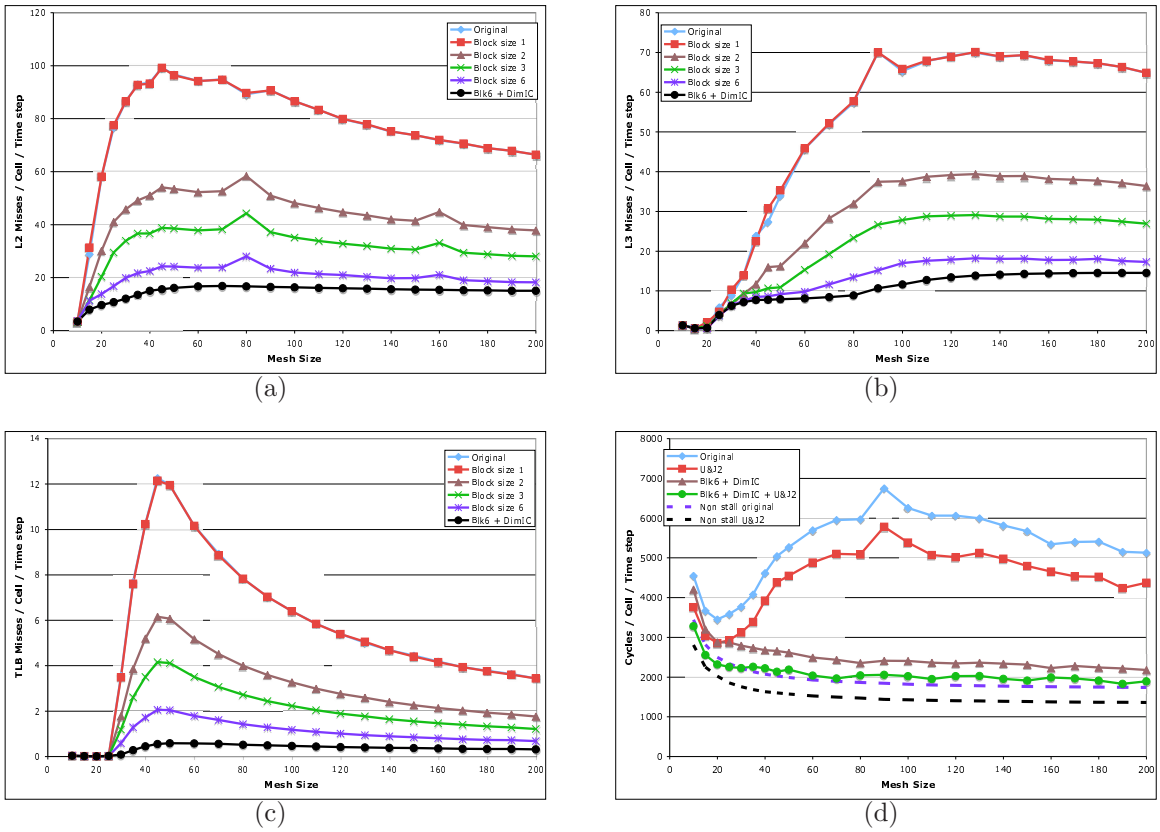


Figure 8.9: Performance of the original and improved Sweep3D codes on an Itanium2 system.

Figure 8.9(d) compares the normalized execution times of the original and transformed codes. We present separate measurements for the execution times of the code with only the unroll & jam transformation presented in section 8.1.1, the code modified for better data reuse, as well as the code containing both transformations. The code with both increased ILP in the I-line recursion loop and improved data reuse has a speedup of 3x and we can observe ideal scaling of the execution time between mesh sizes 20 and 200 which represents a thousand-fold increase of the working set size. Figure 8.9(d) shows also the non-stall execution times, measured with hardware performance counters, of the codes with and without the unroll & jam transforma-

tion. Notice that we eliminated a large fraction of the observed stall time with our transformations. Note also that the non-stall times depicted in the figure are not the absolute minimum times that can be achieved on the Itanium. They are just the minimum times that can be achieved with the instruction schedule generated by the Intel compiler. By applying unroll & jam on the `jkm` loop, we reduced Sweep3D's execution time, as well as its non-stall time, by improving the compactness of the instruction schedule.

8.2 Analysis and Tuning of POP

We applied our modeling techniques to the Parallel Ocean Program (POP) [36], a climate modeling application developed at the Los Alamos National Lab. In this section we present our findings for this application. POP is a more complex application than Sweep3D and it has no single routine that accounts for a significant percentage of the running time. Table 8.2 presents performance data for the top eight routines based on the potential for improvement from additional ILP. Values are predicted for an execution of POP 2.0.1 with the default benchmark size input on an Itanium2 machine model. In addition to the $MaxGain_{ILP}$ metric, Table 8.2 includes the predicted computation time and the rank of each routine if data was sorted by computation cost.

Unlike Sweep3D where effectively all execution time is spent in a single loop nest, in POP the execution cost is spread over a large number of routines. A traditional analysis looking for the most time consuming loops would not work well on this code

Routine	$\mathbf{MaxGain_{ILP}}$		CpuTime		Rnk
boundary_2d_dbl	4.02e08	13.8%	6.99e08	6.2%	4
impvmixt	3.53e08	12.1%	6.17e08	5.5%	5
impvmixt_correct	3.45e08	11.8%	5.85e08	5.2%	7
global_sum_dbl	2.44e08	8.4%	3.38e08	3.0%	15
diag_global_preup	1.51e08	5.2%	3.26e08	2.9%	17
impvmixu	1.51e08	5.2%	3.26e08	2.9%	18
advt_centered	1.50e08	5.1%	3.85e08	3.4%	12
tracer_update	1.43e08	4.9%	7.78e08	6.9%	2

Table 8.2: POP: top routines based on the improvement potential from additional ILP.

since there is not a single loop or routine that accounts for a significant percentage of the running time. Sorting scopes by the $MaxGain_{ILP}$ metric enables us to at least limit our investigation to those scopes that show a non-negligible potential for improvement. The routine with the highest predicted potential for improvement, `boundary_2d_dbl`, at closer inspection proved to contain some frequently executed short loops which do not contain recurrences. The potential for improvement shown for this routine is the result of frequently executed loop exit paths that are not software pipelined. However, the following two routines, `impvmixt` and `impvmixt_correct`, contain loops with recurrences that account for most of the predicted improvement potential of these routines. These loops perform a tridiagonal solve in the vertical for every horizontal grid point and each tracer. Both these routines perform a very similar computation, thus what we describe below applies to both of them.

By visually inspecting the code of routine `impvmixt`, we realized that neither of the two outer loops carry any dependences, the computation for each horizontal grid point being independent. This means we can apply unroll & jam again to increase the ILP. However, the upper bound of the tridiagonal solve loop is a function of

the ocean depth of each grid point. If we want to perform the tridiagonal solve for two grid points at a time, we must either ensure that they have the same depth, or compute the solve in parallel up to the minimum depth of a pair of points followed by a reminder loop that computes the residual points for the deepest point. Because the maximum depth specified in the benchmark input file is relatively small and the depths are integer values, we decided to implement the first solution. For this, we use a temporary array, of size equal to the maximum possible depth, to store the coordinates of the most recently encountered horizontal grid point with a given depth.

For each horizontal grid point the modified code performs these steps:

1. compute current point's depth and check the corresponding entry in the temporary array;

2. if the entry is empty, store the coordinates of current point and go to the next horizontal grid point, otherwise go to step 3;

3. read the coordinates of the grid point of same depth from the temporary array entry and perform the tridiagonal solve in parallel for the current point and the point found in the temporary array. Reset the temporary array entry once we used the point and go to the next horizontal grid point.

Once all points are visited we check every entry in the temporary array and perform the tridiagonal solve individually for all odd points for which we did not find a pair point of same depth in the main loop.

Routines `impvmixt` and `impvmixt_correct` are very similar. They have almost identical code, including the tridiagonal solve loop. We transformed the code for routine `impvmixt_correct` using the same approach we used for the `impvmixt` routine. The other routines shown in Table 8.2 have a smaller potential for improvement from additional ILP. We found that with the exception of routine `impvmixu` which is similar to the routines presented above, the other routines do not contain real recurrences, but the potential for improvement is due to reduction operations that are not sufficiently parallelized by the Sun compiler used to compile the analyzed binary. We expect the Intel compiler will compute the reductions more efficiently given the fact it is targeted to a wider-issue architecture. We decided not to transform the code corresponding to routine `impvmixu` since this routine contributes less than 3% to the total computation time, and this percentage is even smaller once the memory penalty is considered.

We measured the performance of the original and the transformed versions of the code on our Itanium2 based machine. Since the two routines that were modified together account for only around 10% of the computation time in the original code, and since our transformations can in the best case scenario cut their computation cost in half, we expect at most a 5% improvement for the overall program. Once we consider the memory hierarchy delays that our transformations do not attempt to improve, the overall improvement that can be achieved should be even less. Using the default benchmark size input file, 192×128 horizontal grid points, we measured an overall performance increase of 3.78% for the transformed version. Increasing the number of grid points slightly to 208×144, the observed performance increase was 4.55%. While the overall improvement for this code is not substantial for reasons

explained earlier, the transformation of the code was straightforward, and our performance modeling tool led us to these loops and predicted accurately the potential for improvement.

8.3 Analysis and Tuning of GTC

The Gyrokinetic Toroidal Code is a 3D particle-in-cell (PIC) code used for studying the impact of fine-scale plasma turbulence on energy and particle confinement in the core of tokamak fusion reactors [64]. The PIC algorithm consists of three main sub-steps: 1) deposit the charge from particles onto the grid (routine `chargei`), 2) compute and smooth the potential field (routines `poisson` and `smooth`), and 3) compute the electric field and push particles using Newton's laws of physics (routines `pushi` and `gcmotion`). Compared to the Sweep3D benchmark, the GTC code is significantly more complex with the computation kernel spread over several files and routines.

For GTC, we collected edge frequency counts and reuse distance data for a problem size consisting of a single poloidal plane with 64 radial grid points and 15 particles per cell. From the reuse distance histograms collected for each reuse pattern, we computed the number of cache misses, the number of misses due to fragmentation in cache lines, the number of irregular misses, and the number of carried misses as explained in section 7.2, for an Itanium2 cache architecture. All metrics are computed at loop level as well as for individual data arrays.

Figure 8.10 presents a snapshot of our viewer showing the data arrays that account

Scopes	Misses_L3D		FragMiss_L3D	
@Data Names	7.90e07	100.0	1.14e07	100.0
particle_array.zion_	1.45e07	18.4%	8.47e06	74.4%
particle_array.zion0_	5.61e06	7.1%	1.30e06	11.5%
particle_array	2.42e06	3.1%	1.04e06	9.1%
wpi	5.66e05	0.7%	2.83e05	2.5%

Figure 8.10: Data arrays contributing the largest number of fragmentation L3 misses.

for the highest number of L3 cache misses due to fragmentation of data in cache lines. The first metric in the figure represents the total number of L3 cache misses incurred by all accesses to these arrays in the entire program. Data arrays `zion` and its shadow `zion0` are global arrays storing information about each particle in the local tokamak domain. They are defined as 2D Fortran arrays organized as arrays of records with seven data fields for each particle. Array `particle_array` is an alias for the `zion` array, used inside a "`C`" routine `gcmotion`.

Notice that accesses to the two `zion` arrays, including the alias `particle_array`, account for 95% of all fragmentation misses to the L3 cache. This amounts to about 48% of all L3 cache misses incurred on the `zion` arrays, and about 13.7% of all L3 cache misses in the program. Most loops that work with the `zion` arrays reference only a few of the seven fields associated with each particle. Using our viewer, we identified the loops with the highest contribution to the miss and fragmentation metrics. We noticed two loops where only one out of the seven fields of the `zion` array was referenced for each particle. To eliminate unnecessary cache misses due to fragmentation, we transposed the two `zion` arrays, so that each of the seven fields is stored separately in its own vector. This amounts to transforming the array of structures into a structure of arrays.

Scopes	Carried_L2D	Carried_L3D
Experiment Aggregate Metrics	9.35e07 100.0	7.69e07 100.0
loop at main.F90: 146-266	2.23e07 23.8%	2.21e07 28.7%
loop at poisson.F90: 74-119	1.67e07 17.9%	1.65e07 21.4%
pushi_	1.61e07 17.2%	1.61e07 20.9%
loop at main.F90: 139-343	8.68e06 9.3%	8.67e06 11.3%
chargei_	8.71e06 9.3%	8.55e06 11.1%

Figure 8.11: Program scopes carrying the most L3 cache misses.

Figure 8.11 presents the program scopes that carry more than 2% of all L3 cache misses. The loop at `main.F90:139-343` is the main loop of the algorithm iterating over time steps and it carries about 11% of all L3 cache misses. Each time step of the PIC algorithm executes a 2nd order Runge-Kutta predictor-corrector method, represented by the second loop of the main routine, at lines `146-266`. The two main loops carry together about 40% of all L3 cache misses. These are cache misses due to data reuse between the three sub-steps of the PIC algorithm, and across consecutive time steps or the two phases of the predictor-corrector method in each time step. Because each of the three sub-steps of the PIC algorithm requires the previous step to be completed before it can start executing, these cache misses cannot be eliminated by time skewing or pipelining of the three sub-steps. Thus, we focus our attention on the other opportunities for improvement.

The `poisson` routine computes the potential field on each poloidal plane using an iterative Poisson solver. Cache misses are carried by the iterative loop of the Poisson solver (at lines `74-119`), and unfortunately cannot be eliminated by loop interchange or loop tiling because of a true recurrence in the solver. We did however notice

that the highest number of cache misses in the `poisson` routine were incurred on two three dimensional arrays, `ring` and `indexp`, even though they were accessed with unit stride. At a closer inspection we found the upper bound of the innermost loop that was iterating over the inner dimension of these two arrays, was not constant. Thus, only some of the elements on a column were being accessed, resulting in partially utilized cache blocks at the end of each column. Our cache fragmentation analysis cannot detect such cases at this time, because the elements are accessed with stride one, and the elements that are not accessed are contiguous at the end of each column. We reorganized these arrays into contiguous linear arrays which improves spatial locality. This transformation removes only a small fraction of the total number of cache misses incurred on these arrays. There is unfulfilled temporal reuse carried by the iterative loop of the Poisson solver, which cannot be improved.

However, the amount of work in the Poisson solver is proportional to the number of cells in the poloidal plane. As we increase the number of particles that are simulated, the cost of the charge deposition and particle pushing steps increases, while the cost of the Poisson solver stays constant. Thus, the execution cost of the `poisson` routine becomes relatively small in comparison to the cost of the entire algorithm as the number of particles per cell increases.

We focus now on the `chargei` and the `pushi` routines. Our analysis identified that about 11% of all L3 cache misses are due to reuse of data in two loops of the `chargei` routine that iterate over all particles. The first loop was computing and storing a series of intermediate values for each particle; the second loop was using those values to compute the charge deposition onto the grid. However, by the time

the second loop accessed the values computed in the first loop, they had been evicted from cache. By fusing the two loops, we were able to improve data reuse in `chargei`, and to eliminate these cache misses.

The `pushi` routine calculates the electrical field and updates the velocities of the ion particles. It contains several loop nests that iterate over all the particles, and a function call to a "`C`" routine, `gcmotion`. The `gcmotion` routine consists of a single large loop that iterates over all the particles as well. Our analysis identified that for the problem size that we used, `pushi` carries about 20% of all L3 cache misses between the different loop nests and the `gcmotion` routine. This reuse pattern corresponds to the fifth entry in Table 7.1, because the `gcmotion` routine is both a source and a destination scope for some of the reuse arcs carried by `pushi`. While `gcmotion` consists of just one large loop, we cannot inline it in `pushi` because these two routines are written in different programing languages. Instead, we identified a set of loops that we could fuse, strip-mined all of them, including the loop in `gcmotion`, with the same stripe s, and promoted the loops over stripes in the `pushi` routine, fusing them. The result is a large loop over stripes, inside of which are the original loop nests and the function call to `gcmotion`. These transformed loop nests work over a single stripe of particles, which is short enough to ensure that the data is reused in cache.

We also identified a loop nest in routine `smooth` that was contributing about 64% of all TLB misses for the problem size that we used. The outer loop of the loop nest, which was carrying all these TLB misses (see Figure 8.12), was iterating over the inner dimension of a three dimensional array. We were able to apply loop interchange and promote this loop in the innermost position, thus eliminating all these TLB misses.

Scopes	Carried_TLB	
Experiment Aggregate Metrics	1.75e06	100.0
loop at smooth.F90: 336-346	1.12e06	64.1%
loop at main.F90: 146-266	1.84e05	10.5%
loop at poisson.F90: 74-119	1.39e05	7.9%
pushi_	1.27e05	7.2%
loop at main.F90: 139-343	7.11e04	4.1%

Figure 8.12: Program scopes carrying the most TLB misses.

Analyzing performance bottlenecks due to insufficient instruction-level parallelism, we identified a recurrence in a prime factor transform routine `spcpft`. We increased the amount of instruction-level parallelism by applying unroll & jam. We also identified a similar short recurrence in one loop nest of the Poisson solver where we applied unroll & jam to increase fine-grain parallelism.

8.3.1 GTC Performance on the Itanium2

Figure 8.13 presents the single node performance of GTC on a 900MHz Itanium2. The four graphs compare the number of L2, L3 and TLB misses, and the execution time respectively, of the original and the improved GTC codes, as we vary the number of particles per cell on the `x` axis. We applied the transformations in the order in which they are presented in the graphs' legends. Notice how the code performance improved after each transformation. The large reduction in cache and TLB misses observed after the transposition of the `zion` arrays is due in part to a reduction in the number of unnecessary prefetches inserted by the Intel compiler, which was an unexpected side-effect, as well as because of an increase in data locality for other

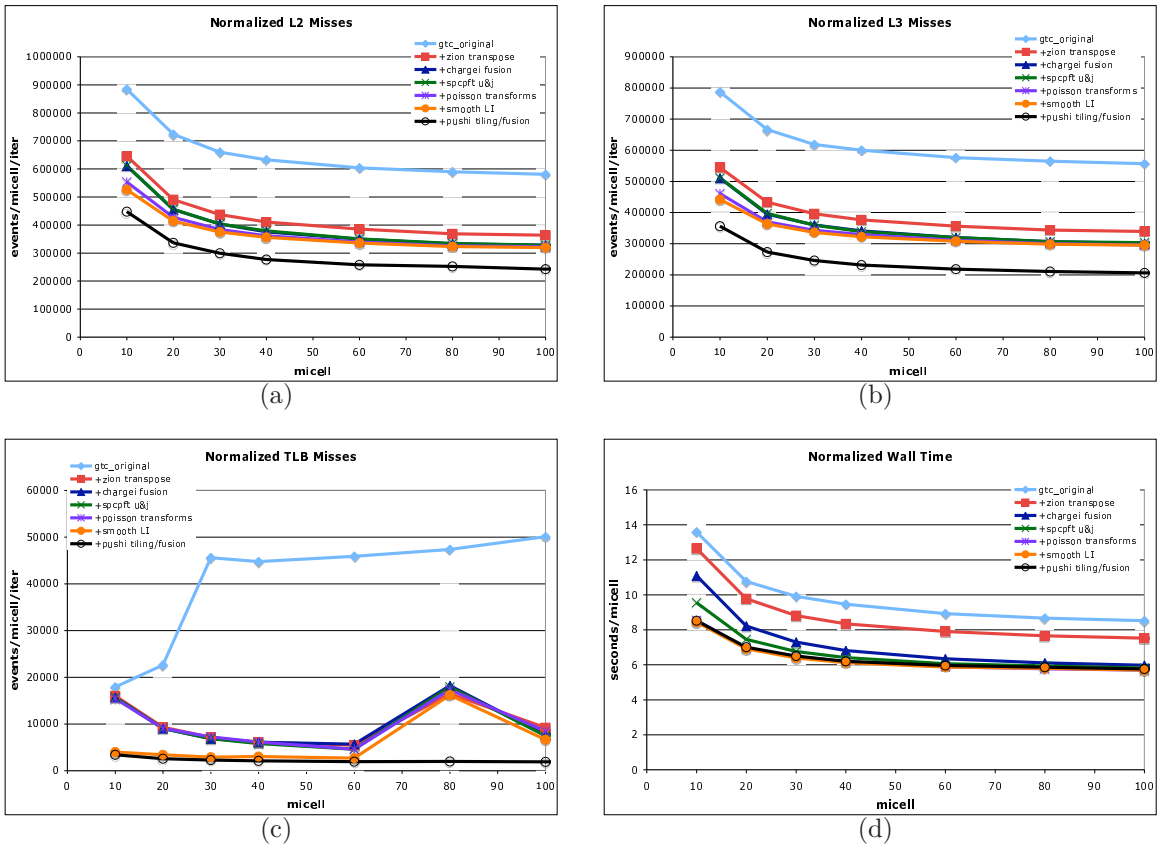


Figure 8.13: GTC performance after each code transformation on an Itanium2 system.

arrays after the loops working on the `zion` array had to stream through much less data because of the reduced fragmentation.

Performance improvements due to the transformations in `smooth`, `spcpft` and `poisson` are significant only when the number of particles is relatively small, since the amount of work in these routines is proportional to the number of cells in the poloidal plane and it does not depend on the number of particles.

Notice also how the tiling/fusion in the `pushi` routine significantly reduced the number of L2 and L3 cache misses, but these improvements did not translate into a smaller execution time. When we tiled & fused the loop nests in `pushi`, we created

a large loop over stripes that overflowed the small 16KB dedicated instruction cache on Itanium. Thus, the improvement in data locality was mitigated by an increase in the number of instruction cache misses. We expect this transformation to have a bigger impact on other architectures that have a larger instruction cache, including Montecito, the new member of the Itanium family of processors.

Overall, we reduced the number of cache misses by at least a factor of two, the number of TLB misses was reduced by a large margin, and we observed a 33% reduction of the execution time, which amounts to a 1.5x speedup.

Scopes	Carried_L2D		Carried_L3D		Carried_TLB	
Experiment Aggregate Metrics	3.11e08	100.0	2.36e08	100.0	6.17e06	100.0
loop at main.F90: 146-266	1.15e08	37.1%	1.15e08	48.7%	9.41e05	15.2%
loop at main.F90: 139-343	4.95e07	15.9%	4.95e07	21.0%	4.11e05	6.7%
pushi_	1.70e07	5.5%	1.70e07	7.2%	1.34e05	2.2%
loop at chargei.F90: 50-138	4.41e07	14.2%	1.65e07	7.0%	5.02e05	8.1%
loop at poisson.F90: 76-162	1.26e07	4.1%	1.20e07	5.1%	9.89e04	1.6%

Figure 8.14: Program scopes carrying the most cache misses in the modified code at 100 particles/cell.

We used measurements from several execution where the number of particles per cell was varied between 3 and 15, to build scalable models of edge execution frequencies and reuse distance histograms at reuse pattern level. From these models we predicted the performance of GTC for 100 particles/cell. Figure 8.14 presents the number of carried misses predicted by our toolkit for the modified code at 100 particles/cell. We notice that the two main loops carry about 70% of all L3 cache misses in the modified code. Routine `pushi` still carries around 7% of all L3 cache misses. The next most significant number of L3 cache misses are carried by the fused loop in `chargei` and those are due to irregular accesses to two arrays. Finally, the iterative

loop of the Poisson solver carries around of 5% of the L3 cache misses. While there are still some opportunities for improvement, the majority of the L3 cache misses are carried by the two main loops now, and they are intrinsic to the algorithm.

8.3.2 GTC Performance on the Opteron

Even though we modeled, predicted and tuned the performance of GTC for an Itanium2 architecture, we measured also the speedup of the transformed code on a Cray XD1 system with a dual core Opteron 275 CPU at 2.2 GHz. Each core has a 2 way set-associative 64 KB L1 cache, a 16 way set-associative 1MB L2 cache, a 32 entries full-associative L1 TLB, and a 512 entries 4 way set-associative L2 TLB. We compiled the codes using the PGI Fortran compiler 6.1-2 and the optimization flags *-r8 -gopt -fast -fastsse -Kieee*.

Figure 8.15 compares the measured numbers of L1, L2, TLB1 and TLB2 misses, as well as the measured wall clock times, of the original and the transformed GTC codes on an Opteron 275 machine. The graphs show a different picture than on the Itanium2 system. While we notice a halving of the L2 TLB misses, we notice a more than doubling of the L1 TLB misses, and a very modest decrease of the L1 and L2 cache misses. In fact, the number of L2 cache misses is lower than in the original program only after the tiling/fusion transformation in `pushi`. The transpose of the `zion` arrays resulted in a substantial increase in the number of L2 cache misses. All these numbers translate into a very modest reduction of the execution time after the `pushi` transformation, while we can notice a slight performance degradation with the other transformations.

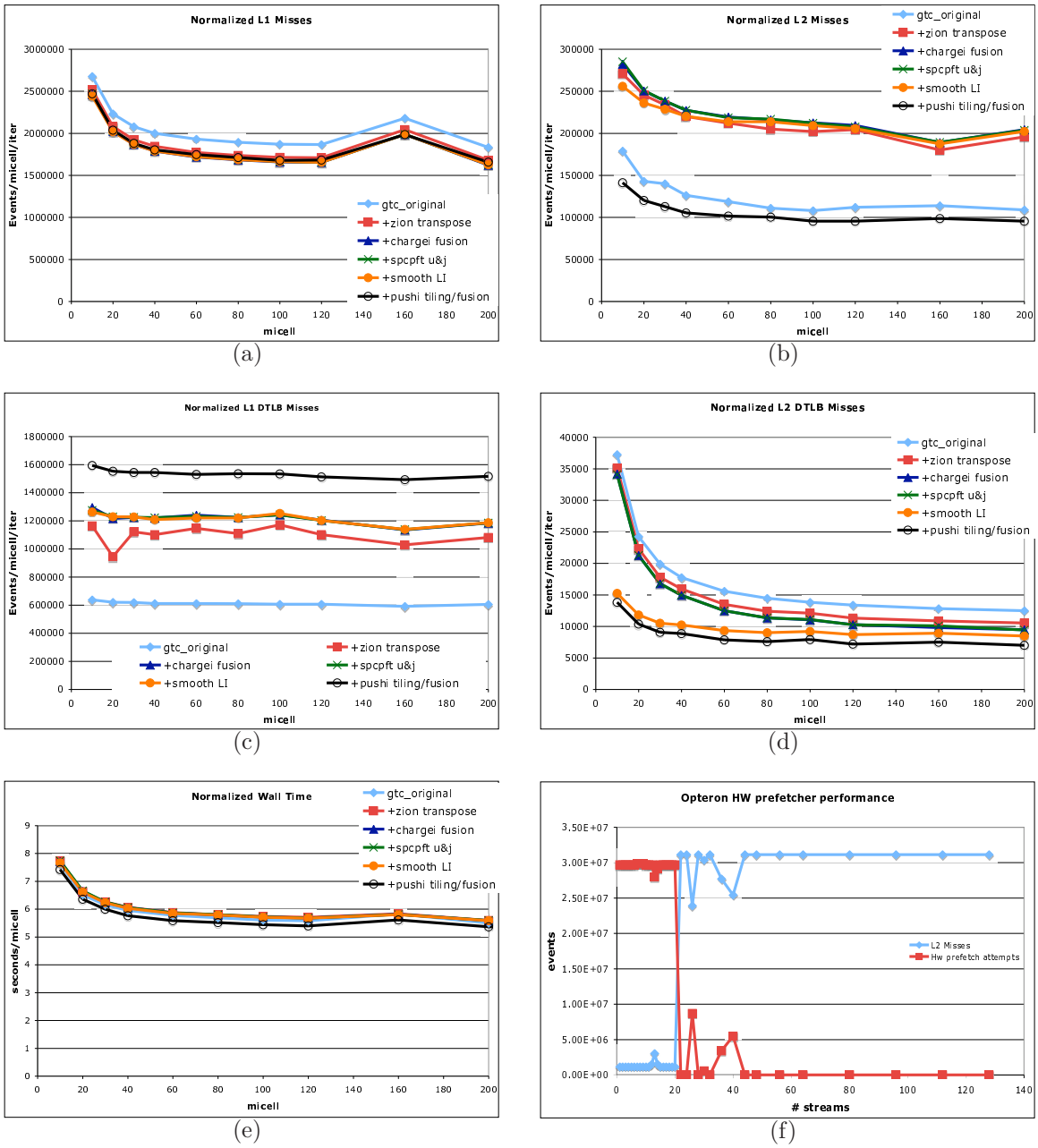


Figure 8.15: (a)-(e) Performance of the GTC code on an Opteron 275 machine; (f) performance of the Opteron HW prefetcher.

We measured the performance of the GTC codes on the Itanium2 and the Opteron using the same program inputs. Moreover, the two architectures have similar sized lower caches. The L3 cache on our Itanium2 system is 1.5 MB with a line size of 128

bytes. The L2 cache of each Opteron 275 core is 1 MB in size, and has a 64 bytes cache line. Thus, it looks like the Itanium2 has a small advantage with its slightly larger cache; also its larger cache line should translate into more spatial reuse. However, if we compare the absolute values measured on the two architectures, we notice that the number of L2 cache misses on the Opteron is lower in absolute terms than the number of L3 cache misses on the Itanium2. This holds true for the transformed codes, but especially for the original code.

The lower number of L2 cache misses measured on the Opteron is the result of the hardware prefetcher on that architecture fetching a good chunk of the data before it is needed by the application. The hardware prefetcher works by inspecting the streams of accesses to memory. When the hardware prefetcher notices that cache lines `i` and `i+1` are accessed, it initiates the fetch of line `i+3` into the L2 cache. The hardware prefetcher can keep track of multiple streams of data and it resets a stream when a page boundary is crossed. However, the hardware is provisioned for tracking a finite number of different streams. If the number of streams grows over that limit, the hardware prefetcher's performance drops.

Figure 8.15(f) shows the results of a micro-benchmark that we wrote to test the performance of the Opteron hardware prefetcher. The micro-benchmark executes a fixed number of stride one accesses to memory using a user specified number of parallel streams. We varied the number of streams from 1 to 128 and we used the CPU's hardware performance counters to measure the number of L2 cache misses and the number of hardware prefetch attempts. The results are very clear. When the number of streams grows over twenty, the efficiency of the hardware prefetcher

plummets.

By transposing the two `zion` arrays, we increased the number of parallel data streams in loops that were accessing more than one record field. Thus, while we improved data locality by reducing fragmentation in cache lines, the observed number of L2 cache misses on the Opteron is higher because a large number of memory accesses that were being hidden by the hardware prefetcher originally, are now exposed to the application. The effect is visible also when we look at the number of L1 TLB misses. With only 32 entries, the L1 TLB experiences a higher number of misses due to the increase in the number of parallel streams. Tuning for the Opteron requires not only improving data reuse at the application level, but also making sure that the number of parallel streams stays low to leverage the hardware prefetcher.

Because the hardware prefetcher tracks streams separately for each memory page, and since the number of parallel streams it can handle is relatively close to the number of entries in the L1 TLB cache, it means that optimizing for the hardware prefetcher is equivalent with optimizing for the L1 TLB. We used the memory reuse distance data to compute memory predictions for the Opteron cache architecture. Figure 8.16 shows the reuse pattern generating most L1 TLB misses on the Opteron. We notice the only loop of the `gcmotion` routine generates 76.7% of all L1 TLB misses.

By looking at the miss counts at variable level for this reuse pattern, we determined that in addition to the `zion` array, accessed through its alias `particle_array`, many other smaller arrays were being accessed in each iteration of this loop. Five of these arrays, declared in routine `pushi`, were being passed as arguments to `gcmotion` by the function call in routine `pushi`. The rest of them were arrays local to the "`C`"

Scopes	Reuse_L1TLB
Experiment Aggregate Metrics	2.82e08 100.0
▼ (expand)	2.16e08 76.7%
▶ alien [DEST: L,Lev 1,P:gcmotion_,escZ_march03.c[1656,1860]]	2.16e08 76.7%
alien [SRC: L,Lev 1,P:gcmotion_,escZ_march03.c[1656,1860]]	2.16e08 76.7%
▼ Carried By	
▼ alien [L,Lev 1,P:gcmotion_,escZ_march03.c[1656,1860]]	2.16e08 76.7%
sraq	4.08e07 14.5%
sra	2.11e07 7.5%
scalars	1.32e07 4.7%
particle_array	1.27e07 4.5%
B	1.26e07 4.5%
sw	1.26e07 4.5%
ghq	1.26e07 4.5%
gha	1.26e07 4.5%
dXgq	1.26e07 4.5%
dXgf	1.26e07 4.5%
dXgr	1.26e07 4.5%
Er_particle	1.26e07 4.5%
dXa	1.26e07 4.5%
wpi_array	1.25e07 4.4%

Figure 8.16: Reuse pattern generating most L1TLB misses on Opteron.

file containing routine `gcmotion`. We noticed that many of the small arrays were being accessed in similar patterns throughout routine `gcmotion`, as well as through the other routines of the "`C`" file.

We merged the five arrays in `pushi` into one array of structures. In the file containing `gcmotion`, we reorganized 14 arrays organized as 3 groups of four and one group of 2. Each group is linearized in a fashion that amounts to an array of structures. The effect of these transformations is a significant reduction in the number of parallel streams in all loops that are accessing them. The merging of the five arrays in `pushi` had the effect of reducing the number of streams in some loop nests of the `pushi` routine that were accessing them as well.

Figure 8.17 presents the performance of the GTC code on an Opteron 275 machine

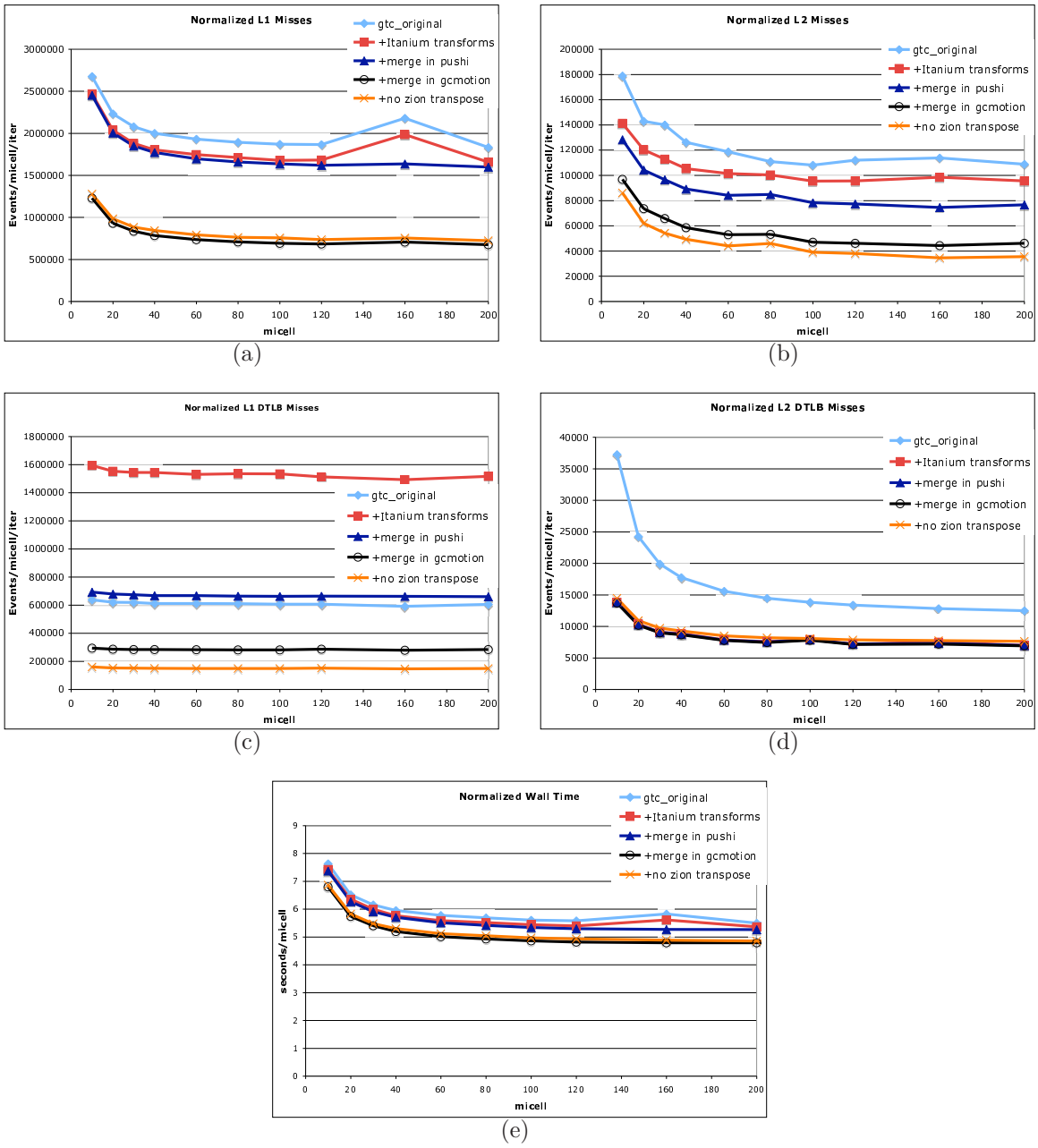


Figure 8.17: GTC performance on an Opteron 275 machine after additional data transformations.

after these additional data transformations. The five graphs compare the number of L1 cache, L2 cache, L1 TLB and L2 TLB misses, and the wall clock execution times respectively, of five different version of the code. The five versions of the code

include: 1) the original GTC code; 2) the code with all transformations targeted to the Itanium; 3) the code at 2) plus the merging of the five arrays in routine `pushi`; 4) reorganization of the 14 arrays in file containing `gcmotion` on top of all the previous transformations; and 5) the code with all previous transformations but the zion transpose.

As expected, the reorganization of data arrays in `pushi` and `gcmotion` has no effect on the large L2 TLB of the Opteron. However, removing the zion transpose optimization, increases slightly the number of L2 TLB misses due to the increase in data fragmentation.

Reducing the number of parallel streams has a significant effect on the number of L1 TLB and L2 cache misses, the latter ones being the result of increased effectiveness of the hardware prefetcher on the Opteron. We notice also that without the zion transpose transformation, the number of L1 TLB and L2 cache misses is even lower, since the number of parallel streams is reduced in many loops and the hardware prefetcher manages to hide most L2 cache misses produced by the increased data fragmentation.

Reducing the number of streams has an effect also on the number of L1 cache misses. We notice that merging the five arrays in the `pushi` routine has a minimal effect, since the L1 cache has significantly more blocks than the number of entries in the L1 TLB. However, the reorganization of the 14 arrays in `gcmotion` results in a very significant reduction of the L1 cache misses in that routine.

Overall, these new data transformations together with the previous transformations reduce the number of cache and TLB misses by at least a factor of two at all

levels of the Opteron's memory hierarchy, and the single node execution time is reduced by 13.5%. While the number of L1 TLB and L2 cache misses is somewhat smaller without the zion transpose, the number of L2 TLB misses is slightly larger and the overall execution time is slightly higher without the zion transpose. Moreover, the zion transpose has much higher benefits on other platforms that do not rely on a hardware prefetcher, plus it has the potential of increasing opportunities for vectorization on architectures with a more complete set of vector instructions. On the Opteron, telling the PGI compiler to use SSE instructions had no effect on the running time of GTC.

8.3.3 Itanium2 vs. Opteron Running the GTC Code

Comparing the performance of GTC on the Itanium2 at 900 MHZ (see Figure 8.13) and on the Opteron 275 at 2.2 GHz (see Figure 8.17), we notice that the number of L2 cache misses on the Opteron is significantly lower than the number of L3 cache misses on the Itanium2. This is due to the hardware prefetcher on the Opteron hiding a large number of memory accesses. At the same time, 100 time steps of GTC with 100 particles/cell take 561 seconds on the Itanium2 at 900 MHz and 486 seconds on the Opteron at 2.2 GHz. Clock for clock, the Itanium2 is 2.1 times more efficient than the Opteron on this code.

Part of the difference in efficiency can be explained by the fact that memory latencies measured in terms of CPU clock cycles, are going to be lower on CPUs that run at a lower frequency. We noticed however that cutting the number of cache and TLB misses by more than a factor of two, resulted in a reduction of the execution

time of only 13.5% on the Opteron. This suggests that the performance of GTC on the Opteron is not as much memory bounded as it is computation and control flow limited. The Itanium2 can issue up to six instructions every clock cycle, and floating-point arithmetic operations, especially division and square root, are executed more efficiently on the Itanium than on the Opteron. As a result, we observed a higher speed-up on the Itanium2 when we reduced the number of cache and TLB misses by similar factors.

Moreover, we expect the fusion/tiling transformation in routine `pushi` to have a bigger impact on an Itanium CPU with a larger dedicated instruction cache, such as Montecito. At the same time, we expect the new K10 Opterons to run the GTC code more efficiently clock for clock, due to general improvements to the Opteron core and the wider and faster SSE path.

Chapter 9

Conclusions

At the beginning of this thesis we emphasized the importance of automatic performance analysis and modeling tools for understanding scalability of applications, identifying performance bottlenecks and opportunities for tuning, guiding mapping of application components to heterogeneous resources, and providing insight into the design of custom architectures. Over the course of several chapters, we presented strategies for scalable and cross-architecture performance predictions, and we presented analysis techniques and new performance metrics that provide insight into performance bottlenecks and guide application tuning.

This thesis tackles two important problems in the field of performance analysis: (1) semi-automatic construction of application performance models that enable scalable and cross-architecture performance predictions; (2) automatic understanding and classification of application performance bottlenecks and estimation of their impact on the execution time.

This thesis describes an approach based on separating the contribution of application characteristics from the contribution of the architecture characteristics to overall application performance. We describe a methodology for constructing models of an

application's characteristics parameterized by problem size or other input parameters. The benefits of this approach are twofold. First, modeling application specific factors in isolation yields architecture-neutral models that can predict execution characteristics on different platforms. Second, models that describe algorithmic and application choices are typically monotonic polynomial functions; in general, models based on measurements on a particular architecture are not. To predict how an application performs on a different architecture, we map the application's characteristics onto a model of the target architecture represented by a description of the resources available on the machine.

Based on this, the thesis describes performance analysis strategies for understanding the mismatch between application and architecture characteristics, which enables us to identify certain types of application performance bottlenecks and to estimate the potential performance improvement from correcting them. The rest of this chapter summarizes the contributions of the thesis, discusses the limitations of this work and the remaining open issues.

9.1 Contributions

An algorithm for modeling memory reuse distance histograms

Section 5.2 describes an algorithm for modeling the structure and scaling of memory reuse distance histograms as a function of problem size. Scalable models of memory reuse distance histograms enable us to understand the distribution of reuse distances for program inputs that we did not analyze directly. Other approaches described in

the literature use a fixed strategy to divide the reuse distance data into bins across all problem sizes, independent of how the reuse distance values are distributed. In contrast, we automatically identify an appropriate number of bins and their bounds to accurately represent the data for each reference across all problem sizes. In addition, we identified accuracy issues due to code optimizations such as loop peeling and unrolling, when models are constructed at reference level from data collected using non-unit block sizes. We describe a solution based on static-analysis-driven aggregation of histograms corresponding to related references.

Cache miss predictions for set-associative caches

Memory reuse distance models translate directly into predictions of misses for fully-associative caches with LRU-like replacement policies. In section 5.3.2 we describe a probabilistic algorithm that uses memory reuse distance data to predict the number of misses for caches with arbitrary associativity levels. Using this model, we accurately predicted the number of cache and TLB misses for the ASCI Sweep3D code and several NAS benchmarks, for a large range of problem sizes and several cache architectures.

Cross-architecture performance predictions

In Chapter 6 we introduce a machine description language (MDL) to model the main architecture characteristics affecting instruction scheduling and instruction latencies. Among other constructs, the MDL enables us to define replacement rules to account for differences in the instruction sets of different architectures. Based on our experience, replacement rules are essential for producing accurate cross-architecture

predictions. Next, we describe a generic modulo instruction scheduler that is initialized with a model of a target architecture. The scheduler enables us to compute predictions of application execution costs for arbitrary target machines. We present an algorithm for finding hierarchical *super-structures* in dependence graphs, where by *super-structure* we mean a sub-graph with unique entry and exit nodes. These *super-structures* can be logically reduced to a single edge, thus reducing the complexity of large dependence graphs. This approach enables us to compute modulo instruction schedules for loops that contain a large number of implicit recurrences, such as loops with multiple inner loops and function calls.

Understanding performance bottlenecks

Computing meaningful cross-architecture performance predictions requires a good understanding of an application's characteristics, and of how these application characteristics map onto the resources of a target machine. Chapter 7 describes analysis strategies for identifying several types of application performance bottlenecks. Based on this analysis, we compute performance metrics that guide tuning by highlighting the factors that limit performance at various points in a program. Our instruction scheduler decomposes an application's execution cost into memory penalty time, application dependence time, and resource contention time, based on insight gained during the instruction scheduling process. We introduce two performance metrics that quantify the potential for improvement from increased fine-grain parallelism in an application, or from additional machine resources, respectively. Next, we describe data locality analysis techniques based on collecting and modeling reuse distance

data separately for each reuse pattern of a reference. This approach uncovers the most significant reuse patterns contributing to an application's cache miss counts, and identifies program transformations that have the potential to improve memory hierarchy utilization. We describe also a static analysis algorithm that identifies opportunities for improving spatial locality through data layout transformations.

Model guided application tuning

Chapter 8 describes the process of tuning three scientific applications based on the performance metrics computed by our toolkit. For Sweep3D, we identified a loop with high potential for improvement from increased fine-grain parallelism. Transforming the loop increased overall application performance by 16% on an Itanium2 based machine. Next, we identified a loop that carried 75% of all L2 cache misses and 68% of all L3 cache misses in the program. The insight gained from understanding the reuse patterns contributing the largest number of cache misses in the program, enabled us to transform the code to increase temporal data locality. The transformed code incurs less than 25% of the cache misses observed with the original code. After all code transformations the overall execution is faster by a factor of three.

For the Parallel Ocean Program (POP) from Los Alamos National Laboratory, we identified routines with poor instruction-level parallelism (ILP), and correctly predicted the potential for improving them. Transforming the code to increase ILP yielded 4% better performance.

For the Gyokinetic Toroidal Code (GTC) from Princeton Plasma Physics Laboratory, our analysis identified two arrays of structures that were being accessed with a

non-unit stride, which almost doubled number of cache misses to these arrays above ideal. We also identified the main loops carrying cache and TLB misses. Reorganizing the arrays of structures into structures of arrays, and transforming the code to shorten the reuse distance of some of the significant reuse patterns, reduced cache misses by a factor of two and execution time by 33% on an Itanium2 machine. When running the transformed GTC code on an Opteron machine, we realized that our data transformations for reducing cache line fragmentation were increasing the number of parallel data streams in some loop nests beyond the capabilities of the Opteron's hardware prefetcher. The result was that a large number of cache misses could not be hidden by the hardware prefetcher. The additional cache misses exposed to the application negated any improvements in data locality produced by our code transformations. We found that on an Opteron, optimizing for the hardware prefetcher was important for performance and we determined that optimizing for the hardware prefetcher was equivalent to optimizing for the small L1 TLB. As a result, we identified a set of data arrays experiencing a large number of L1 TLB misses; we reorganized these arrays to reduce the number of parallel data streams in key loops. The end result of these transformations was a reduction of cache and TLB misses by at least 50% and a 13.5% reduction in execution time on the Opteron.

9.2 Limitations and Open Issues

Predictions of memory penalty

Currently, when we compute predictions of execution time, we assume that the full

latency penalty for cache misses is exposed. This assumption is not unreasonable in cases when data prefetching is disabled (see section 6.5) because there is little memory parallelism in many applications. However, some applications are optimized to overlap their accesses to memory by clustering together memory reads that are likely to miss in cache [52]. We have a partial implementation of an algorithm for understanding the built-in memory parallelism in applications. Working on a loop's dependence graph, we compute maximal cliques of independent load instructions using the algorithm presented in [10]. At the same time, we group references into *reuse groups*, where two references are considered part of a reuse group if our analysis of symbolic access formulas describing their access patterns determines that they access same cache lines. For each reuse group we determine the leading reference as the reference to first access a new memory block. Currently, we compute an intrinsic memory parallelism factor at loop level, based on the computed cliques of independent loads, but assuming that all references go to memory. To compute the observed memory parallelism factor, we must consider cliques formed only by leading references of reuse groups and we must take into account their predicted miss rate values. However, when miss rates are less than 100%, an open issue is understanding how frequently independent references miss in the same iteration of a loop.

Cross-compiler performance predictions

Accurate predictions of application execution costs for a target architecture on which we did not perform any direct measurements is one of the most difficult tasks in the area of performance analysis. We compute cross-architecture performance predic-

tions by analyzing and modeling application characteristics that are independent of the target architecture. However, code optimizations can greatly affect performance as seen in Chapter 8. High level loop transformations restructure the code to improve data locality or to increase the amount of instruction-level parallelism in inner loops. By changing the execution order of a program, these transformations alter an application's memory access patterns and its instruction schedule dependences; this effectively changes the application-centric factors that are modeled by our toolkit. When trying to perform cross-architecture performance predictions and validate them using different compilers on each platform, significant differences in compiler capabilities make accurate predictions hard. Thus, our cross-architecture prediction results are based on the assumption that the computation order and memory access patterns on the target architecture are similar.

Predictions in the presence of special architectural or OS features

Special architectural features can also affect the accuracy of performance predictions. For example, when we computed predictions of cache and TLB misses for the NAS LU benchmark on an Itanium2 machine, we observed more L2 cache misses during an actual run than the values predicted by our toolkit. We found that on the Itanium running Linux, operating system's page tables are cacheable in the L2 cache. The effect is that a full cache line of page table entries is loaded in the L2 cache on a TLB miss; this has the potential of lowering the average TLB miss penalty when successive TLB misses to neighboring pages are serviced. However, if an application accesses memory with a large stride, larger than the size of a memory page times

the number of page table entries that fit into a cache line, each TLB miss will have to go to L3 or to memory, causing an L2 cache miss. Our analysis does not predict these cache misses because they are produced by the operating system on behalf of the application, not by the application directly. Also, hardware prefetching schemes, such as on the Opteron, are effective at fetching data in advance for streams of strided accesses. Our analysis does not account for hardware or software prefetching schemes at this time. However, prefetching algorithms implemented in hardware are not very complex, and we plan to explore predicting their effects through a combination of static and dynamic analysis.

Predictions for other architectures

At this time, we compute predictions of execution time for Itanium2 and MIPS R12000 architectures. Computing predictions for architectures based on other processors requires building models of those architectures using our machine description language. Because our predictions are based on analysis of SPARC binaries that use a RISC instruction set, our prediction work was focused on architectures with similar RISC instruction sets. While the more recent x86 processors internally execute micro-operations that are similar to RISC instructions, the instruction set used by their front-end is based on a more complex CISC format. Computing predictions of execution time for x86 architectures from analysis of SPARC binaries is still an open issue.

Bibliography

[1] V. S. Adve, R. Bagrodia, J. C. Browne, E. Deelman, A. Dubeb, E. N. Houstis, J. R. Rice, R. Sakellariou, D. Sundaram-Stukel, P. T. Teller, and M. K. Vernon. POEMS: End-to-end performance design of large parallel adaptive computational systems. *Software Engineering*, 26(11):1027–1048, 2000.

[2] R. Allen and K. Kennedy. *Optimizing compilers for modern architectures: a dependence-based approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[3] G. Ammons, T. Ball, and J. R. Larus. Exploiting hardware performance counters with flow and context sensitive profiling. *SIGPLAN Not.*, 32(5):85–96, 1997.

[4] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS parallel benchmarks 2.0. Technical Report NAS-95-020, NASA Ames Research Center, Dec. 1995.

[5] V. Bala, E. Duesterwald, and S. Banerjia. Dynamo: a transparent dynamic optimization system. *ACM SIGPLAN Notices*, 35(5):1–12, 2000.

[6] T. Ball and J. R. Larus. Optimally profiling and tracing programs. *ACM Transactions on Programming Languages and Systems*, 16(4):1319–1360, July 1994.

[7] B. Bennett and V. Kruskal. LRU stack processing. *IBM Journal of Research and Development*, 19(4):353–357, July 1975.

[8] K. Beyls and E. D'Hollander. Reuse distance as a metric for cache behavior. In *IASTED conference on Parallel and Distributed Computing and Systems 2001 (PDCS01)*, pages 617–662, 2001.

[9] K. Beyls and E. H. D'Hollander. Intermediately executed code is the key to find refactorings that improve temporal data locality. In *CF '06: Proceedings of the 3rd Conference on Computing Frontiers*, pages 373–382, New York, NY, USA, 2006. ACM Press.

[10] C. Bron and J. Kerbosch. Algorithm 457: finding all cliques of an undirected graph. *Communications of the ACM*, 16(9):575–577, September 1973.

[11] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *The International Journal of High Performance Computing Applications*, 14(4):317–329, Winter 2000.

[12] D. Callahan, J. Cocke, and K. Kennedy. Estimating interlock and improving balance for pipelined architectures. *Journal of Parallel and Distributed Computing*, 5(4):334–358, 1988.

[13] L. N. Chakrapani, J. Gyllenhaal, W. mei W. Hwu, S. A. Mahlke, K. V. Palem, and R. M. Rabbah. Trimaran: An infrastructure for research in instruction-level parallelism. In *Lecture Notes in Computer Science*, volume 3602, pages 32–41. Springer-Verlag, 2005.

[14] T. M. Chilimbi, B. Davidson, and J. R. Larus. Cache-conscious structure definition. In *PLDI '99: Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, pages 13–24, New York, NY, USA, 1999. ACM Press.

[15] K. Cooper, T. Harvey, and K. Kennedy. A simple, fast dominance algorithm. Technical Report TR06-38870, Dept. of Computer Science, Rice University, 2001.

[16] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1990.

[17] C. Ding and Y. Zhong. Reuse distance analysis. Technical Report TR741, Dept. of Computer Science, University of Rochester, 2001.

[18] P. C. Diniz and J. Abramson. SLOPE: A compiler approach to performance prediction and performance sensitivity analysis for scientific codes. *Cyberinfrastructure Technology Watch: Special Issue on HPC Productivity*, 2(4B), Nov. 2006. `http://www.ctwatch.org`.

[19] DOE Accelerated Strategic Computing Initiative. The ASCI Sweep3D Benchmark Code. `http://www.llnl.gov/asci_benchmarks/asci/limited/sweep3d/asci_sweep3d.html`.

[20] C. Fang, S. Carr, S. Onder, and Z. Wang. Reuse-distance-based Miss-rate Prediction on a Per Instruction Basis. In *The Second ACM SIGPLAN Workshop on Memory System Performance*, Washington, DC, USA, June 2004.

[21] S. Ghosh and M. M. amd Sharad Malik. Cache miss equations: An analytical representation of cache misses. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, pages 317–324, Vienna, Austria, July 1997.

[22] R. Hastings and B. Joyce. Purify: Fast detection of memory leaks and access errors. In *Proceedings of the Winter USENIX*, 1992.

[23] M. D. Hill and A. J. Smith. Evaluating associativity in cpu caches. *IEEE Trans. Comput.*, 38(12):1612–1630, 1989.

[24] A. Hoisie, O. Lubeck, H. Wasserman, F. Petrini, and H. Alme. A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs. In *Proceedings of the 2000 International Conference on Parallel Processing*, 2000.

[25] C. Hristea, D. Lenoski, and J. Keen. Measuring memory hierarchy performance of cache-coherent multiprocessors using micro benchmarks. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing (CDROM)*, pages 1–12. ACM Press, 1997.

[26] R. A. Huff. Lifetime-sensitive modulo scheduling. In *PLDI '93: Proceedings of the ACM SIGPLAN 1993 Conference on Programming Language Design and Implementation*, pages 258–267, New York, NY, USA, 1993. ACM Press.

[27] Intel Corporation. Intel VTune performance analyzers. `http://www.intel.com/software/products/vtune/`.

[28] Intel Corporation. *Intel Itanium2 Processor Reference Manual for Soft-*

ware Development and Optimizations, 2003. `http://www.intel.com/design/itanium2/manuals/251110.htm`.

[29] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. Introduction to the cell multiprocessor. *IBM J. Res. Dev.*, 49(4/5):589–604, 2005.

[30] D. E. Knuth and F. R. Stevenson. Optimal measurement points for program frequency counts. *BIT*, 13(3):313–322, 1973.

[31] M. S. Lam and R. P. Wilson. Limits of control flow on parallelism. In *ISCA '92: Proceedings of the 19th Annual International Aymposium on Computer Architecture*, pages 46–57, New York, NY, USA, 1992. ACM Press.

[32] J. Larus and E. Schnarr. EEL: Machine-Independent Executable Editing. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 291–300, June 1995.

[33] A. Lebeck and D. Wood. Cache Profiling and the SPEC Benchmarks: A Case Study. *IEEE Computer*, 27(10):15–26, Oct. 1994.

[34] W. W. Lee. Gyrokinetic approach in particle simulation. *Physics of Fluids*, 26:556–562, Feb. 1983.

[35] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Trans. Program. Lang. Syst.*, 1(1):121–141, 1979.

[36] Los Alamos National Laboratory. The Parallel Ocean Program (POP). `http://climate.lanl.gov/Models/POP`.

[37] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: building customized program analysis tools with dynamic instrumentation. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 190–200, New York, NY, USA, 2005. ACM.

[38] Y. Luo, O. M. Lubeck, H. Wasserman, F. Bassetti, and K. W. Cameron. Development and validation of a hierarchical memory model incorporating cpu- and memory-operation overlap model. In *WOSP '98: Proceedings of the 1st International Workshop on Software and Performance*, pages 152–163, New York, NY, USA, 1998. ACM Press.

[39] A. Mandal, K. Kennedy, C. Koelbel, G. Marin, B. Liu, L. Johnsson, and J. Mellor-Crummey. Scheduling strategies for mapping application workflows onto the grid. In *14th IEEE Symposium on High Performance Distributed Computing (HPDC 2005)*. IEEE Computer Society Press, 2005.

[40] G. Marin. Semi-Automatic Synthesis of Parameterized Performance Models for Scientific Programs. Master's thesis, Dept. of Computer Science, Rice University, Houston, TX, Apr. 2003.

[41] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems*, pages 2–13. ACM Press, 2004.

[42] G. Marin and J. Mellor-Crummey. Scalable cross-architecture predictions of memory hierarchy response for scientific applications. *Proceedings of the Los Alamos Computer Science Institute Sixth Annual Symposium*, 2005. `http://www.cs.rice.edu/~mgabi/papers/MM-lacsi05.pdf`.

[43] MathWorks. *Optimization Toolbox: Function quadprog*. `http://www.mathworks.com/access/helpdesk/help/toolbox/optim/quadprog.shtml`.

[44] R. Mattson, J. Gecsei, D. Slutz, and I. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[45] J. Mellor-Crummey. Using hpcviewer to browse performance databases, Feb. 2004. `http://www.hipersoft.rice.edu/hpctoolkit/tutorials/Using-hpcviewer.pdf`.

[46] J. Mellor-Crummey, V. Adve, B. Broom, D. Chavarrla-Miranda, R. Fowler, G. Jin, K. Kennedy, and Q. Yi. Advanced Optimization Strategies in the Rice dHPF Compiler. *Concurrency and Computation: Practice and Experience*, Volume 14(8-9):pp. 741–767, 2002.

[47] J. Mellor-Crummey, R. Fowler, and D. Whalley. Tools for application-oriented performance tuning. In *Proceedings of the International Conference on Supercomputing (ICS'01)*, pages 154–165, Sorrento, Italy, June 2001.

[48] J. Mellor-Crummey, R. J. Fowler, G. Marin, and N. Tallent. HPCVIEW: A Tool for Top-down Analysis of Node Performance. *The Journal of Supercomputing*, 23(1):81–104, 2002.

[49] J. Mellor-Crummey, D. Whalley, and K. Kennedy. Improving memory hierarchy performance for irregular applications using data and computation reorderings. *International Journal of Parallel Programming*, 29(3):217–247, 2001.

[50] F. Mueller, T. Mohan, B. de R. Supinski, S. A. McKee, and A. Yoo. Partial data traces: Efficient generation and representation. In *Workshop on Binary Translation*, IEEE Technical Committee on Computer Architecture Newsletter, Oct. 2001.

[51] The OProfile website. `http://oprofile.sourceforge.net/docs`.

[52] V. S. Pai and S. V. Adve. Code transformations to improve memory parallelism. In *International Symposium on Microarchitecture MICRO-32*, pages 147–155, Nov 1999.

[53] D. A. Patterson and J. L. Hennessy. *Computer architecture: a quantitative approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1990.

[54] U. Prestor. Evaluating the memory performance of a ccNUMA system. Master's thesis, Dept. of Computer Science, University of Utah, Salt Lake City, UT, Aug. 2001.

[55] R. Prosser. Applications of boolean matrices to the analysis of flow diagrams. In *Eastern Joint Computer Conference*, pages 133–138, Spartan Books, NY, USA, Dec 1959.

[56] P. Ranganathan, V. S. Pai, H. Abdel-Shafi, and S. V. Adve. The Interaction of Software Prefetching with ILP Processors in Shared-Memory Systems. In

ISCA '97: Proceedings of the 24th Annual International Symposium on Computer Architecture, pages 144–156, New York, NY, USA, 1997. ACM.

[57] B. R. Rau. Iterative modulo scheduling: an algorithm for software pipelining loops. In *MICRO 27: Proceedings of the 27th Annual International Symposium on Microarchitecture*, pages 63–74, New York, NY, USA, 1994. ACM Press.

[58] T. Romer, G. Voelker, D. Lee, A. Wolman, W. Wong, H. H. Levy, and B. Bershad. Instrumentation and optimization of Win32/Intel executables using Etch. In *Proceedings of the USENIX Windows NT Workshop*, pages 1–7, Seattle, WA, USA, Aug. 1997.

[59] A. Snavely, N. Wolter, and L. Carrington. Modeling application performance by convolving machine signatures with application profiles. In *IEEE 4th Annual Workshop on Workload Characterization*, pages 149–156, Washington, DC, USA, 2001. IEEE Computer Society.

[60] R. Srinivasan and O. Lubeck. MonteSim: a Monte Carlo performance model for in-order microachitectures. *SIGARCH Comput. Archit. News*, 33(5):75–80, 2005.

[61] A. Srivastava and A. Eustace. Atom: A system for building customized program analysis tools. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 196–205, Orlando, FL, May 1994.

[62] D. Sundaram-Stukel and M. K. Vernon. Predictive analysis of a wavefront application using LogGP. In *PPoPP '99: Proceedings of the Seventh ACM SIGPLAN*

Symposium on Principles and Practice of Parallel Programming, pages 141–150, New York, NY, USA, 1999. ACM.

[63] R. E. Tarjan. Testing flow graph reducibility. *Journal of Computer and System Sciences*, 9:355–365, 1974.

[64] N. Wichmann, M. Adams, and S. Ethier. New Advances in the Gyrokinetic Toroidal Code and Their Impact on Performance on the Cray XT Series, May 2007. The Cray User Group, CUG 2007.

[65] Y. Zhong, S. G. Dropsho, and C. Ding. Miss Rate Prediction across All Program Inputs. In *Proceedings of International Conference on Parallel Architectures and Compilation Techniques*, New Orleans, Louisiana, Sept. 2003.

[66] Y. Zhong, M. Orlovich, X. Shen, and C. Ding. Array regrouping and structure splitting using whole-program reference affinity. In *PLDI '04: Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, pages 255–266, New York, NY, USA, 2004. ACM Press.

Appendix A

Enhancements to the EEL Library

Our binary analysis tools are built on top of the Executable Editing Library (EEL) [32]. The EEL library was released in 1995 and it has not been publicly maintained since then. Since both the Sun Forte compiler and the GNU gcc compiler were updated constantly in this time, we had to bring EEL up to date on understanding control flow mechanisms used by today's compilers.

Special effort was put into recovering the control flow of indirect jumps. We greatly extended the previous method for correctly understanding the dispatched indirect jump instructions which account for the vast majority (often 100%) of indirect jumps in a program. Also, we replaced the old method for dealing with register indirect jumps which considered the target address computed along one path only. The new improved method is a fix point algorithm that checks each new path leading to the jump instruction and adjusts the set of possible targets until no new path is discovered.

Additional extensions of the static analysis capabilities of the EEL library include support for V8plus binaries and an improved method for identifying "*hidden routines*" in the *.text* segment, in the sense that the new method produces much fewer false positives. EEL uses the term *hidden routine* to refer to pieces of code that are neither

identified as routines in the symbol table, nor reachable from the nearest preceding routine entry specified in the symbol table. Since compilers include a large number of read-only data in the *.text* segment, there is a high chance that some words of data will have the same binary representation as valid instructions. EEL used to mistakenly consider many such words of data as *hidden routines*. We conduct some more checks before we classify sequences of seemingly valid instructions as *hidden routines*.

While all the extensions enumerated above improve the quality and the accuracy of the results produced by the static analysis, these extensions have a more important contribution at increasing the robustness of the edited binaries produced by the EEL library. A better understanding of the control flow in programs can only result in more stable instrumented binaries.

EEL permits addition and deletion of code at arbitrary locations in the program. Since the size of the text segment cannot be changed as other sections in the binary follow at addresses right after the end of the *text* segment, EEL creates a completely new section named *.edited_text* which is placed after all the other sections in the executable. The new section includes the code from the original *.text* section modified as wanted, and it can grow as large as needed without affecting any other existing sections. The principal challenge for having such a powerful capability, as moving the code to a different location, is to update all the control flow instructions to transfer the control flow to the correct instruction in the new section. For this reason, understanding the original control flow correctly is such a necessary task.

Unfortunately, the target of some indirect jumps cannot be determined using

static analysis. For instance, in the "C" language, function pointers can be stored into variables or passed as arguments to other functions. A complex and costly inter-procedural analysis may determine the target of some of these pointers. EEL implements a simpler solution that uses the space occupied by the old *.text* section to store a translation table. Every instruction in the original section is replaced with an unconditional annulled branch instruction that transfers control to the new location of the instruction. With this mechanism, whenever control flow reaches the old location of a routine entry because of an "unknown" indirect jump, the unconditional branch will transfer the control back to the correct instruction. But all relative branch instructions have a limited range. If the distance between the old and the new address of an instruction is greater than the spanning range of the available unconditional branch instruction, the translation table will contain only the address where the instruction is now located. To accommodate these situations, EEL replaces the indirect jumps whose target cannot be determined, with a code patch that uses the translation table to compute the destination target at run-time.

The above solution works well in practice most of the time. However, occasionally, unpatched code from system shared libraries invokes a routine from the original *.text* segment. Such cases occur in the finalization code of Fortran programs on SPARC machines. We modified the EEL mechanism to handle also such cases. Instead of storing in the translation table the address of the new location of every instruction in the old *.text* section, for routine entries we place a sequence of three valid instructions that act as a trampoline and can transfer control to any place in the address space. Although the trampoline code overwrites the entries corresponding to the second and

the third instructions of a routine, we consider that in practice, unknown indirect jumps and control transfer instructions from system shared libraries do not jump to the second or the third instruction of a routine. Until now, we did not encounter any case in which this assumption did not hold.

Finally, we added support in the EEL library to modify the *.dynamic* section of a program to include an arbitrary number of new shared library names that must be loaded when the program starts. No binding of the global symbols from the new libraries is performed by the loader. However, the code in the special initialization function *_init* is executed when the library is loaded. In the case of a user compiled shared library, the initialization routine can be replaced with custom code. The code in the initialization function can export the addresses of the desired symbols in a special area of the program's address space that is reserved in the instrumented binary. We use this mechanism to force instrumented programs load the *libdl.so* library and export the entry addresses of *dlopen* and *dlsym* routines to the profiling code.

Appendix B

Routine Level Cache Miss Predictions on IA64

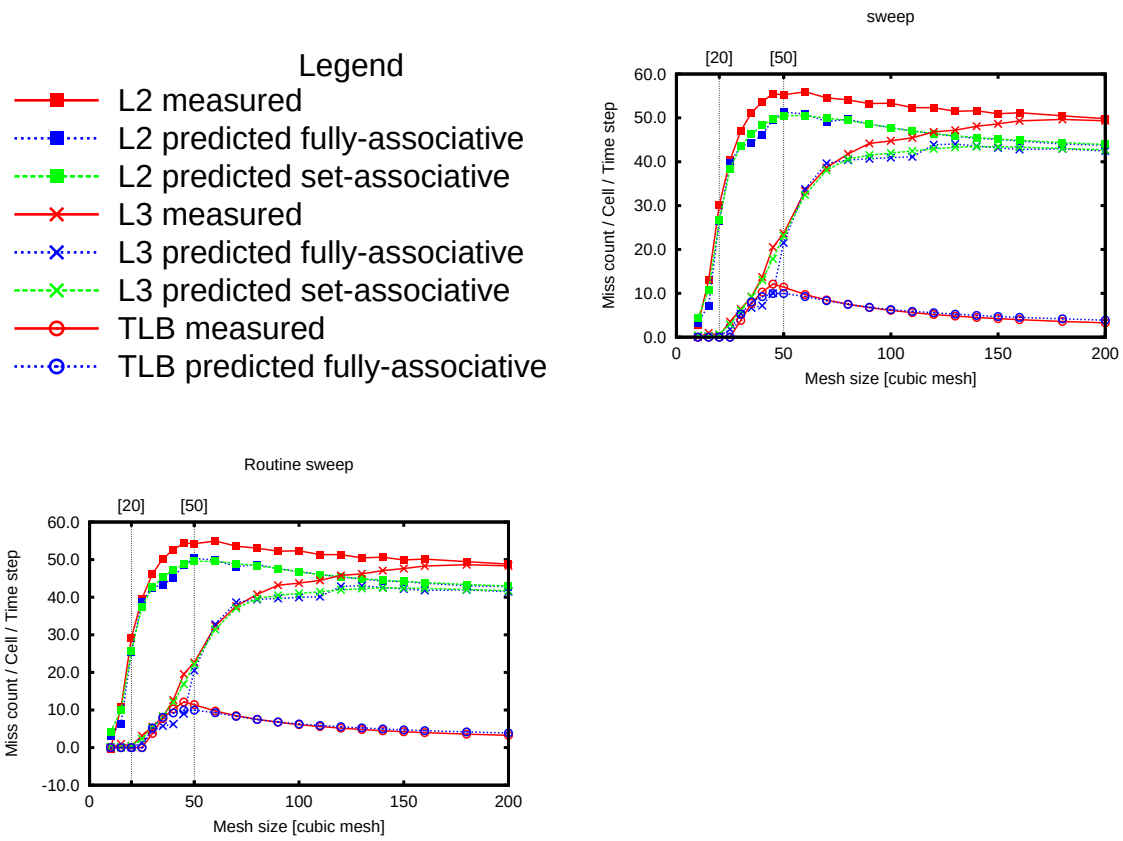


Figure B.1: Routine-level memory predictions for sweep on IA64

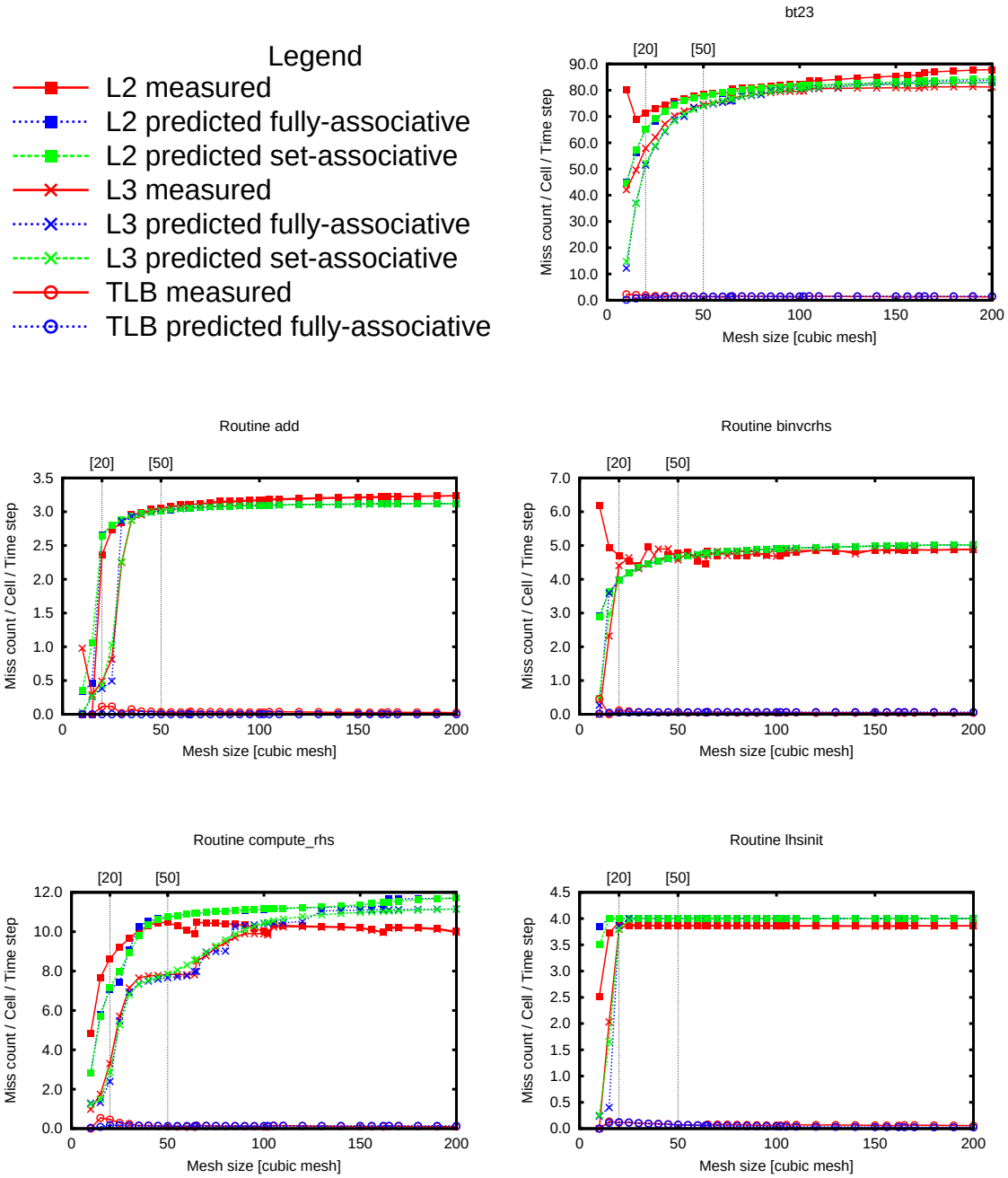


Figure B.2: Routine-level memory predictions for bt23 on IA64

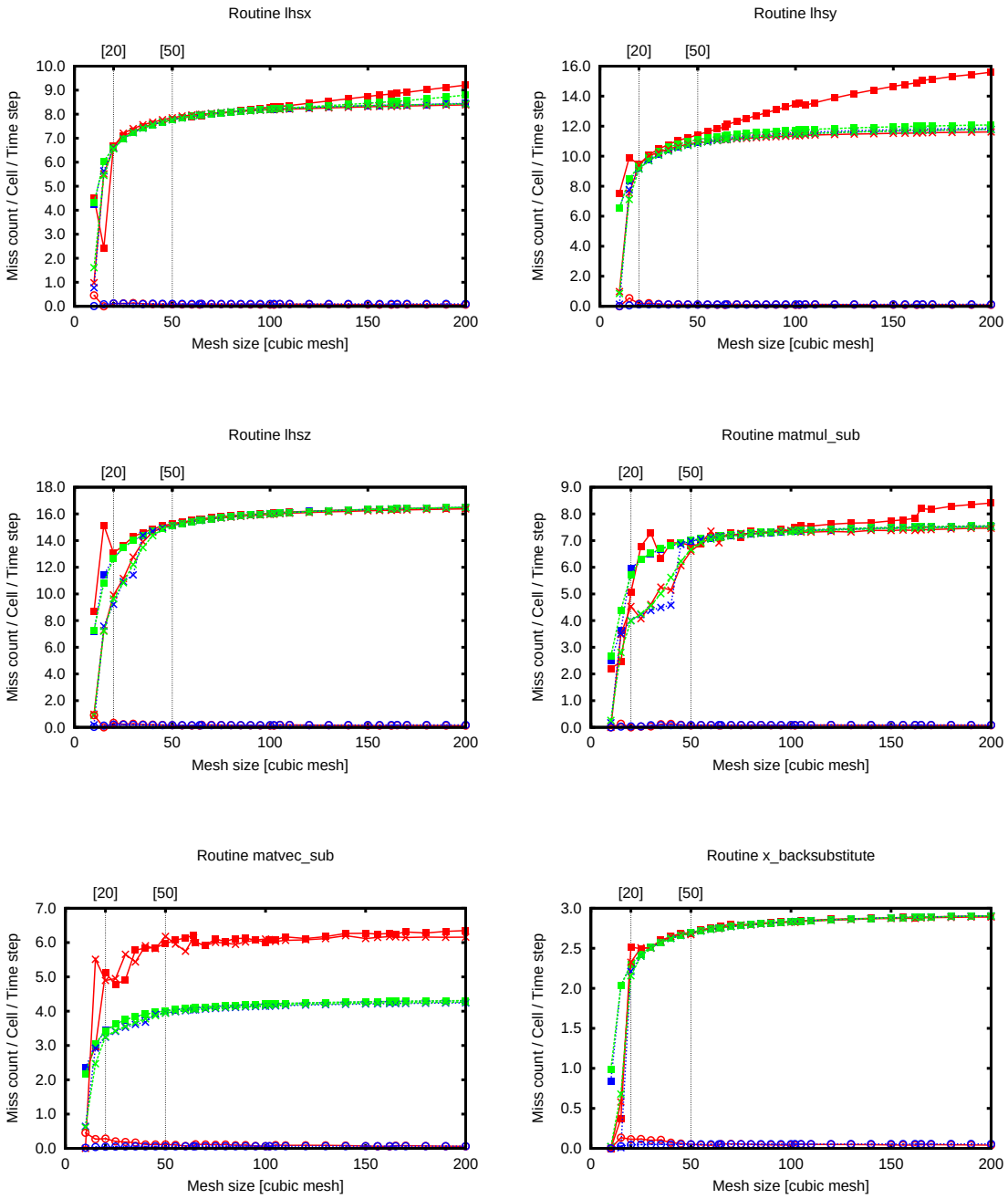


Figure B.2: Routine-level memory predictions for bt23 on IA64 (con't)

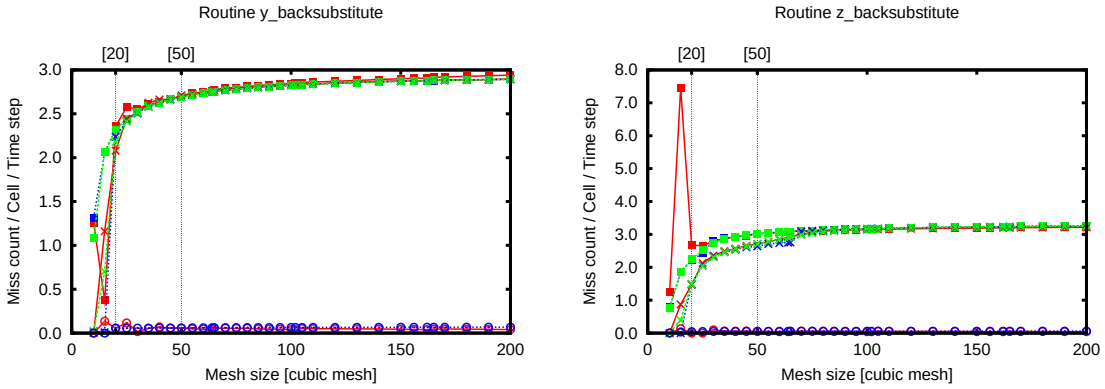


Figure B.2: Routine-level memory predictions for bt23 on IA64 (con't)

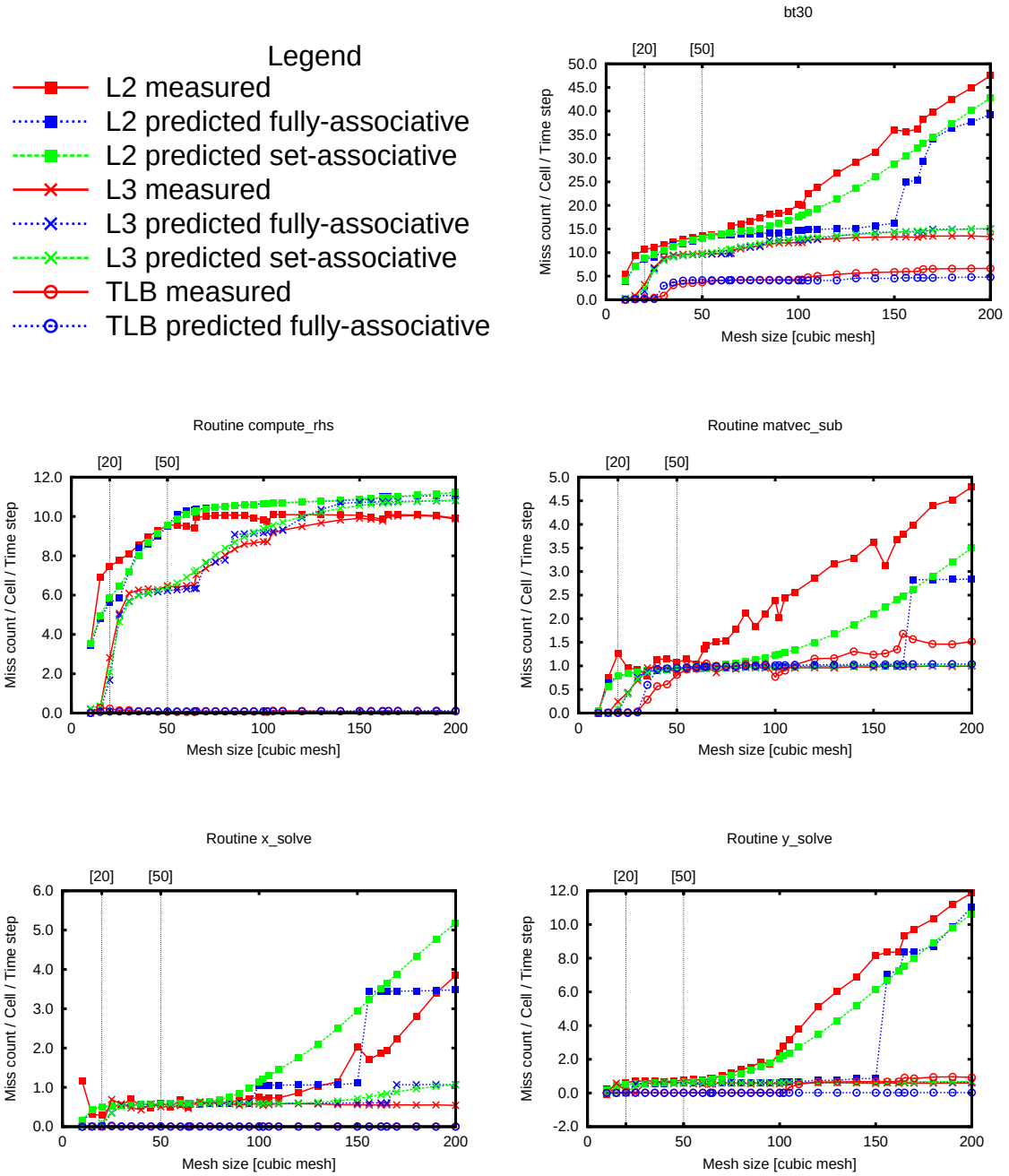


Figure B.3: Routine-level memory predictions for bt30 on IA64

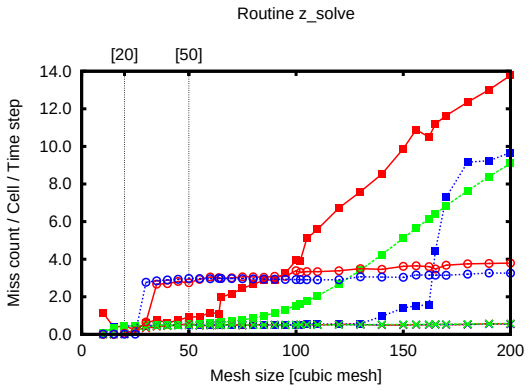


Figure B.3: Routine-level memory predictions for bt30 on IA64 (con't)

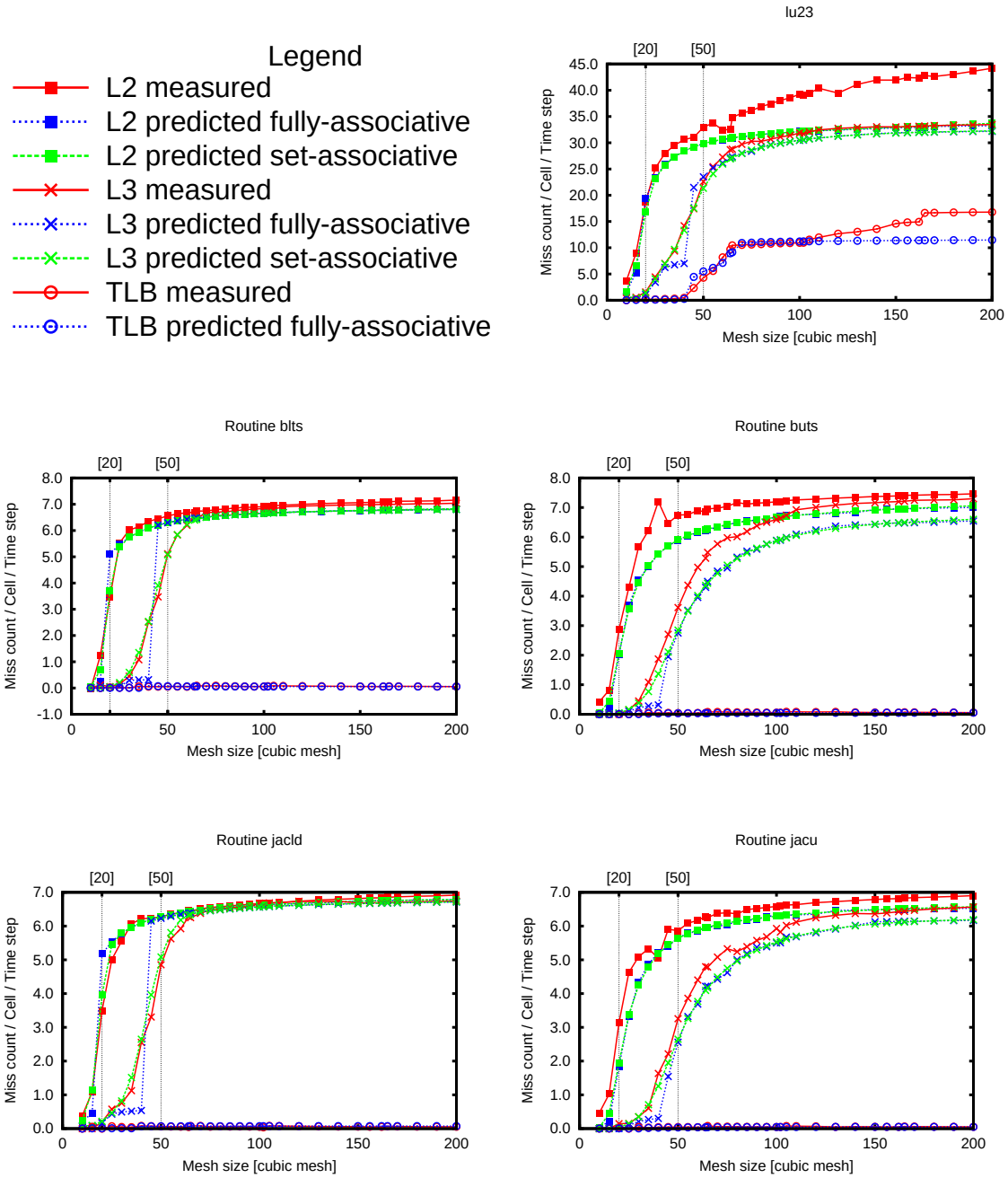


Figure B.4: Routine-level memory predictions for lu23 on IA64

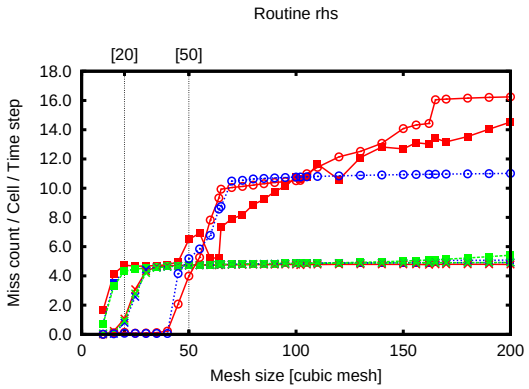


Figure B.4: Routine-level memory predictions for lu23 on IA64 (con't)

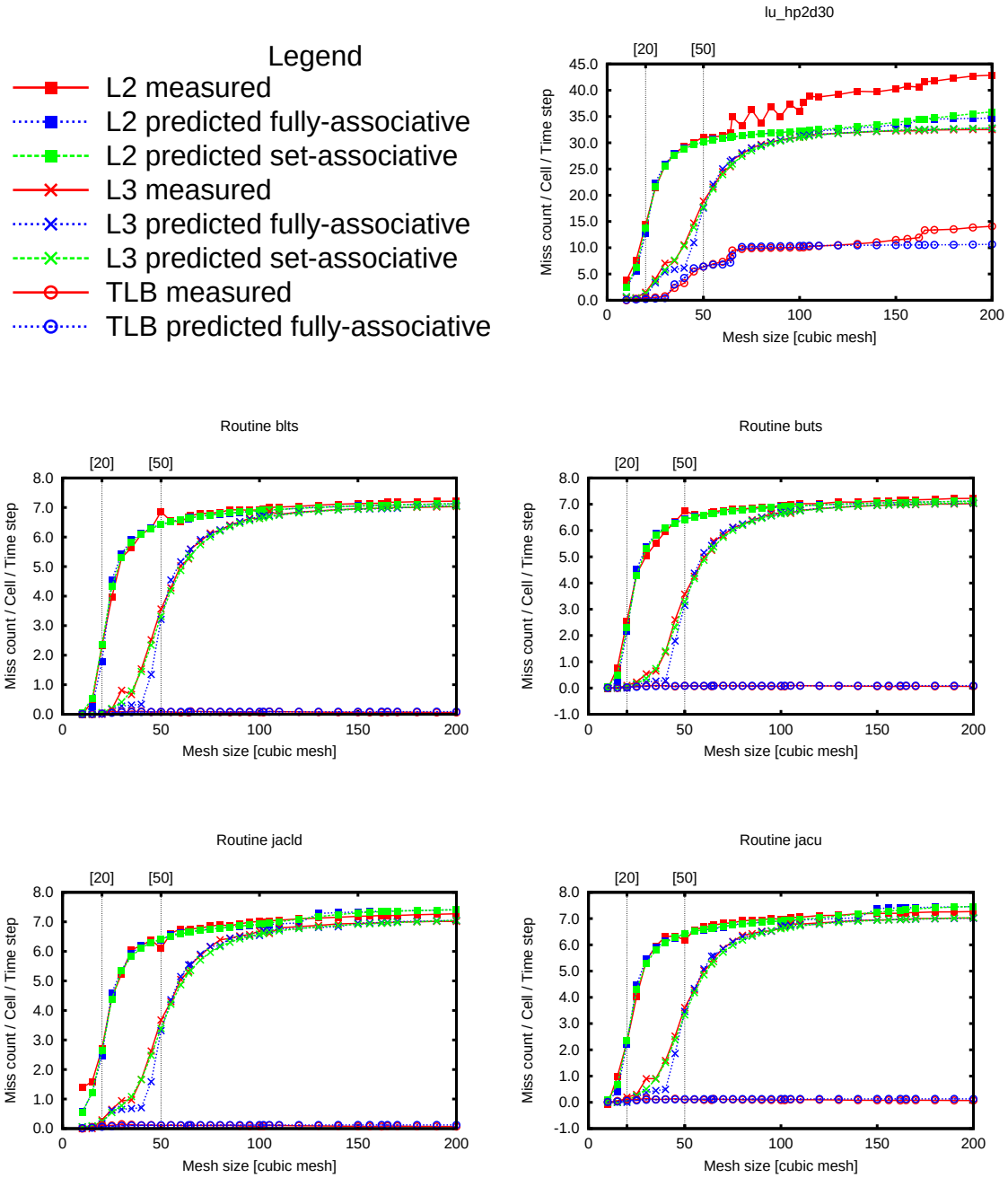


Figure B.5: Routine-level memory predictions for LU hp2d NPB 3.0 on IA64

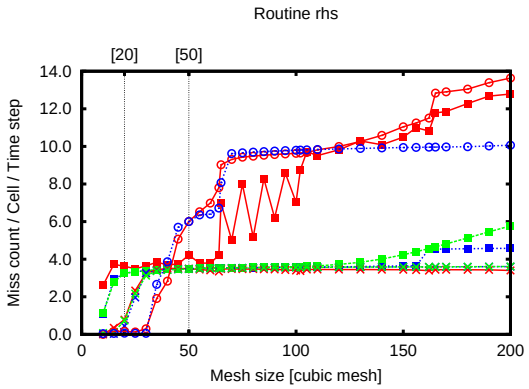


Figure B.5: Routine-level memory predictions for LU hp2d NPB 3.0 on IA64 (con't)

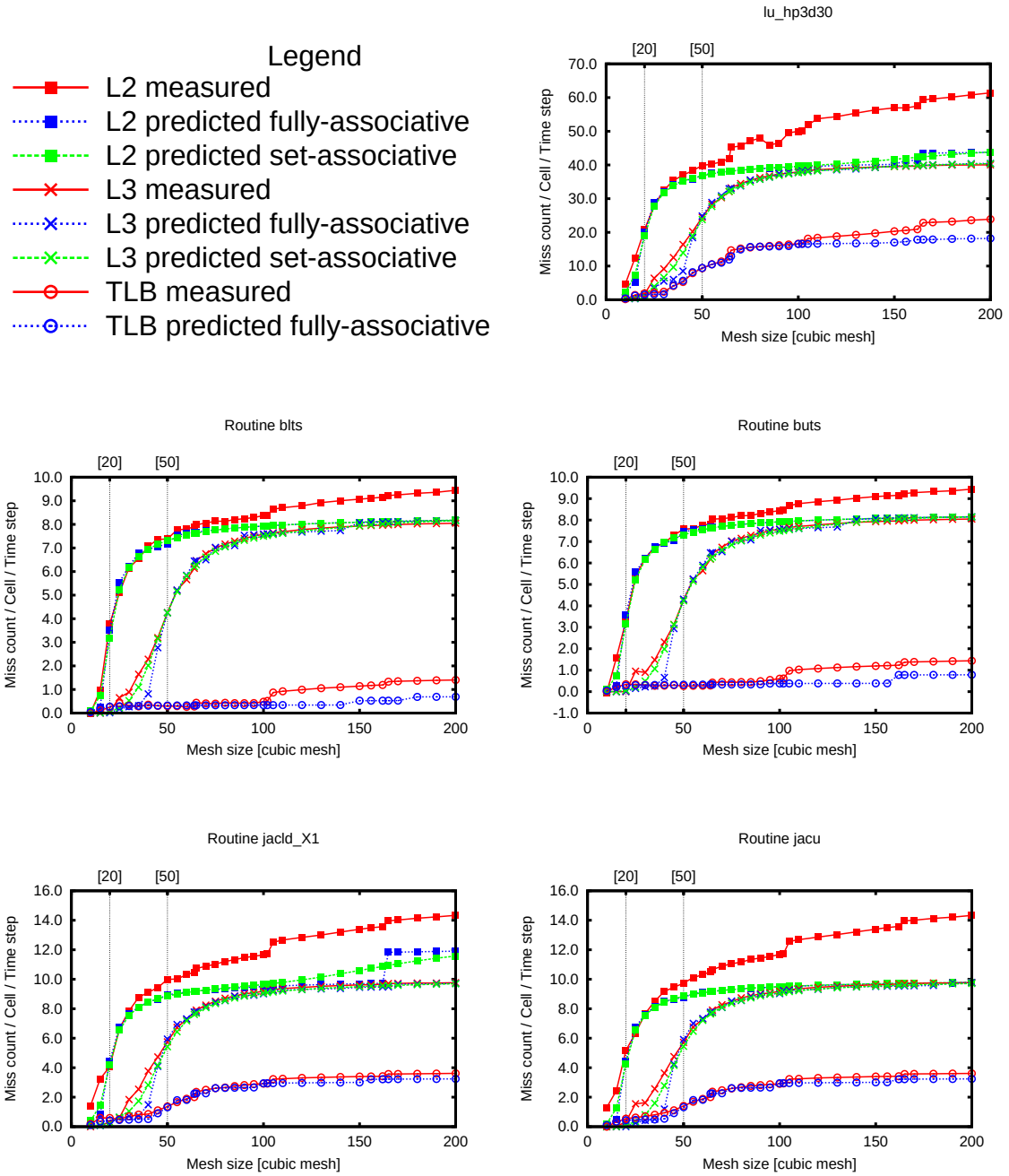


Figure B.6: Routine-level memory predictions for LU hp3d NPB 3.0 on IA64

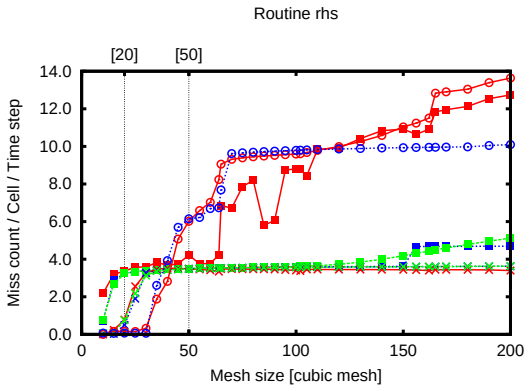


Figure B.6: Routine-level memory predictions for LU hp3d NPB 3.0 on IA64 (con't)

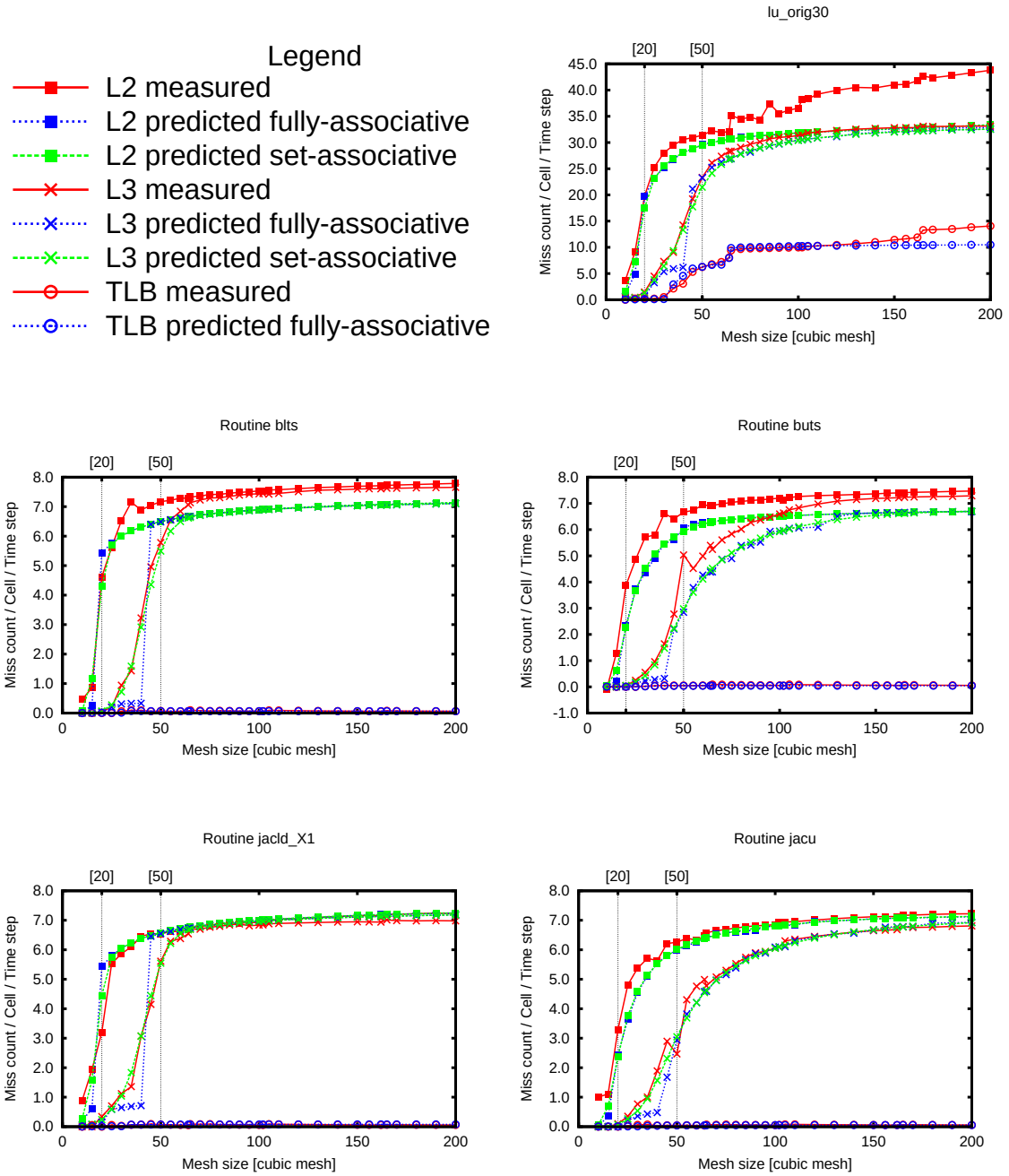


Figure B.7: Routine-level memory predictions for LU orig NPB 3.0 on IA64

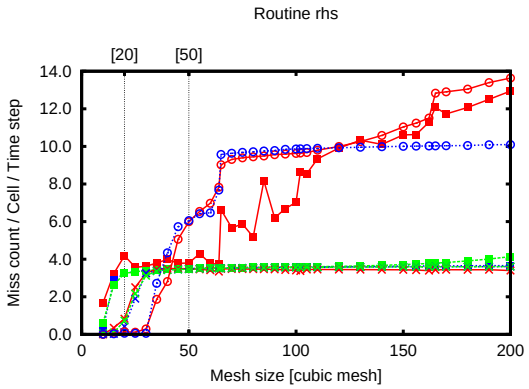


Figure B.7: Routine-level memory predictions for LU orig NPB 3.0 on IA64 (con't)

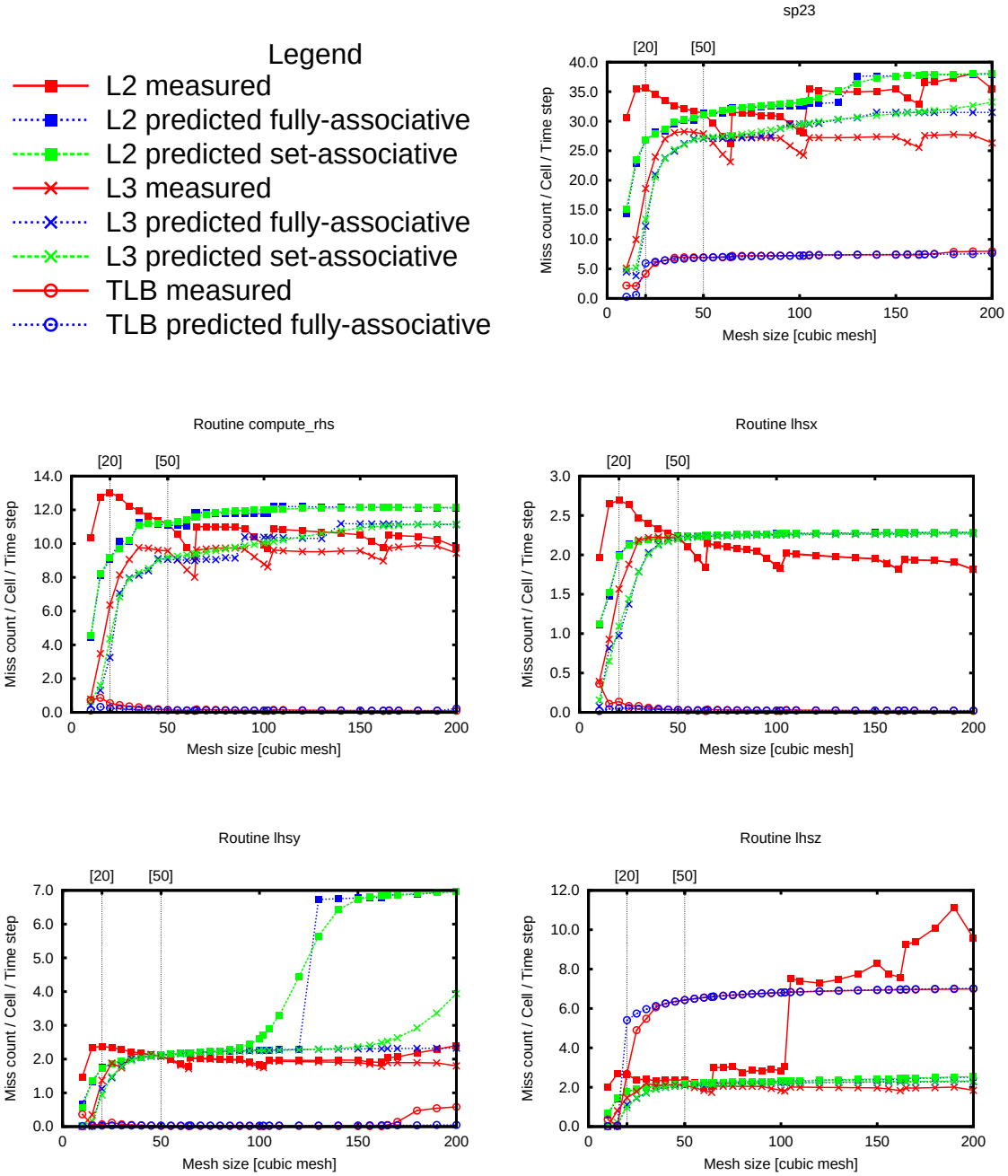


Figure B.8: Routine-level memory predictions for sp23 on IA64

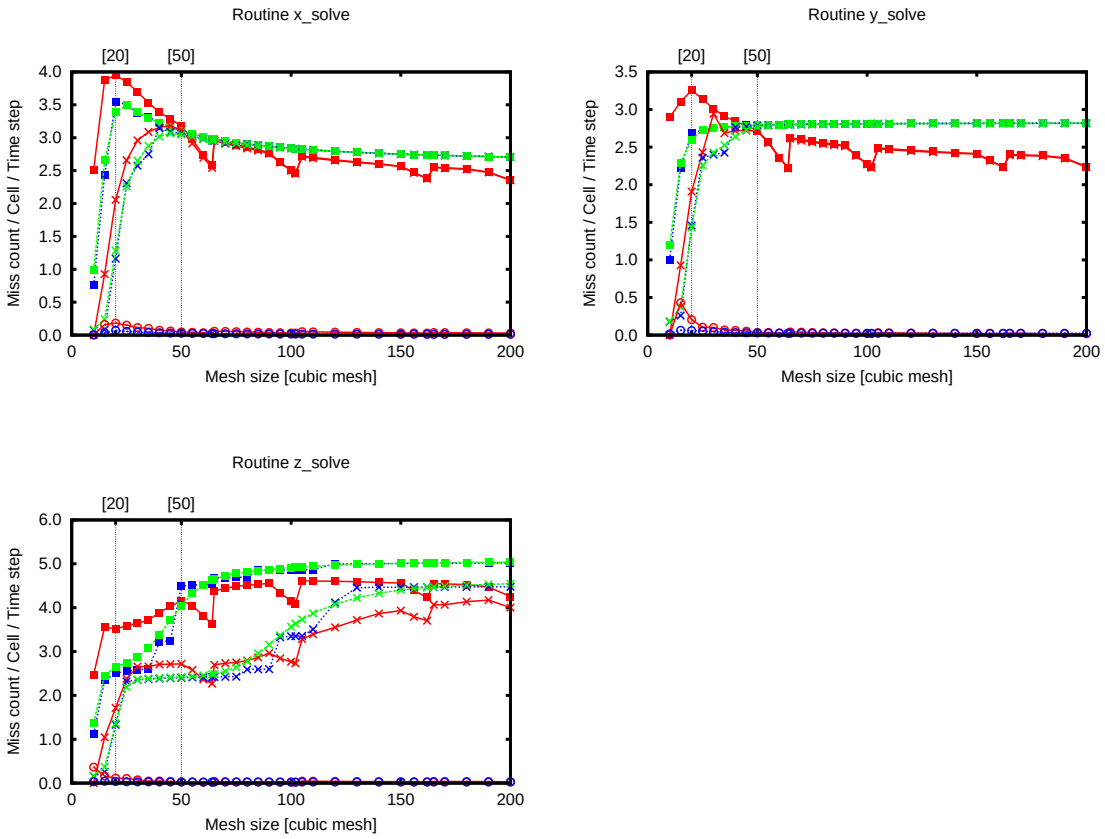


Figure B.8: Routine-level memory predictions for sp23 on IA64 (con't)

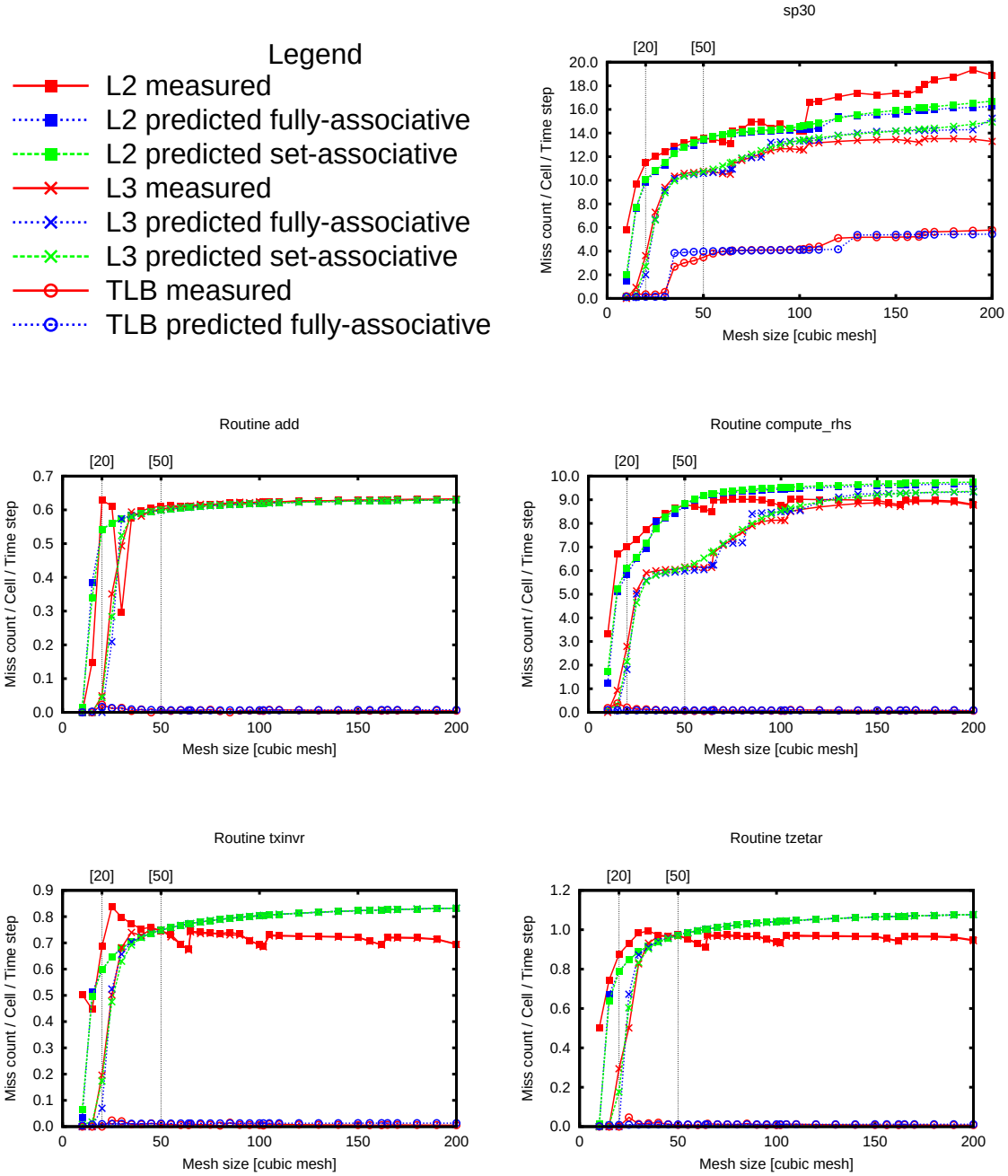


Figure B.9: Routine-level memory predictions for sp30 on IA64

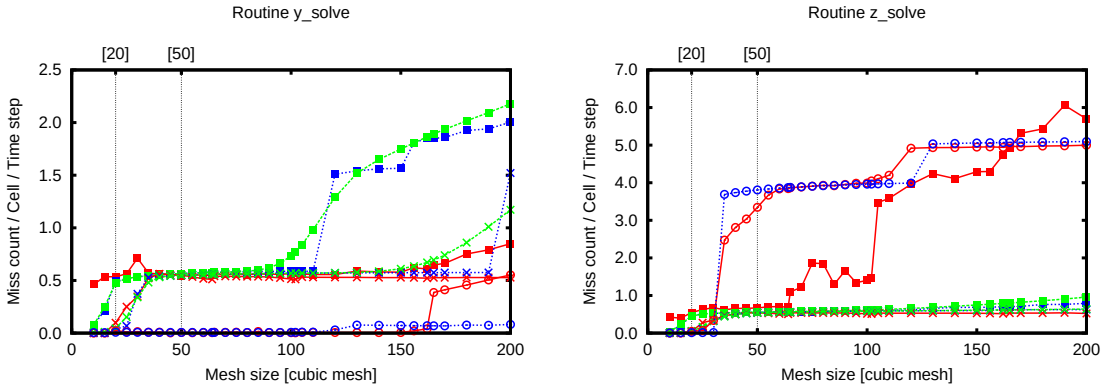


Figure B.9: Routine-level memory predictions for sp30 on IA64 (con't)

Appendix C

Routine Level Cache Miss Predictions on R12K

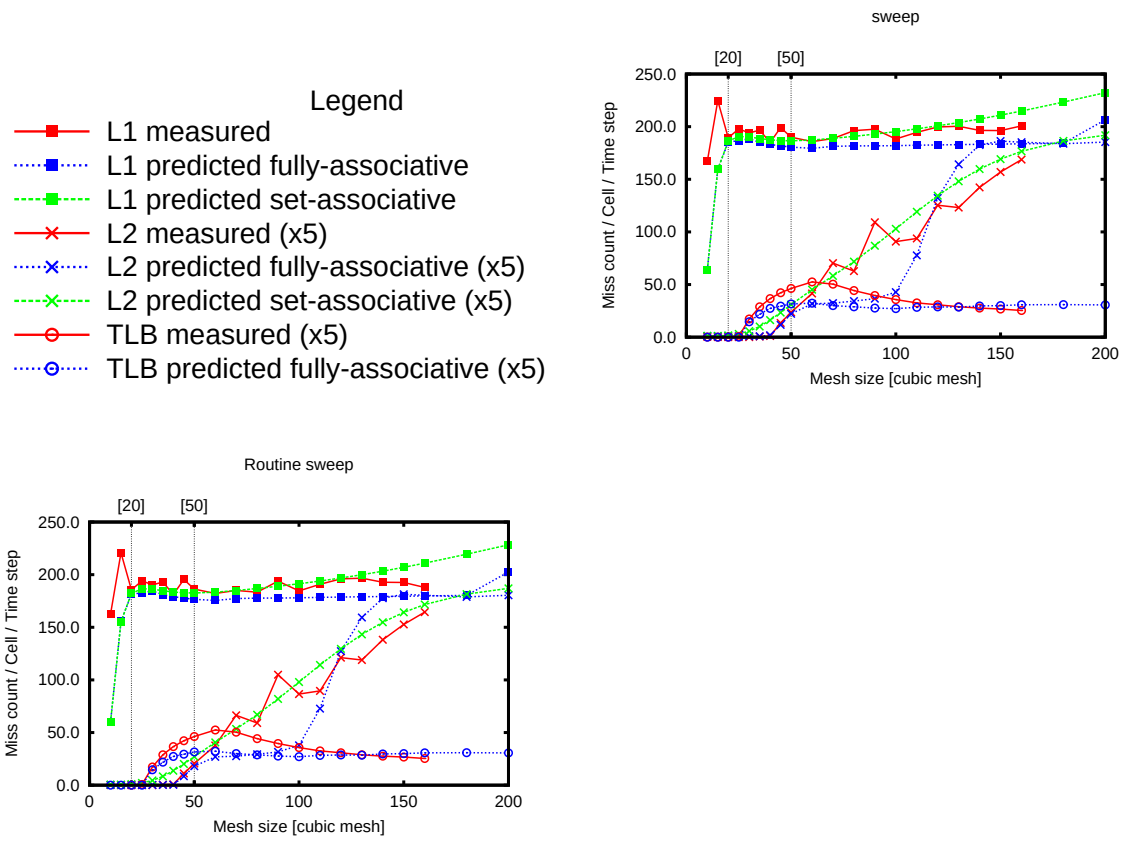


Figure C.1: Routine-level memory predictions for sweep on R12K

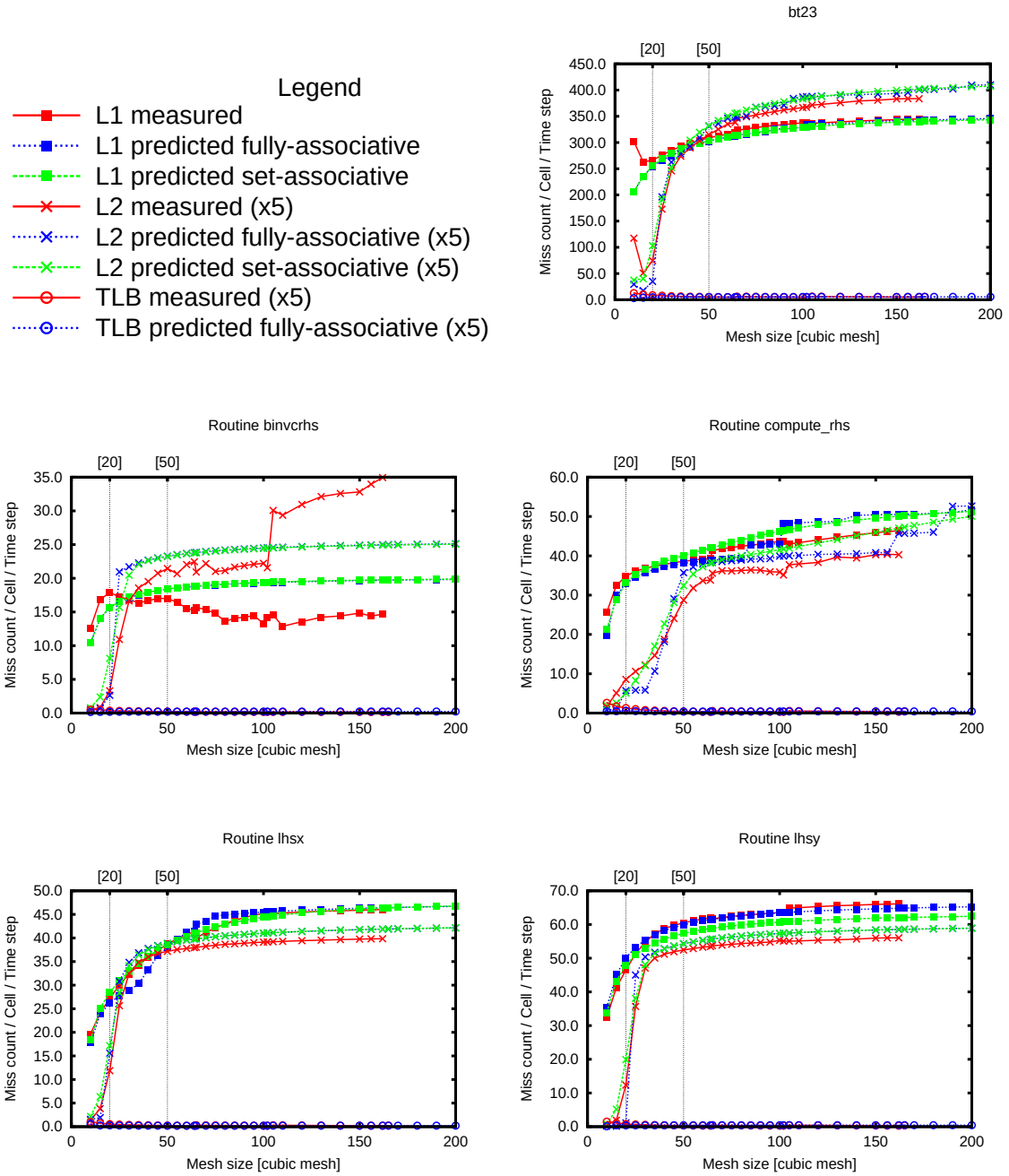


Figure C.2: Routine-level memory predictions for bt23 on R12K

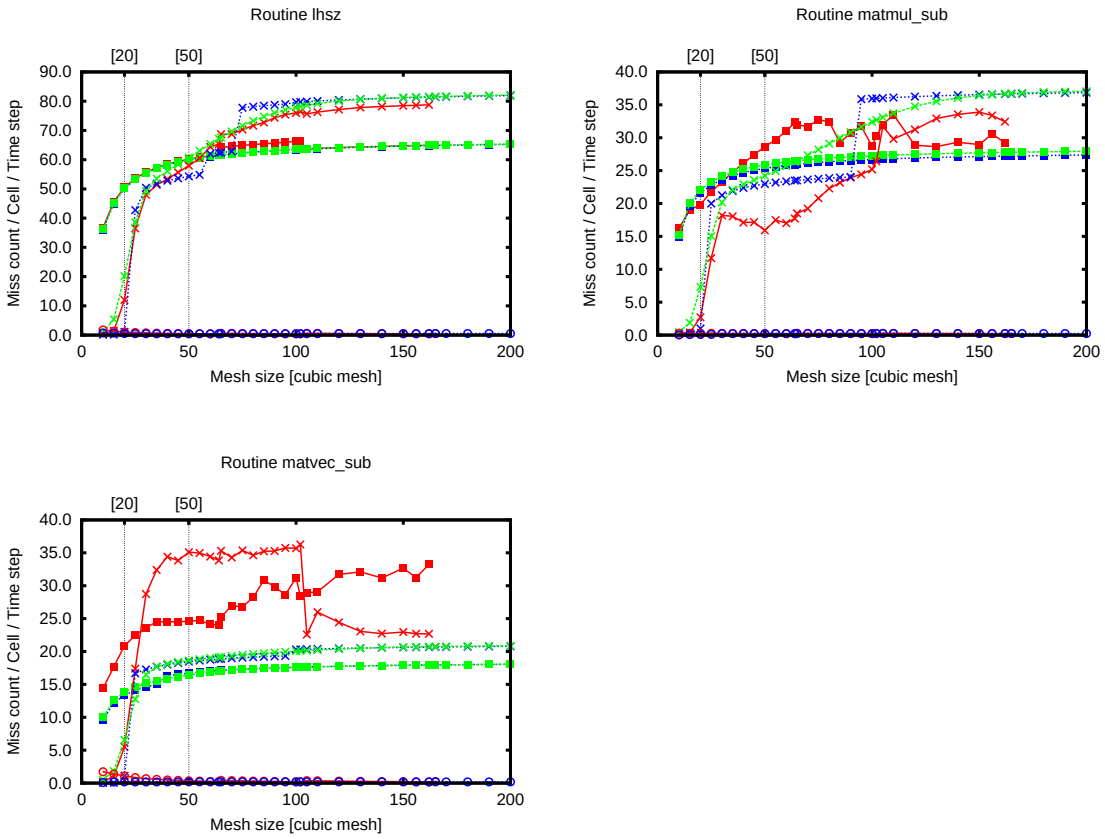


Figure C.2: Routine-level memory predictions for bt23 on R12K (con't)

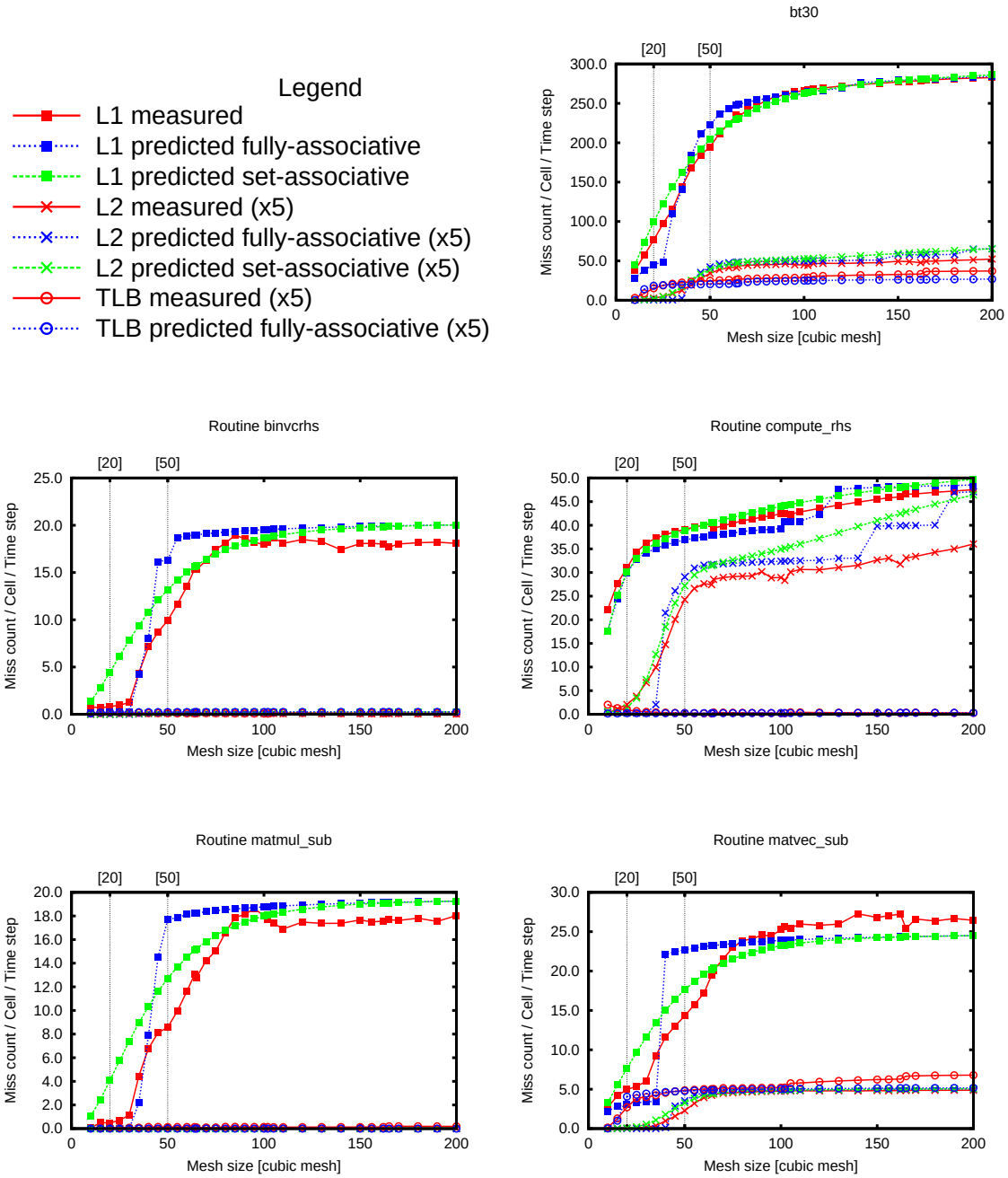


Figure C.3: Routine-level memory predictions for bt30 on R12K

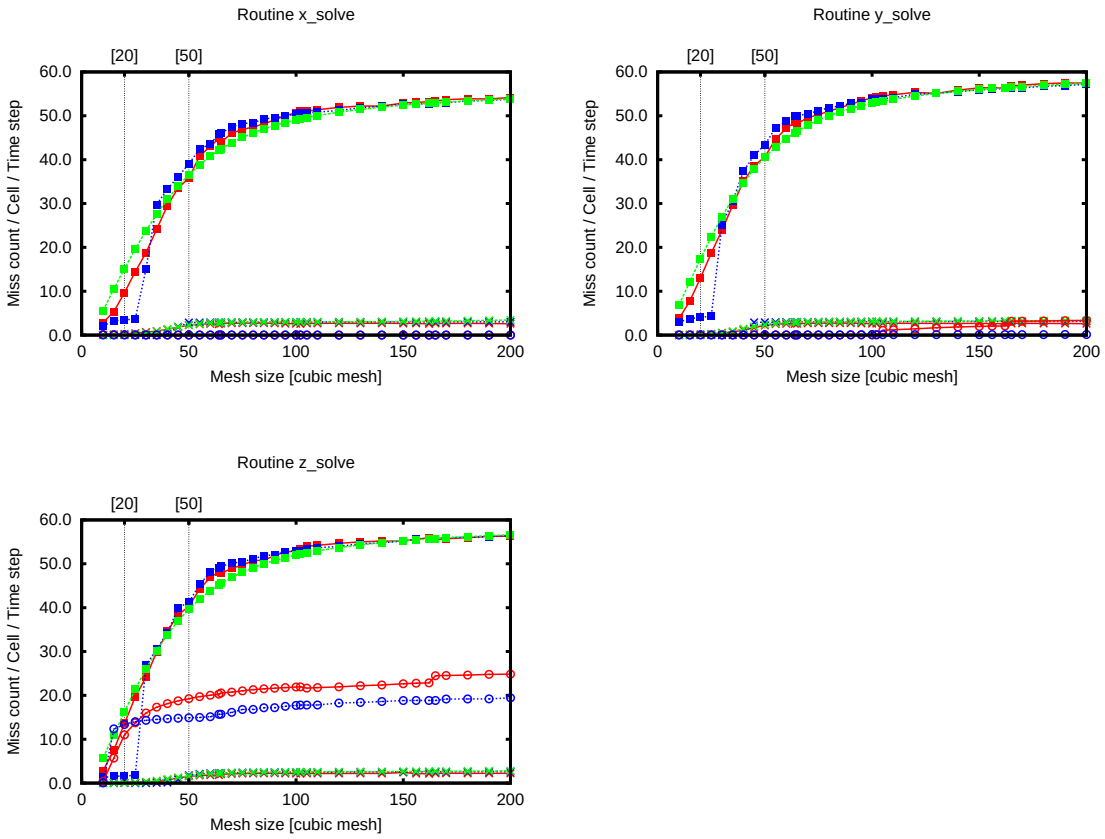


Figure C.3: Routine-level memory predictions for bt30 on R12K (con't)

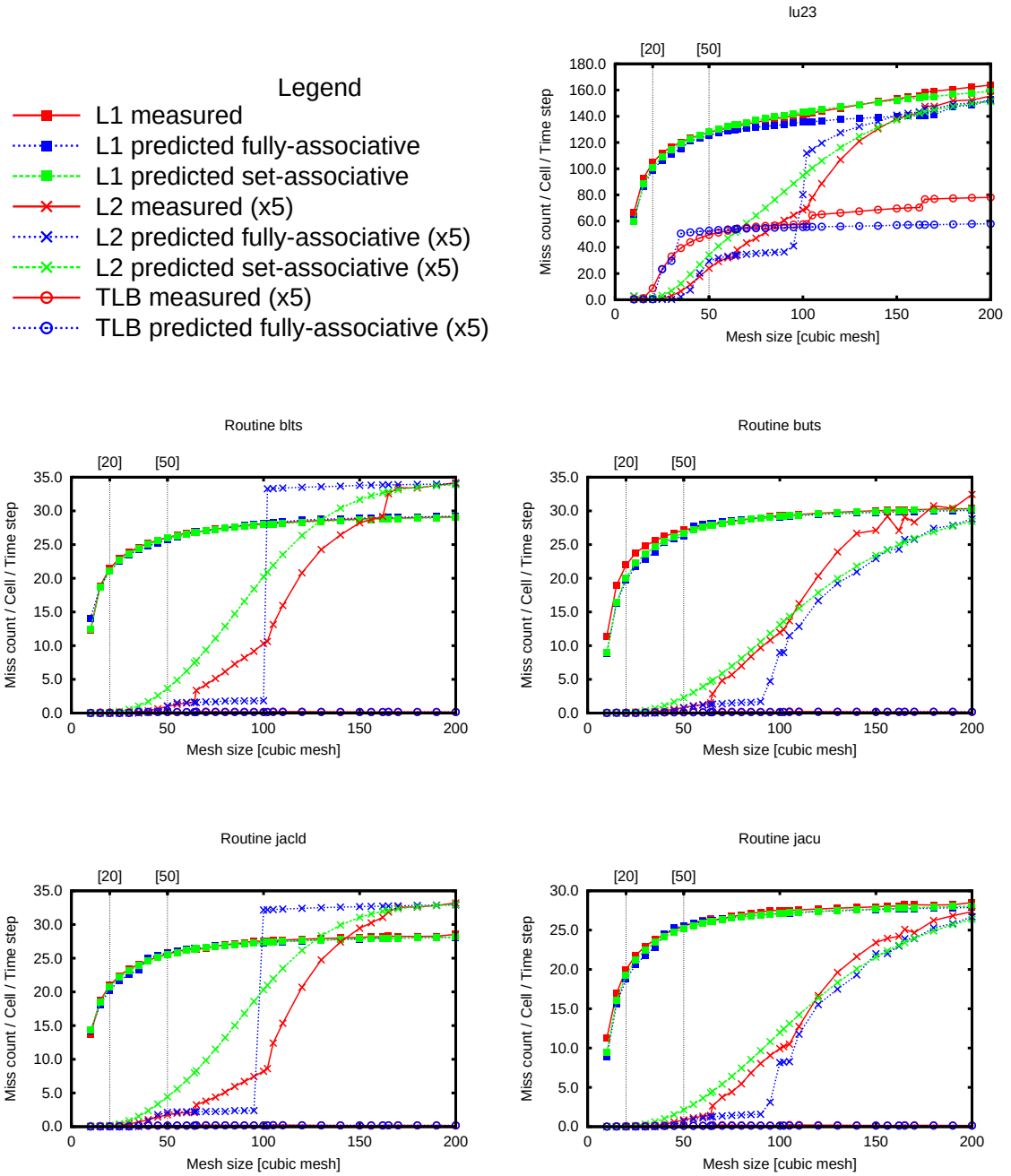


Figure C.4: Routine-level memory predictions for lu23 on R12K

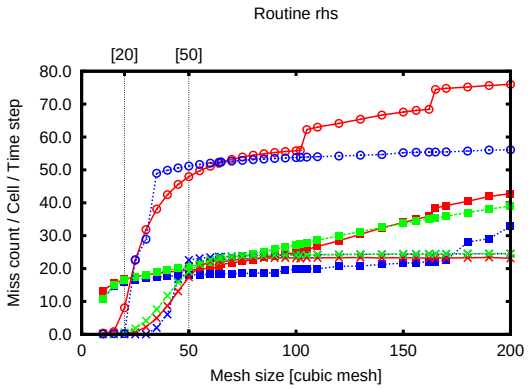


Figure C.4: Routine-level memory predictions for lu23 on R12K (con't)

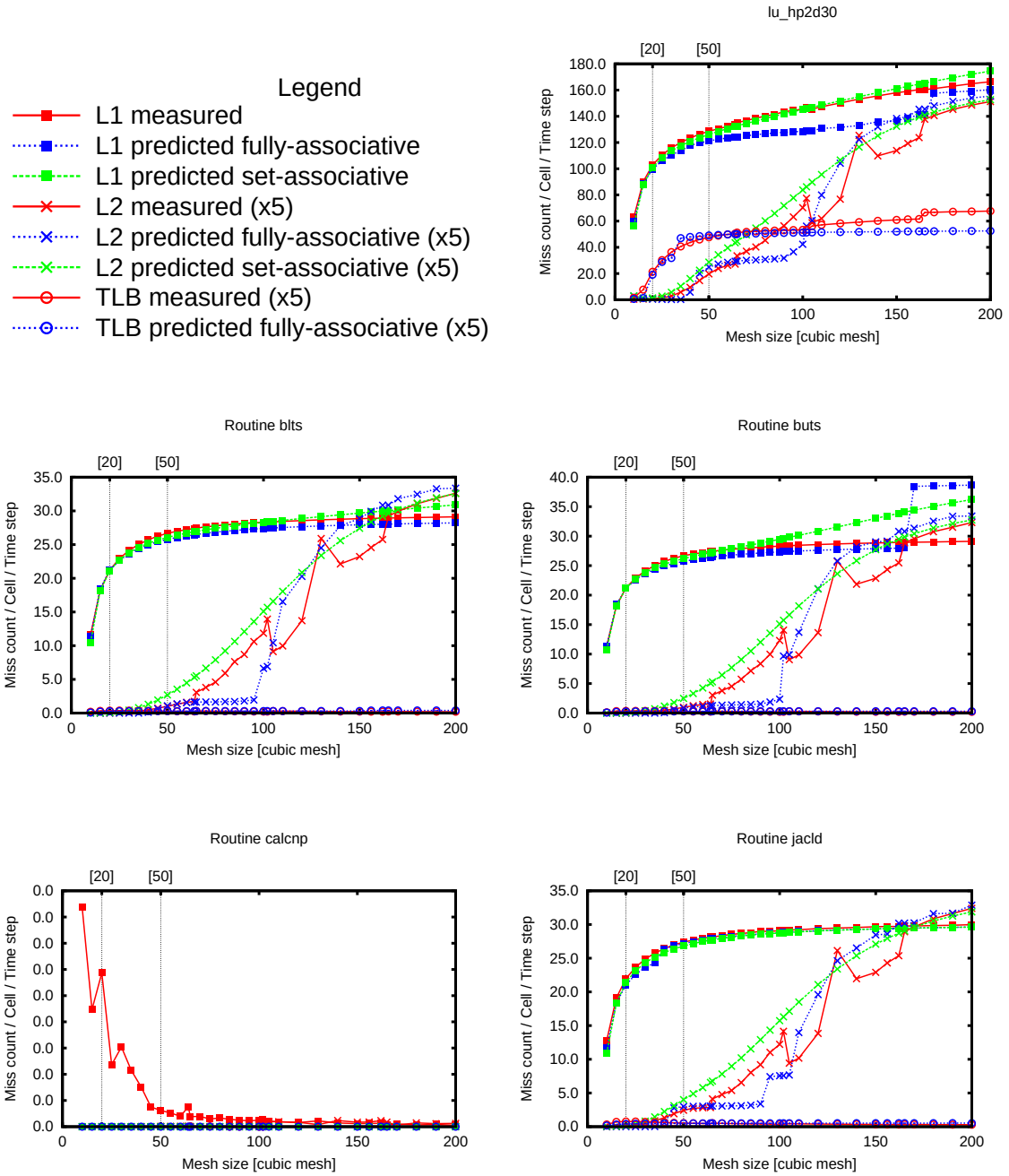


Figure C.5: Routine-level memory predictions for LU hp2d NPB 3.0 on R12K

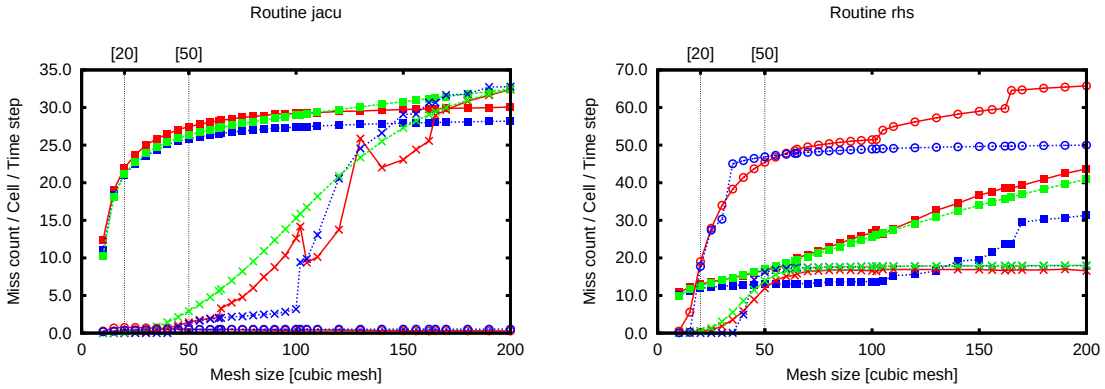


Figure C.5: Routine-level memory predictions for LU hp2d NPB 3.0 on R12K (con't)

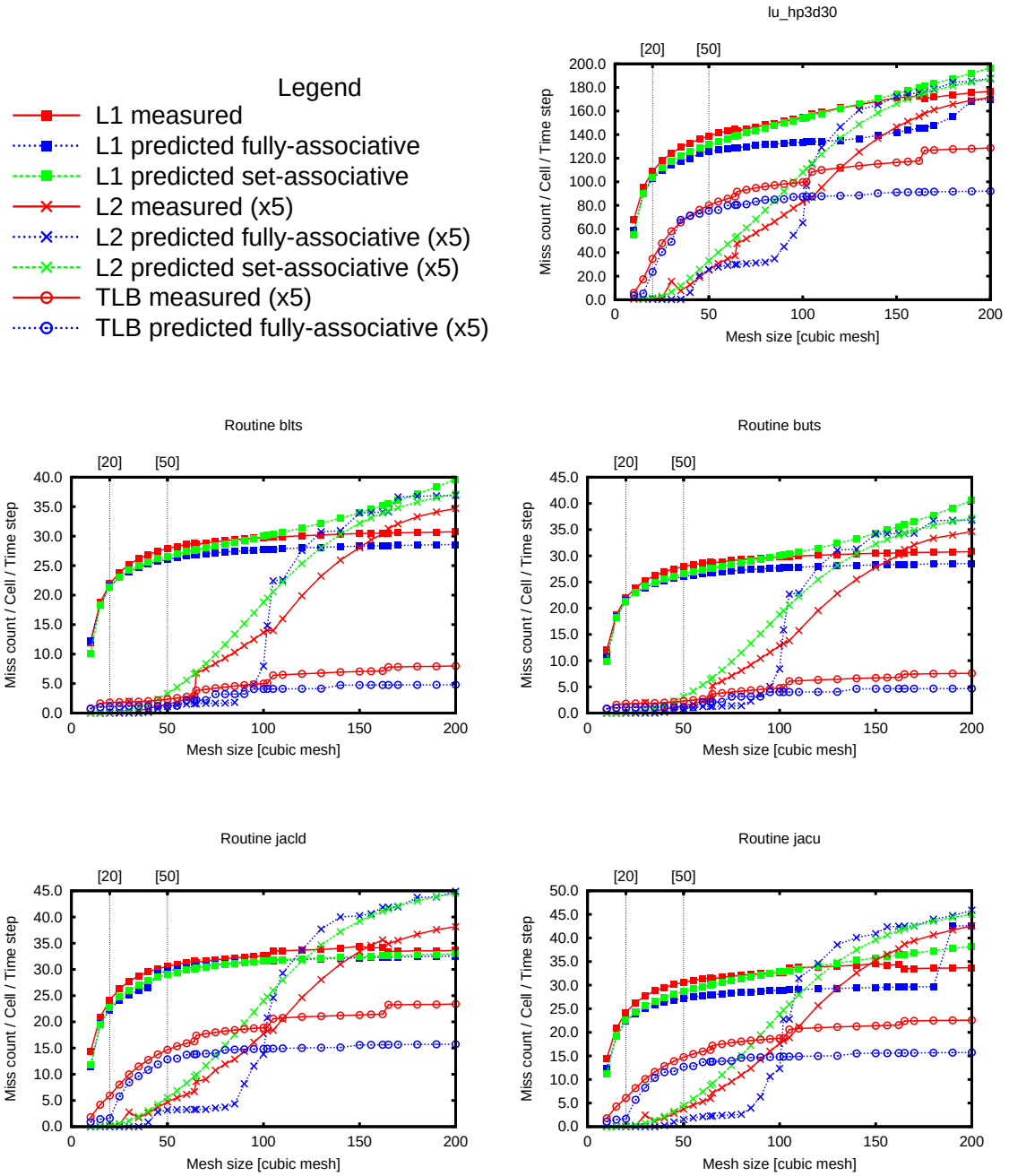


Figure C.6: Routine-level memory predictions for LU hp3d NPB 3.0 on R12K

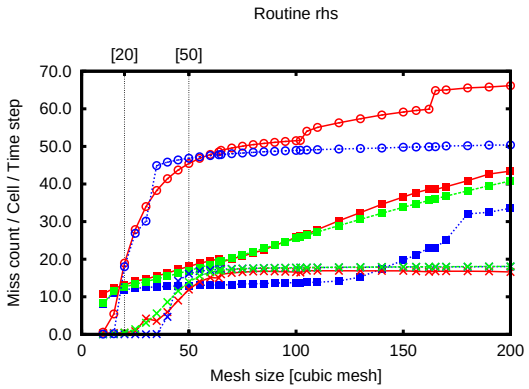


Figure C.6: Routine-level memory predictions for LU hp3d NPB 3.0 on R12K (con't)

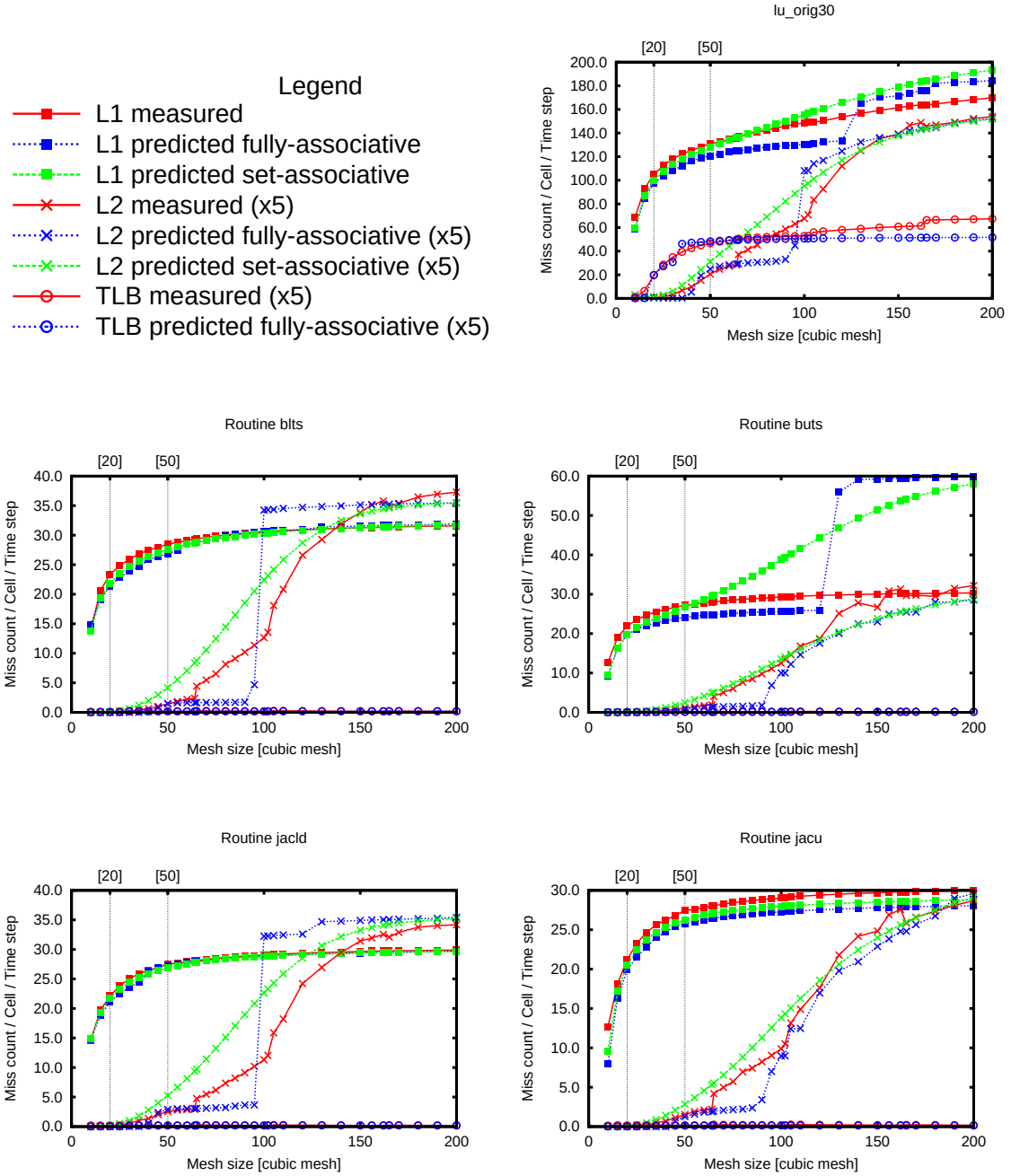


Figure C.7: Routine-level memory predictions for LU orig NPB 3.0 on R12K

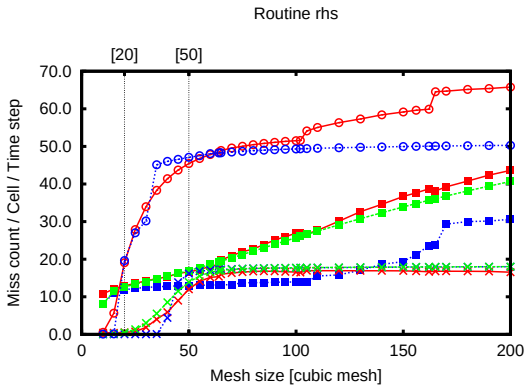


Figure C.7: Routine-level memory predictions for LU orig NPB 3.0 on R12K (con't)

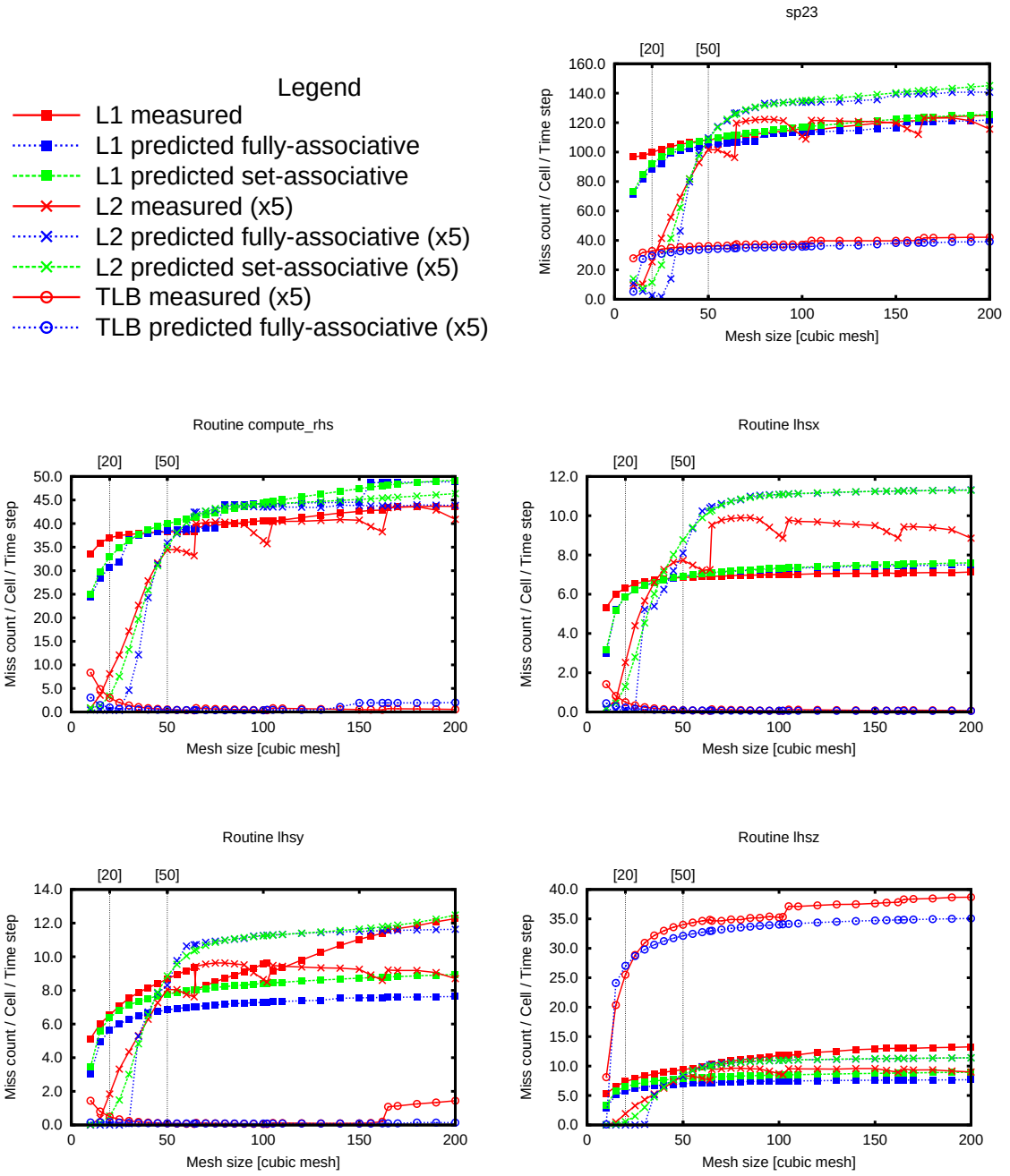


Figure C.8: Routine-level memory predictions for sp23 on R12K

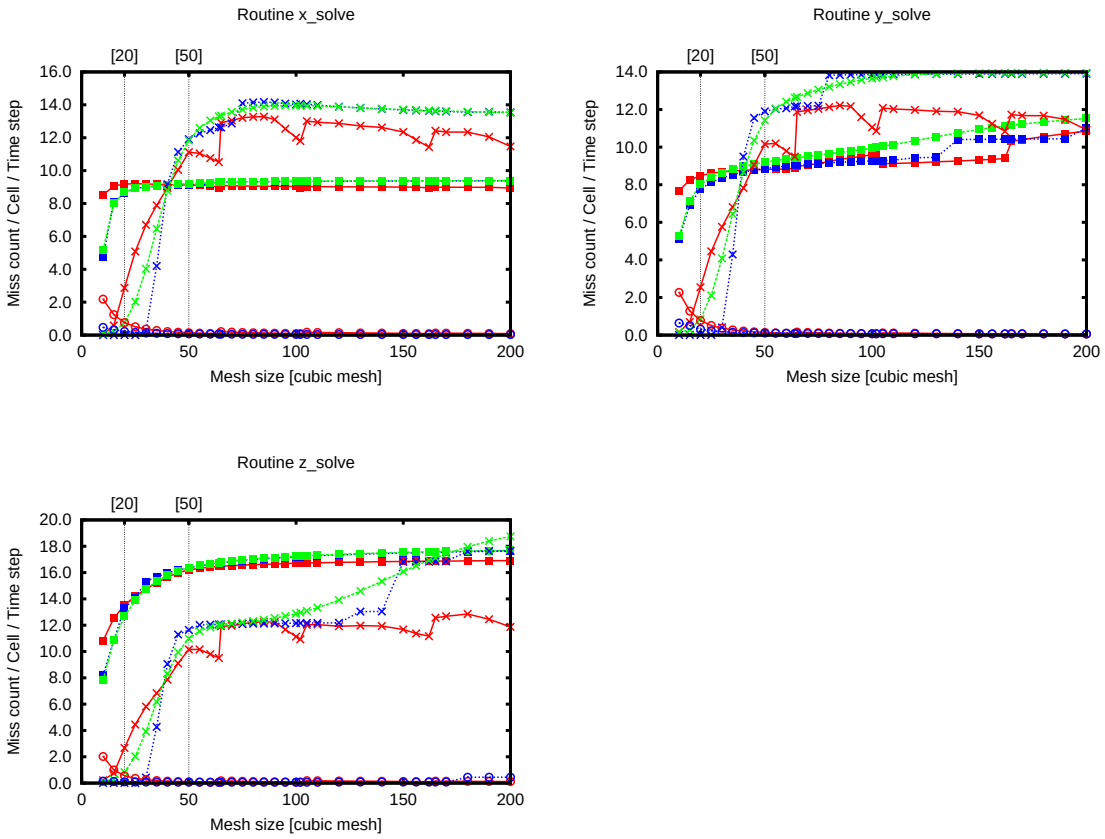


Figure C.8: Routine-level memory predictions for sp23 on R12K (con't)

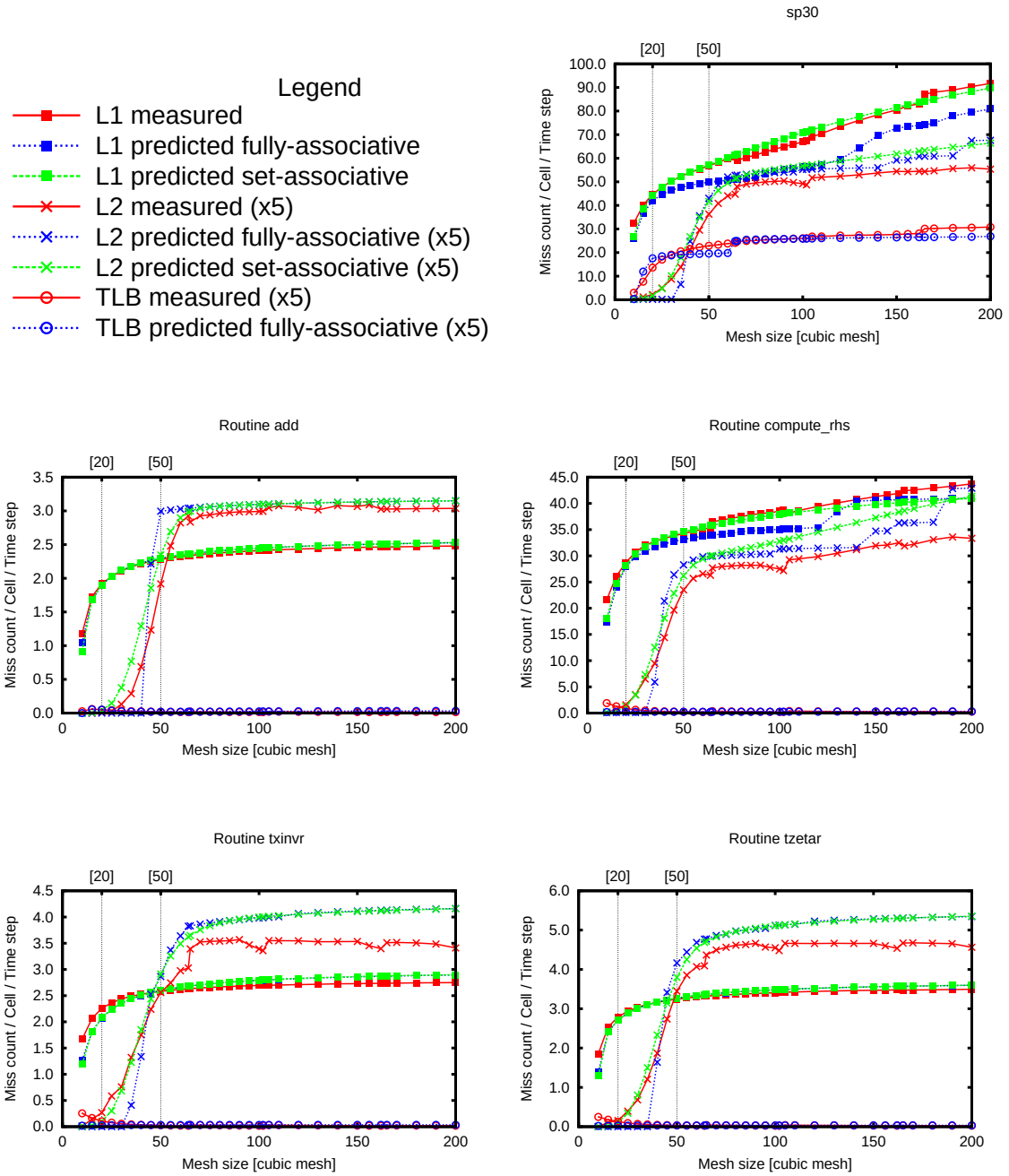


Figure C.9: Routine-level memory predictions for sp30 on R12K

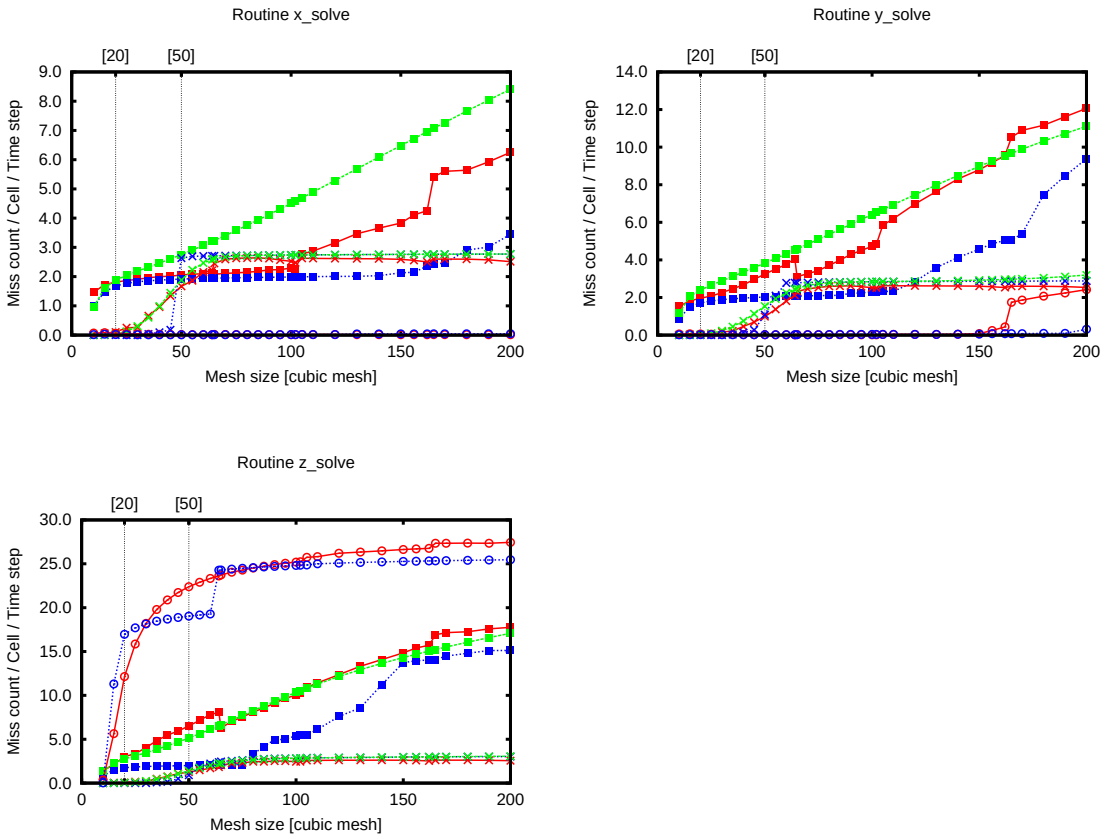


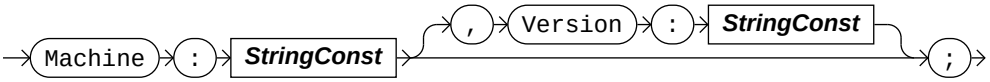
Figure C.9: Routine-level memory predictions for sp30 on R12K (con't)

Appendix D

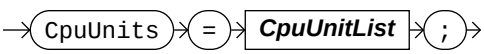
Grammar of the Machine Description Language

MachineDescription

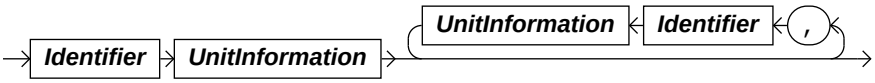
MachineName



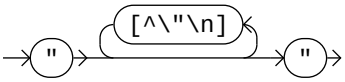
ExecUnits



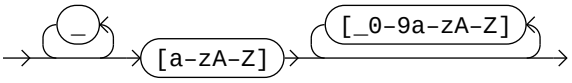
CpuUnitList



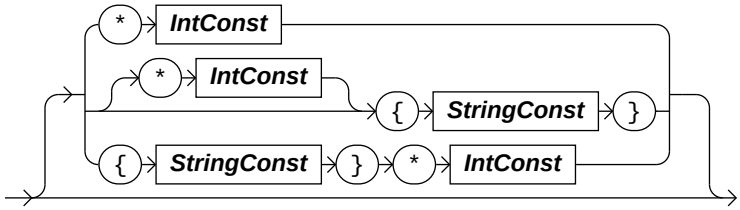
StringConst



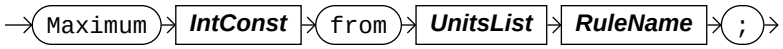
Identifier



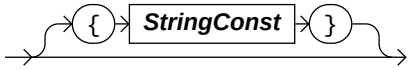
UnitInformation



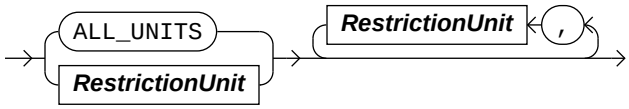
UnitsRestriction



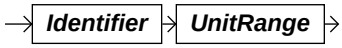
RuleName



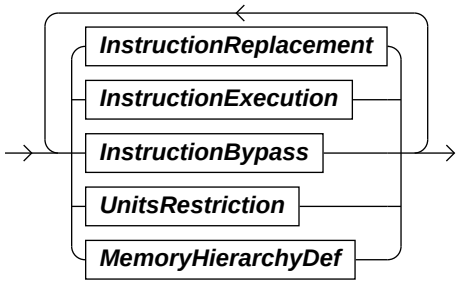
UnitsList



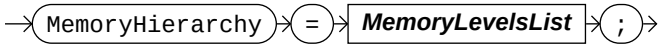
RestrictionUnit



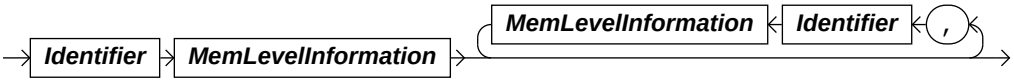
InstructionDescriptions



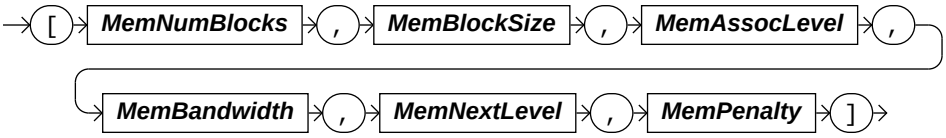
MemoryHierarchyDef



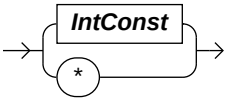
MemoryLevelsList



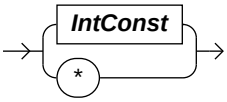
MemLevelInformation



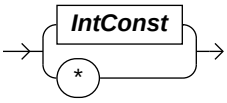
MemNumBlocks



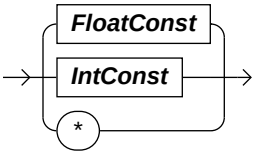
MemBlockSize



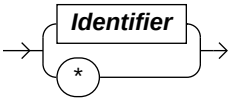
MemAssocLevel



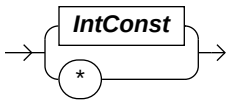
MemBandwidth



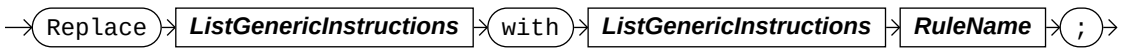
MemNextLevel



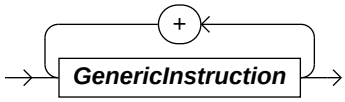
MemPenalty



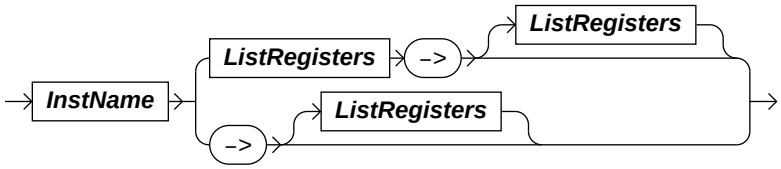
InstructionReplacement



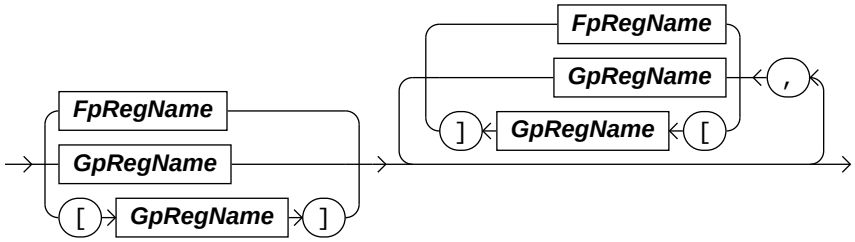
ListGenericInstructions



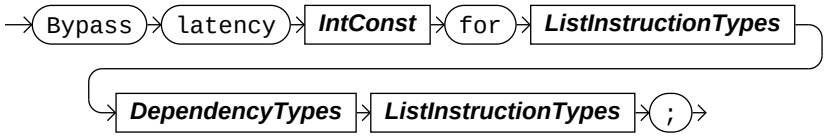
GenericInstruction



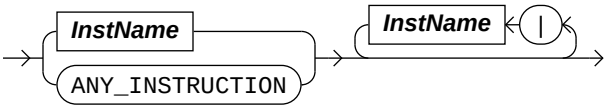
ListRegisters



InstructionBypass

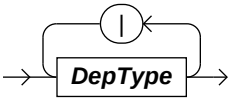


ListInstructionTypes

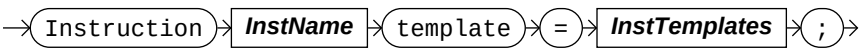


DependencyTypes

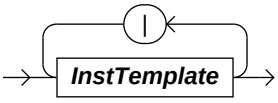
ListDependencies



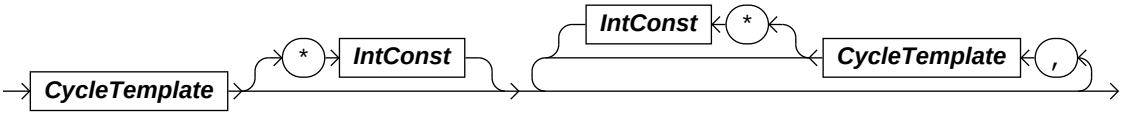
InstructionExecution



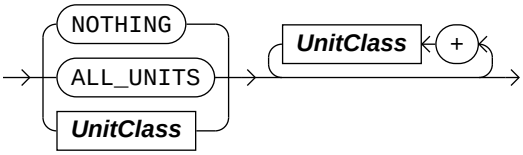
InstTemplates



InstTemplate

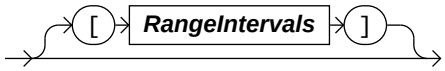


CycleTemplate

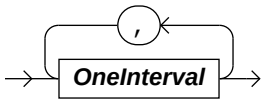


UnitClass

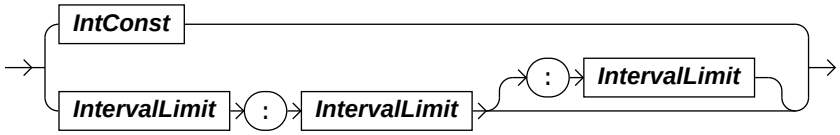
UnitRange



RangeIntervals



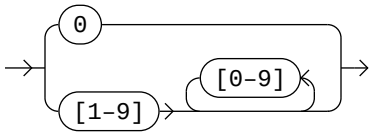
OneInterval



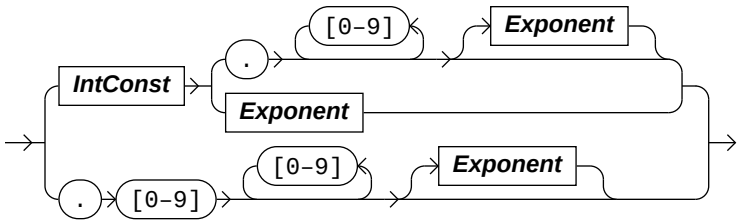
IntervalLimit

UnitCount

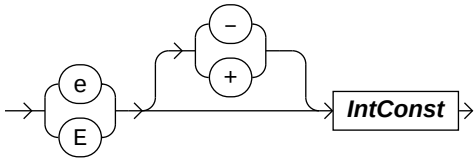
IntConst



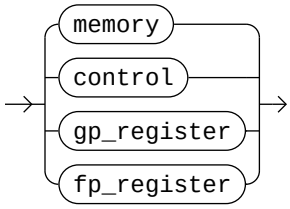
FloatConst



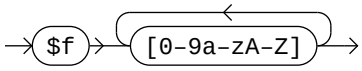
Exponent



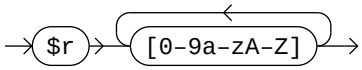
DepType



FpRegName



GpRegName



InstName