

**THE ROLE OF INSTRUMENTATION AND MAPPING
IN PERFORMANCE MEASUREMENT**

by

SAMEER SURESH SHENDE

A DISSERTATION

**Presented to the Department of Computer and Information Science
and the Graduate School of the University of Oregon
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy**

August 2001

“The Role of Instrumentation and Mapping in Performance Measurement,” a dissertation prepared by Sameer Suresh Shende in partial fulfillment of the requirements for the Doctor of Philosophy degree in the Department of Computer and Information Science.

This dissertation has been approved and accepted by:



Dr. Janice E. Cuny, Co-chair of the Examining Committee



Dr. Allen D. Malony, Co-chair of the Examining Committee

02/22/01
Date

Committee in charge: Dr. Janice E. Cuny, Co-chair
 Dr. Allen D. Malony, Co-chair
 Dr. Zena Ariola
 Dr. Roger Haydock
 Dr. Peter Beckman

Accepted by:



Vice Provost and Dean of the Graduate School

© 2001 Sameer Suresh Shende

An Abstract of the Dissertation of

Sameer Suresh Shende for the degree of Doctor of Philosophy

in the Department of Computer and Information Science

to be taken

August 2001

Title: THE ROLE OF INSTRUMENTATION AND MAPPING

IN PERFORMANCE MEASUREMENT

Approved:


Dr. Janice E. Cuny, Co-chair


Dr. Allen D. Malony, Co-chair

Technology for empirical performance evaluation of parallel programs is driven by the increasing complexity of high performance computing environments and programming methodologies. This complexity – arising from the use of high-level parallel languages, domain-specific numerical frameworks, heterogeneous execution models and platforms, multi-level software optimization strategies, and multiple compilation models – widens the semantic gap between a programmer’s understanding of his/her code and it’s runtime behavior. To keep pace, performance tools must provide for the effective instrumentation of complex software and the correlation of runtime performance data with user-level semantics.

To address these issues, this dissertation contributes:

- a strategy for utilizing multi-level instrumentation to improve the coverage of performance measurement in complex, layered software;
- techniques for mapping low-level performance data to higher levels of abstraction in order to reduce the semantic gap between user's abstractions and runtime time behavior; and
- the concept of instrumentation-aware compilation that extends traditional compilers to preserve the semantics of fine-grained performance instrumentation despite aggressive program restructuring.

In each case, the dissertation provides prototype implementations and case studies of the needed tools and frameworks.

This dissertation research aims to influence the way performance observation tools and compilers for high performance computers are designed and implemented.

CURRICULUM VITA

NAME OF AUTHOR: Sameer Suresh Shende

PLACE OF BIRTH: Bombay, India

DATE OF BIRTH: July 19, 1970

GRADUATE AND UNDERGRADUATE SCHOOLS ATTENDED:

University of Oregon
Indian Institute of Technology, Bombay

DEGREES AWARDED:

Doctor of Philosophy in Computer and Information Science, 2001, University of Oregon
Master of Science in Computer and Information Science, 1996, University of Oregon
Bachelor of Technology in Electrical Engineering, 1991, Indian Institute of Technology, Bombay

AREAS OF SPECIAL INTEREST:

Instrumentation for Performance Evaluation Tools
Parallel and Distributed Processing

PROFESSIONAL EXPERIENCE:

Graduate Research Fellow, Department of Computer and Information Science,
University of Oregon, Eugene, OR, 1994-2001

Graduate Research Assistant, Advanced Computing Laboratory, Los Alamos
National Laboratory, NM, Summer 1997, 1998

Systems Analyst, Applied Technology Group, TATA Unisys Ltd., Bombay, India,
1991-1994

Project Engineer, VLSI Design Center, Department of Computer Science, Indian
Institute of Technology, Bombay, India, 1991

PUBLICATIONS:

- S. Shende, A. D. Malony, "Integration and Application of the TAU Performance System in Parallel Java Environments," Proceedings of the Joint ACM Java Grande - ISCOPE 2001 Conference, June 2001.
- S. Shende, A. D. Malony, R. Ansell-Bell, "Instrumentation and Measurement Strategies for Flexible and Portable Empirical Performance Evaluation," Proceedings of International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001), CSREA, June 2001.
- H. Truong, T. Fahringer, G. Madsen, A. Malony, H. Moritsch, S. Shende, "On Using SCALEA for Performance Analysis of Distributed and Parallel Programs," (to appear) Proceedings of SC 2001 Conference, November 2001.
- A. Malony and S. Shende, "Performance Technology for Complex Parallel and Distributed Systems," Proceedings of DAPSYS 2000, in P. Kacsuk and G. Kotsis (editors), *Distributed and Parallel Systems: From Instruction Parallelism to Cluster Computing*, Kluwer, Norwell, MA, pp. 37-46, 2000.
- S. Shende, and A. D. Malony, "Performance Tools for Parallel Java Environments," Proceedings of the Second Workshop on Java for High Performance Computing, International Conference on Supercomputing, 2000.
- K. A. Lindlan, J. Cuny, A. D. Malony, S. Shende, B. Mohr, R. Rivenburgh, C. Rasmussen. "A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates," Proceedings of SC2000: High Performance Networking and Computing Conference, November 2000.
- S. Vajracharya, S. Karmesin, P. Beckman, J. Crotinger, A. Malony, S. Shende, R. Oldehoeft, S. Smith, "SMARTS: Exploiting Temporal Locality and Parallelism through Vertical Execution," Los Alamos National Laboratory Technical Report LA-UR-99-16, Los Alamos, NM, 1999 (also appears in Proceedings of 1999 International Conference on Supercomputing, ACM, pp. 302-310, 1999).

- S. Shende, A. D. Malony, J. Cuny, K. Lindlan, P. Beckman and S. Karmesin, "Portable Profiling and Tracing for Parallel Scientific Applications using C++," Proceedings of SPDT'98: ACM SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 134-145, Aug. 1998.
- S. Shende, "Profiling and Tracing in Linux," Proceedings of Extreme Linux Workshop #2, USENIX Annual Technical Conference, 1999.
- T. Sheehan, A. Malony, S. Shende, "A Runtime Monitoring Framework for the TAU Profiling System," in S. Matsuoka, R. Oldehoeft, and M. Tholburn (editors), *Computing in Object-Oriented Parallel Environments*, Third International Symposium ISCOPE '99, LNCS, No. 1732, Springer-Verlag, Berlin, pp. 170-181, Dec. 1999.
- S. Shende, A. Malony, and S. Hackstadt, "Dynamic Performance Callstack Sampling: Merging TAU and DAQV," in B. Kågström, J. Dongarra, E. Elmroth and J. Wasniewski (editors) *Applied Parallel Computing. Large Scale Scientific and Industrial Problems*, 4th International Workshop, PARA '98, LNCS, No. 1541, Springer-Verlag, Berlin, pp. 515-520, 1998.
- K. Lindlan, A. Malony, J. Cuny, S. Shende, and P. Beckman, "An IL Converter and Program Database for Analysis Tools," Proceedings of SPDT '98: ACM SIGMETRICS Symposium on Parallel and Distributed Tools, Aug. 1998.
- S. Shende, J. Cuny, L. Hansen, J. Kundu, S. McLaughry, O. Wolf, "Event and State Based Debugging in TAU: A Prototype," Proceedings of ACM SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 21-30, May 1996.
- S. Shende, B. Krishna, "Simulation of Concurrent Runtime Environment of PARAM on UNIX," in Proceedings of High Performance Computing, SRC 1994, Computer Society of India, pp. 14-18, June 1994.
- S. Shende, R. Talashikar, M. Bhandarkar, "Design and Implementation of imagePRO/NET," in V. Bhatkar, et. al. (editors), *Supercomputing Using Transputers*, Narosa, New Delhi, pp. 89-97, 1994.

ACKNOWLEDGEMENTS

I am indeed fortunate in finding great co-advisors in Janice Cuny and Allen Malony. I thank them for their guidance, for teaching me how to do research, for giving me the freedom to explore new ideas, for providing a stimulating environment, and for their friendship. They were the driving force behind this dissertation. I thank Bernd Mohr and Zena Ariola for providing valuable insights. I thank Pete Beckman for giving this research an important direction at a critical time and Rod Oldehoeft for encouragement.

I thank Clara Jaramillo for helping me understand the inner workings of the optimizing lcc compiler. I thank my colleagues Kathleen Lindlan, Robert Ansell-Bell, Steven Hackstadt, Timothy Sheehan, and Stephen McLaughry for their support.

I thank the following people for giving me an opportunity to apply this work to their projects: Dennis Gannon, Larry Snyder, Chris Johnson, Roger Haydock, Mary Lou Soffa, Thomas Fahringer, John Reynders, Julian Cummings, Steve Karmesin, William Humphrey, Suvas Vajracharya, Stephen Smith, Steven Parker, J. Davison de St. Germain, Wolfram Arnold, Todd Veldhuizen, Federico Bassetti, and Craig Rasmussen.

This research was sponsored by the U.S. Department of Energy, the Los Alamos National Laboratory, and the National Science Foundation.

Finally, I thank my wife, Kirsten, and my parents, Suhas and Suresh. Kirsten's love and encouragement helped me achieve my goals.

To my Mother.

TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION	1
1.1 At What Level Within the Environment Should Performance Instrumentation be Done?	3
1.2 How Do We Relate the Performance Data Back to Constructs that the User Understands?	4
1.3 How Can We Implement Fine-Grained Performance Instrumentation at the Source Level?	6
1.4 Contributions	7
1.5 Organization of the Dissertation	7
II. RELATED WORK	9
2.1 Introduction	9
2.2 Post-Mortem Performance Evaluation	10
2.2.1 Profiling	10
2.2.2 Event Tracing	16
2.3 On-line Performance Evaluation	21
2.4 Conclusions	24
III. MULTI-LEVEL INSTRUMENTATION	26
3.1 Introduction	26
3.2 Instrumentation Levels	27
3.2.1 Source-Level Instrumentation	29
3.2.2 Preprocessor Level	32
3.2.3 Compiler-Based Instrumentation	34
3.2.4 Library Level Instrumentation	36
3.2.5 Linker Level Instrumentation	38
3.2.6 Executable Level Instrumentation	39
3.2.7 Runtime Instrumentation	41
3.2.8 Virtual Machine Level Instrumentation	43
3.3 Multi-Level Instrumentation Model	45

Chapter	Page
3.4 Case Study: Integration of Source, Library and JVM Level Instrumentation	48
3.4.1 Virtual Machine Level Instrumentation	50
3.4.2 Library-Level Instrumentation	53
3.4.3 Source-Level Instrumentation	54
3.4.4 Selective Instrumentation	56
3.4.5 Tracing Hybrid Executions	57
3.5 Conclusions	61
IV. MAPPING PERFORMANCE DATA	62
4.1 Introduction	62
4.2 The Need for Mapping	63
4.3 Our Approach	68
4.4 Semantic Entities, Attributes, and Associations (SEAA)	71
4.5 Case Study: POOMA II, PETE, and SMARTS	74
4.5.1 PETE	75
4.5.2 SMARTS	79
4.5.3 Instrumentation Challenges Created by PETE and SMARTS ..	81
4.5.4 Mapping Challenges in POOMA	85
4.5.5 Our Mapping Approach	87
4.6 Case Study: Uintah	89
4.7 Conclusions	95
V. INSTRUMENTATION AWARE COMPILATION	97
5.1 Introduction	97
5.2 Issues in Instrumentation	98
5.3 Instrumentation-Aware Compilation	104
5.3.1 Prototype Implementation	107
5.3.2 Perturbation Issues	114
5.3.3 Results	115
5.3.4 Incomplete Source-Level Information	117
5.4 Case Study: Successive Program Transformations in ZPL	122
5.5 Conclusions	128
VI. CONCLUSIONS	130

Chapter	Page
APPENDIX: MAINTAINING SOURCE-LEVEL MAPPINGS	134
BIBLIOGRAPHY	141

LIST OF TABLES

Table	Page
1. Relative advantages and disadvantages of each instrumentation level	28
2. Types of upward mappings $S \rightarrow R$	70
3. Mapping information for three benchmarks	119

LIST OF FIGURES

Figure	Page
1. Levels of program transformations	4
2. Routine-based profiles are insufficient in dealing with complex transformations	5
3. Levels of program transformations	27
4. Tracking per-thread memory allocation in PaRP using source-level instrumentation	30
5. Multi-level instrumentation as implemented in the TAU performance system	46
6. TAU instrumentation for Java and the mpiJava package	51
7. Source-level instrumentation in an mpiJava program	54
8. Profile display of an integrated performance view of Life application	55
9. Tracing inter-thread message communication in mpiJava	58
10. Performance data from source, library and virtual machine levels of a Java application	59
11. Dynamic call tree display shows Java source, MPI, and virtual machine events	60
12. Scientist describes some scientific model	64
13. Profile without mapping	65
14. Profile with mapping	65
15. Bridging the semantic gap	66
16. Mapping instrumentation	67

Figure	Page
17. Parallel multi-dimensional array operations in POOMA.	77
18. Equivalent evaluation of array expressions in C	78
19. Parse tree for an expression (form taken from [41])	78
20. PETE can generate code that is comparable in performance to C	79
21. Compile-time (PETE) and run-time (SMARTS) optimizations employed by POOMA	80
22. Profile generated by a vendor-supplied performance tool.	82
23. Expression templates embed the form of the expression in a template name	84
24. Synchronous timers cannot track the execution of POOMA array statements.	85
25. One-to-many mappings in POOMA	86
26. Mapping costs of individual POOMA statements	88
27. Event traces of mapped iterates show the contribution of array statements ..	89
28. Task execution needs to be explored further in Uintah	90
29. Uintah node profile shows that task execution is a computationally intensive activity.	91
30. Mapping reveals the relative contribution of different tasks	92
31. Global timeline shows the distinct phases of task execution	93
32. Color-coded activity chart highlights the relative contribution of different tasks	94

Figure	Page
33. Summary chart	95
34. Four versions of a program, demonstrating the effect of code restructuring optimizations on the accuracy of a timer. Uninstrumented code is on the top; instrumented code is on the bottom. Unoptimized code is on the left; optimized code is on the right.	99
35. Using a timer to measure the cost of executing three statements	100
36. Unoptimized assembly code for routine f	101
37. Optimized assembly code for routine f	103
38. Traditional optimizing compiler	104
39. Instrumentation-aware compilation model is cognizant of optimizations	105
40. Structure of a typical optimizing compiler	107
41. Structure of an instrumentation-aware compiler	108
42. Structure of a de-instrumentor	110
43. Algorithm for fine-grained source-level performance instrumentation	112
44. Optimized code generated by the instrumentation-aware compiler	116
45. Performance data from an optimized program compiled with the instrumentation-aware compiler	118
46. Source code of an MPI application	120
47. Mapping table for a small portion of the code	121
48. ZPL's compilation model	123

Figure	Page
49. Multi-stage instrumentation-aware compilation in ZPL	124
50. Source code of a ZPL program	125
51. Mapping Table from the ZPL compiler	125
52. Optimized C program generated by the ZPL compiler	126
53. Instrumented C code the ZPL application	127
54. Performance data from the optimized ZPL program compiled by the instrumentation-aware compiler	128

CHAPTER I

INTRODUCTION

Computational scientists are finding it increasingly difficult to understand the behavior of their parallel programs. Fueled by ever increasing processor speeds and high speed interconnection networks, advances in high performance computer architectures have allowed the development of increasingly complex large scale parallel systems. To program these systems, scientists rely on multi-layered software environments with compiler, parallel runtime system and numerical library support for high-level programming abstractions. These environments shield the users from the intricacies of low-level operations while improving program performance with more aggressive, architecture-specific optimizations. While necessary to reign in programming complexities though, these advanced software environments distance the user from system performance behavior. The semantic gap between the abstractions used in programming and the low-level behavior of the transformed code make it difficult for scientists to observe, analyze and understand their code. This is a particular problem when, as is often the case, acceptable levels of performance are not met. Scientists as well as software developers, need the support of software tools to help identify the source of performance anomalies.

To observe the behavior of a program, instructions must be inserted into the program to record interesting aspects of its execution and provide insight into its performance characteristics. The performance space of a program can be viewed along three distinct instrumentation and measurement axes:

- *How* are performance measurements defined and instrumentation alternatives chosen;
- *When* is performance instrumentation added and/or enabled (pre-compile-time, compile-time, link-time, runtime); and
- *Where* in the program performance measurements are made (granularity and location);

Scientists need a flexible way of designing performance problem solving experiments in order to navigate through this potentially large multi-dimensional performance space [72]. Each experiment is intended to generate performance data and results that can bring insight to performance concerns. However, performance experimentation must resolve a performance observation dilemma [71]: too much instrumentation can generate too much performance data that can perturb a parallel application and modify its behavior, while too little instrumentation may not reveal enough detail to be useful. Prudent choices must be made at the instrumentation phase, to generate just the right amount of instrumentation that would highlight a performance anomaly, without unduly perturbing the application. This requires choosing appropriate points along the instrumentation and measurement axes to compose a performance experiment. If a tool restricts the choices in selection and control of performance experimentation, it will be difficult to use such a tool to explore and interpret the performance of a complex parallel application. A measure of a tool's flexibility is in the

freedom it offers in performance experiment design by choosing points along the three *how/when/where* axes of the performance space by providing advances in performance instrumentation and mapping techniques and technology. Our work aims to provide greater flexibility in exploring the performance space. In providing greater flexibility, there are three issues that arise: how to get complete coverage of observable data from the environment, how to maintain the user's level of abstraction, and how to provide fine-grained access to program data and constructs.

1.1 At What Level Within the Environment Should Performance Instrumentation be Done?

Programs undergo many phases of transformations before execution as they move through the *source-compile-link-execute* levels as shown in Figure 1. Instrumentation can be introduced at any level. Until now, however, tools have been designed to gather performance data at only one level. Thus, they do not afford complete observation coverage. For example, a hybrid parallel application that uses message passing for inter-task communication and multi-threaded parallelism within tasks cannot be fully observed by monitoring communication events.

- *We present a multi-level instrumentation framework that would allow us to gather data appropriate to any code transformation and runtime execution level and seamlessly move between levels as the focus of performance exploration changes.*

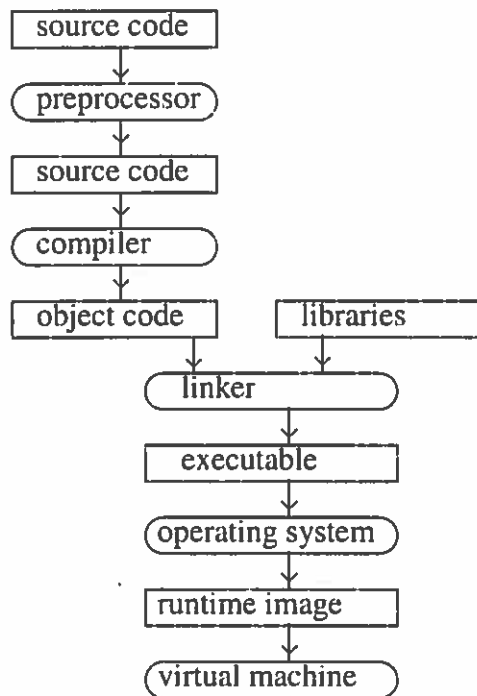


Figure 1. Levels of program transformations

Targeting instrumentation at multiple levels of program transformations requires cooperation between multiple instrumentation interfaces. This allows us to support multiple execution and measurement models to provide flexibility in composing a performance experiment and to cover observable behavior more completely.

1.2 How Do We Relate the Performance Data Back to Constructs that the User

Understands?

Typically, tools present performance data in terms of entities available at the level where it was gathered. The form of these constructs is often very different from the source code constructs. For example, in Figure 2 we see how high-level array expressions in C++

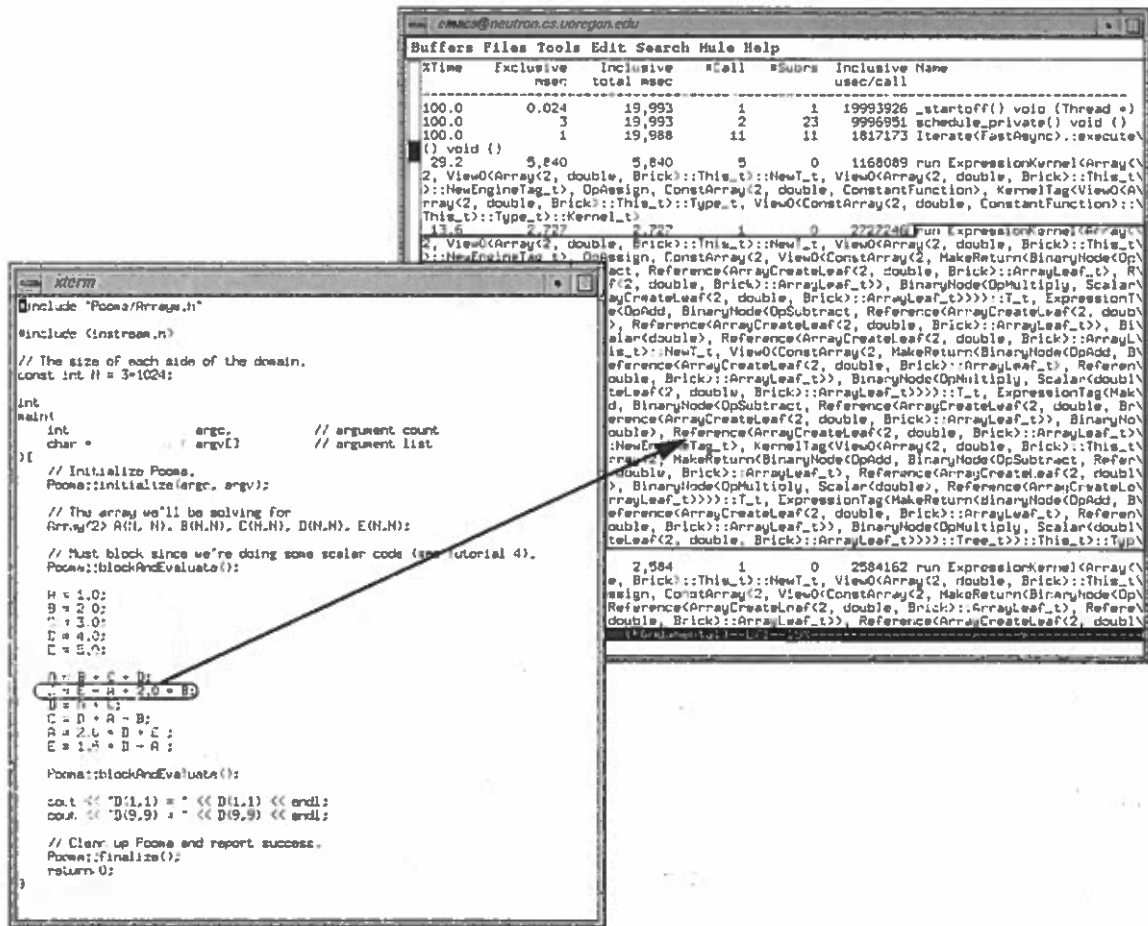


Figure 2. Routine-based profiles are insufficient in dealing with complex transformations

[101] are transformed during compilation into low-level routines that have an incomprehensible (“mangled”) form. In the figure, the array expression “C= E - A + 2.0 * B;” circled in the source code is represented in the performance output by the large, outlined expression in the middle of the figure. Relating this expression back to the above statement is tedious (if not impossible), even for an expert familiar with the inner workings of the compile time and runtime optimizations employed by the numerical framework.

- *Our work on performance mappings allows us to relate lower-level abstractions back to the original higher-level constructs in the presence of program transformations.*

The techniques relate semantic constructs to the program data, enabling collected performance data to be correlated with the high-level language constructs that are meaningful to the user.

1.3 How Can We Implement Fine-Grained Performance Instrumentation at the Source

Level?

Performance experimentation is an iterative process in which scientists often want to refine the focus of their instrumentation and measurement to selectively highlight different aspects of program execution. Most tools support only routine-based instrumentation. This allows tools a degree of language independence, but it limits the user's ability to make fine-grained measurements at the source code level. To fully explore the performance of a program, scientists must be able to refine the focus of instrumentation so that they can relate performance data to semantic entities as small as individual statements. Unfortunately, such source-based instrumentation may conflict with the optimizations performed by the compiler. Optimizers routinely move code during transformation, making it difficult to accurately attribute the cost of statements that are moved out of the scope of timing focus. A compiler may also assume that an instrumentation call modifies global variables, and thus it may fail to apply optimizations that it would have in the absence of instrumentation.

- *In this work, we develop a model of instrumentation-aware compilers that generate optimized code that performs correct measurements.*

Such a scheme requires a model for attributing costs and an algorithm that correctly instruments the optimized code using mappings that relate post-optimized code to source-level statements.

1.4 Contributions

To address the limitations of complete coverage, user-level abstractions, and granularity of instrumentation, this dissertation contributes:

- a strategy for utilizing multi-level instrumentation to improve the coverage of performance measurement in complex, layered software;
- techniques for mapping low-level performance data to higher levels of abstraction in order to reduce the semantic gap between user's abstractions and runtime behavior; and
- the concept of instrumentation-aware compilation that extends traditional compilers to preserve the semantics of fine-grained performance instrumentation despite aggressive program restructuring.

1.5 Organization of the Dissertation

Chapter II presents work related to this area. Chapter III presents our multi-level instrumentation model and a prototype implementation. It includes a case study of a

message passing application written in Java [110] that highlights the benefits of instrumentation at the source level, at the library level, and at the virtual machine level. Chapter IV introduces our model for mapping low-level information to user-level abstractions. It includes two case studies that show how such a model could be effectively applied. Chapter V describes instrumentation-aware compilation that extends traditional compilers to preserve the meaning of instrumentation calls. A prototype compiler is presented in detail. It includes a case study in instrumenting a high-level ZPL program requiring two levels of compilation. We demonstrate how the instrumentation-aware compilation model can be applied at both stages, along with multi-level instrumentation to map the performance data to higher-level abstractions that a ZPL programmer may understand. Finally, Chapter VI presents conclusions and directions for future research in this area.

CHAPTER II

RELATED WORK

2.1 Introduction

The process of improving the performance of a program is cyclic. It starts with observing the behavior of an application in terms of its performance characteristics. Analysis of this behavior enables performance diagnosis, that is, the development and evaluation of a hypothesis for poor performance. The hypothesis leads to the determination of program changes and the efficacy of those changes is evaluated again in terms of observed performance characteristics. The ability to observe the performance of an application is thus fundamental.

In order to observe performance, additional instructions or probes are typically inserted into a program. This process is called instrumentation. The execution of a program is regarded as a sequence of significant events. As events execute, they activate the probes which perform measurements (such as calculating the duration of a message send operation). Thus, instrumentation exposes key characteristics of an execution. Performance evaluation tools present this information in the form of performance metrics.

Performance bottleneck detection tools go one step further and automate the process of identifying the cause of poor performance.

In this chapter, we present the contributions of various projects in this field. For the sake of classification, we divide the performance observation tools in two main categories: post-mortem and on-line performance tools. Post-mortem tools, discussed in Section 2.2, help the user characterize the behavior of the application by analyzing the results of an empirical performance experiment after the application terminates. While this approach is useful for studying the behavior of the application, it has limitations in dealing with long-running applications. For this class of problems, tools for on-line performance monitoring, discussed in Section 2.3, are better suited.

2.2 Post-Mortem Performance Evaluation

Post-mortem performance evaluation tools for parallel programs traditionally fall into two categories: profiling and event-tracing. We discuss each separately.

2.2.1 Profiling

Profiling tries to characterize the behavior of an application in terms of aggregate performance metrics. Profiles are typically represented as a list of various metrics (such as wall-clock time) that are associated with program-level semantic entities (such as routines or statements in the program). Time is a common metric used but any monotonically

increasing resource function can be used. The two main approaches to profiling include *sampling-based* profiling and *instrumentation-based* profiling [40].

2.2.1.1 Sampling-based Profiling

Sampling-based profiling periodically records the program state and, based on measurements made on those states, estimates the overall performance. Prof [38][114] is typical. It uses a hardware interval timer that generates a periodic interrupt after a certain amount of time elapses. At that interrupt, the process state is sampled and recorded. Based on the total number of samples recorded, the interval between the interrupts and the number of samples recorded when a particular code segment is executing, statistical techniques are employed to generate an estimate of the relative distribution of a quantity over the entire program. Since events that occur between the interrupts are not seen by the performance tool, the accuracy of this approach depends on the length of the interval which typically varies from 1 to 30 milliseconds. Increasing the interval reduces the fixed measurement overhead of profiling, but decreases its resolution. As processor speeds continue to increase, however, sampling based on time can lead to sampling errors and inaccuracies. An alternative, implemented in the unix profiling tool Speedshop [114] uses sampling based on the number of elapsed instructions between two interrupts. This removes the dependency on the clock speed. Another alternative is to use the hardware performance monitors provided by most modern microprocessors.

Hardware performance monitors can be used both as interval markers and as sampled data. They are implemented as on-chip registers that can transparently count quantities, such as the number of instructions issued, cycles completed, floating point operations performed, number of data and instruction cache misses seen. Most microprocessors provide two or three registers (which implies that two or three counters can be used simultaneously), a mechanism to read and reset the registers, and an interrupt capability when a counter overflows. Tools such as Intel's VTune [47], SGI's SpeedShop [114] and Compaq's DCPI [3] use interrupt intervals based on the number of instructions issued. SpeedShop allows the use of other quantities measured by the R10000+ processor hardware performance counters [141] such as generation of an interrupt after a given number of secondary data cache misses. It provides two preset experiment types for each of the counters that use prime numbers for the elapsed counts; if two profiles of the application using different prime numbers are similar, then the experiment is deemed to be valid. PCL [14] and PAPI [18] provide uniform interfaces to access hardware performance counters on multiple platforms. Performing measurements exclusively in hardware is non-intrusive but the software-based instrumentation to access their data is not. A combination of both software and hardware-based measurement schemes is intrusive, but less so than if the measurements were made completely in software.

In sampling-based profiling, either the program counter (and thus the currently executing routine) is sampled or the callstack is sampled. In the first case, the program counter is translated to the currently executing block of code and the number of samples associated with that block of code is incremented. This approach is used in SGI's

Speedshop [114]. In the second case, the callstack is sampled, and the time for the block of executing code is incremented. This allows the computation of both exclusive and inclusive time. Exclusive time is the time spent executing the given routine not including time spent on other routines that it called. Inclusive time includes the contributions from callees computed by traversing the callstack. Based on the number of samples taken in a given block, the time spent in the block is estimated using the sampling frequency and the processor clock frequency.

The *gprof* [37] tool extends *prof* to show the caller-callee relationship using call graphs instead of flat profiles. Based on the number of times a routine (parent) invokes another routine (child), *gprof* estimates the contribution of a routine along a caller-callee edge in the call-graph. It divides the time spent in a routine among its callers in proportion to the number of times the routine was called by them. It reports this call-graph information to one level. While this may be misleading in cases where the different invocations of the routines perform unequal work, it highlights the importance of call-graphs.

The *Cpprof* tool [40] uses call paths, or sets of stack traces (sequences of functions that denote a thread's calling order at a given instant of time) to present performance in terms of call sequences between routines. This is the logical next step from *gprof*-based profiles which can be considered as call paths of length 2. Call path refinement profiles is a novel reduction technique that can report the cost of all edges that represent calling sequences between any two entities in the call graph. It can aggregate the output of performance queries to limit uninteresting details and answer the call path related question

succinctly. However, this degree of flexibility requires a significant amount of memory (as compared to gprof) to store every unique stack trace in memory. In Hprof [132], a profiler for the Java language, samples of stack traces are used (similar to call path profiling approach) to construct a CPU time profile. It uses instrumentation at the virtual machine level to support profiling of multi-threaded Java programs.

Compaq's DCPI [3] tool uses continuous sampling of the operating system and applications that execute under it. It randomizes the interval between samples to produce an accurate report of where the program spends its time down to the individual instruction level. DCPI reports, for each instruction, the frequency of its execution, the cycles per instruction it spent at the head of the queue waiting to execute, and the causes for its stalls.

Although, sampling-based schemes suffer from incomplete coverage of the application (especially for applications that have a short life-span), they have a distinct advantage of fixed, low instrumentation overhead and consequently reduced measurement perturbation in the program.

2.2.1.2 Instrumentation-Based Profiling

With instrumentation-based profiling, measurements are triggered by the execution of instructions added to the code to track significant events in the program such as the entry or exit of a routine, the execution of a basic block or statement, and the send or receipt of a message communication operation. Typically, such profilers present the cost of

executing different routines in a program. Examples include Quantify [97], Optimize It! [133], TAU [74], Speedshop [114].

Of particular interest is the instrumentation code itself. Tools insert instrumentation code during different phases of compilation and execution. Quantify, a commercial profiler from Rational Software, uses an object-rewriting technique to insert instrumentation in a C++ or C program. It does not require any application source code and can even instrument libraries with runtime linkages (such as dynamic linked libraries). This instrumentation is triggered at routine transitions and performance measurements are made at these points.

Optimize It!, a profiler for the Java language, places instrumentation at the virtual machine level [132] to generate summary statistics of performance metrics. It loads an in-process profiling agent that registers a set of call-back routines for program and virtual machine events. These routines are invoked by the interpreter in the virtual machine as it executes the Java bytecode. Thus, instrumentation code is added at runtime, without any modifications to the source code, the bytecode or the virtual machine image. Optimize It! allows a user to choose from both sampling and instrumentation-based profiling techniques.

Speedshop's ideal time experiment uses Pixie [134][116] to re-write a binary image to count basic blocks in the executable. These counts are then converted to idealized exclusive times using a machine model. The ideal time represents the best-case executions, and actual execution may take longer as it does not account for delays due to cache misses and stalls.

The PT project from the University of Wisconsin differs from the above profilers in an important aspect: instead of counting basic blocks at entry and exit points (thereby recording each instance, as in Pixie), it inserts instrumentation along the edges of the program's control flow graph (CFG)¹ for optimal placement of instrumentation. In their work, they report a reduction in profiling overhead by a factor of four or more [10]. In short, instead of instrumenting nodes in the control flow graph, they choose to instrument selectively along edges, based on a heuristic that uses either performance data from a prior run or the structure of the CFG to guide its instrumentation decisions. Like Quantify and Pixie, QP also instruments an executable using binary re-writing techniques.

TAU [113] has the ability to instrument at multiple stages of code transformation. It allows for routine, basic-block and statement-level timers to generate profiles as well as event-traces. For profiling, it allows a user to choose from measurement options that range from wallclock time, CPU time and process virtual time to hardware performance counters [111].

2.2.2 Event Tracing

While profiling is used to get aggregate summaries of metrics in a compact form, it cannot highlight the time varying aspect of the execution. To study the post-mortem spatial and temporal aspect of performance data, event tracing, that is, the activity of

¹ A CFG connects basic blocks (represented as nodes) to show the flow of control among instructions in a routine. A basic block contains sequences of instructions that have a single entry and exit point and no branches in or out of the block.

capturing an event or an action that takes place in the program, is more appropriate. Event tracing usually results in a log of the events that characterize the execution. Each event in the log is an ordered tuple typically containing a time stamp, a location (e.g., node, thread), an identifier that specifies the type of event (e.g., routine transition, user-defined event, message communication, etc.) and event-specific information. Event tracing is commonly employed in debugging [24][60][109][83][84][85][86][87][88][142] and performance analysis [34][70][91][99][73][82][139][140].

In a parallel execution, trace information generated on different processors must be merged. This is usually based on the time-stamp which can reflect logical time [65] or physical time. The logical time [61] uses local counters for each process incremented when a local event takes place. The physical time uses reference time obtained from a common clock, usually a globally synchronized real-time clock [45].

Trace buffers hold the ordered and merged logs. They can be shared or private. On shared memory multiprocessors, multiple threads can easily access a shared trace buffer providing an implicit ordering and merging of trace records and maximizing the buffer memory utilization. Private buffers can be used in both shared memory and distributed memory systems. Since the buffer is private, there is no contention for it, but the trace buffer memory is not optimally utilized due to varying rates of event generation in tasks. In either case, the trace buffer needs to be periodically flushed to disk and this can be done either by the local tasks or by an external trace collector task that shares the trace buffer with the rest of the tasks. The trace buffer can be in the form of a vector or a circular buffer (ring). In TAU [113] a private trace buffer in the form of a vector is used without a trace

collector task, whereas in BRISK [7], a shared circular trace buffer is used with a collector task. There are two policies used in flushing the buffer to the disk: flush-on-full and flush-on-barrier. In the former, the contents of a trace buffer are flushed to stable storage when the buffer is filled, as in TAU [81]; in the latter, all tasks flush the trace buffer to disk when the parallel program reaches a synchronization point, as in Split-C [39]. When the program terminates, trace buffers are flushed to disk as well. In Cedar's tracing system [71], a user can select between a statically allocated fixed size trace buffer or dynamically allocated trace buffers using linked buffer blocks, and the user can choose where the trace buffers will be stored in the Cedar's memory hierarchy. Also, runtime trace I/O can be selected that causes the trace collection task to run concurrently with the program task and the trace I/O can be sent to a file or to a task on a remote computer over a network. After the events are logged, these traces often need some form of post-processing [137]. This could take the form of merging event traces from multiple tasks, and/or conversion to another trace format.

Instrumentation can perturb an application and modify its behavior. Event tracing is the most invasive form of instrumentation because the volume of data generated is quite large and thus it may perturb the application more than other forms of performance measurement. Malony [71] showed that perturbation analysis can be done in two phases. Firstly, given the measured costs of instrumentation, the trace event times can be adjusted to remove these perturbations. Secondly, given instrumentation perturbations that can reorder trace events, the event sequences need to be adjusted based on knowledge of event dependencies, maintaining causality. Thus, both time-based and event-based perturbation

models are used to compensate for the perturbation. While analyzing the traces, AIMS [139] and [104] try to re-create the execution of un-instrumented versions of message passing programs by compensating for this intrusion. This is an important step in ensuring that traced event orderings are preserved while removing the perturbation (e.g., ensuring that a message is not received before the corresponding send operation).

The Pablo project [99] from the University of Illinois, Urbana Champaign, uses event tracing to develop performance tools for both message passing and data parallel programs. It provides a source code instrumentation interface that inserts user-specified instrumentation in the program. Pablo de-couples the meaning of the performance data from its structure using the self defining data format (SDDF) [99] for generating trace-logs. This de-coupling forms the basis for user-defined data visualization where data reduction and display modules can be plugged together for specialized data analysis. The user can interconnect nodes for performance data transformation, to form an acyclic data analysis graph. Performance data flows through this graph and performance knowledge is extracted from this data. The performance metrics are correlated with source code locations in the application and presented alongside the source code in the SvPablo [28] interface. SvPablo is an integrated environment that allows for instrumentation, supporting both manual and automatic modes using parsers for Fortran dialects and C, and analysis of performance data. The analysis graphical user interface supports a diverse collection of data presentation modules [42] such as bargraphs, bubble charts, strip charts, contour plots, interval plots, kivi diagrams, 2-D and 3-D scatter plots, matrix displays, pie charts, and polar plots [100]. The Pablo group has explored support for display technologies such

as sound [75] and immersive virtual environments [123] for displaying higher dimensional data.

Another significant contribution of the Pablo project is in the area of integration of performance tools with compilers for high-level parallel languages, such as Fortran-D [1]. Using information visible during program transformations in the compiler, it can generate a wealth of performance data that corresponds to source code abstractions. The Cray MPP Apprentice [96] tool for profiling also benefits from a similar integration with compilers.

All of these tools must manage the large volume of trace data. Pablo allows the user to invoke a user defined event handler to perform on-the-fly data reduction and supports automatic throttling of event data generation to avoid swamping. When a threshold is reached for an event, event recording is disabled or replaced by periodic logging. Other techniques to control the volume of performance data, include selectively tracing routines and statements that belong to a set of profile groups specified during program execution, as in TAU [113], and call-graph based techniques, as in QPT [64]. In QPT, the tracing counterpart of the QP tool described earlier, weights on edges of the control flow graph dictate whether an edge is instrumented or not [10].

We now shift our focus from post-mortem approaches, such as profiling and tracing, to on-line performance evaluation approaches.

2.3 On-line Performance Evaluation

The motivation for runtime performance monitoring are many: it is suitable for long-running programs or server processes that are not meant to terminate; it does not have to contend with a huge volume of performance data as in event-tracing as performance analysis occurs concurrently with program execution; and it highlights the performance data for the current execution state of the application which can be related to other system parameters (e.g., disk activity, CPU and memory utilization, load on the system, and network contention). Examples of on-line performance evaluation tools include the program monitor used by the Issos system [92] that performs quality-of-service (QoS) measurements, Autopilot [102][131] for use in adaptive resource management, and TAU [107][112] for on-line-monitoring of profile data.

The goal of performance evaluation is to diagnose a performance problem. To aid in detecting performance bottlenecks, the Paradyn [79] project from the University of Wisconsin automates the process of performance bottleneck detection. Paradyn searches for performance bottlenecks using the W^3 search model [43]. It tries to answer *why* a program is performing poorly, *where* its bottleneck lies, and *when* it occurs. These why, where and when questions make up the axes of a performance exploration space. As the application executes and generates performance data, Paradyn's performance consultant evaluates a hierarchical set of rules to refine a performance hypothesis (e.g., is the program CPU bound?). Paradyn inserts and deletes instrumentation code to perform

measurements while the application executes using a runtime-code patching interface, DyninstAPI [43][20][44][45], for instrumentation.

Paradyn provides a hierarchical display of performance data, refined from its predecessor, IPS-2 [80], to show the progress of the search for bottlenecks. The hierarchy consists of a root node which represents the program and other nodes include machines, source code, and processes. As the search progresses, nodes and edges in the tree are expanded and pruned until a bottleneck is identified.

Paradyn-J [89][90] is an extension of Paradyn that detects bottlenecks in interpreted, just-in-time compiled, and dynamically compiled programs. In the case of Java, there is an interdependence between the application program and the interpreter embedded in the virtual machine. Paradyn-J provides a representational model that captures the interaction between the two and exposes it to performance measurement tools. In Paradyn-J, two hierarchical tree displays represent the performance characteristics of the application and the virtual machine. This presents valuable performance data for both the application developer as well as the virtual machine developer, and it allows for a correlation of performance data while searching for performance bottlenecks.

ParaMap [49][48] is a closely related project that aims to map low-level performance data back to source-level constructs for high-level parallel languages. It introduces a noun-verb (NV) model to describe the mapping from one level of abstraction to another. A noun is any program entity and a verb represents an action performed on a noun. Sentences, composed of nouns and verbs, at one level of abstraction, map to

sentences at higher levels of abstraction. ParaMap uses three different types of mappings: static, dynamic and a new technique based on a data structure called the set of active sentences. Our work builds upon these mapping abstractions as described in Section 4.3.

Some tools incorporate the use of historical performance data collected from prior executions. Paradyn, for example, can make use of such data in speeding up the search for bottlenecks [56]. This addresses a broader problem of managing performance data from multiple experiments and allowing a user to examine the evolution of the program in terms of its performance data. Within Pablo, historical data is used in developing models that predict the total execution time of a parallel program as a symbolic expression using the number of processors, problem size, and other system specific parameters [77]. This is useful in determining how well a program will scale on a parallel machine. Predicting the scalability of a program has also been addressed in extraP [106] and Cray ATExpert [59].

In contrast, P³T (Parameter based Performance Prediction Tool) [29] estimates key performance parameters of a data parallel program, rather than total execution time. It uses a combination of static and dynamic program information to estimate performance parameters for parallel programs such as data locality in terms of cache misses [30], load balance in terms of work distribution, communication overhead in terms of the volume of data transferred, network contention and bandwidth. These estimates are fed into the Vienna Fortran compiler [143] to enable it to generate more efficient code. While Pablo uses event tracing, P³T uses profiles.

Tools can also be distinguished by the intended users of their output. In Pablo, the primary consumer of scalability data is the user, while in P³T, it is the compiler, in QP/QPT, it is the instrumentor, and in Paradyne, it is the performance tool's bottleneck search engine.

2.4 Conclusions

In this chapter, we classify performance tools into two broad categories: ones that present performance data after the program terminates, and ones that do this during execution. Typically, a performance tool embeds:

- an *instrumentation model* that defines how and at what stage of program transformation, are performance measurement probes inserted, activated and controlled;
- a *performance measurement model* that defines what performance measurements are recorded and the form of this performance information;
- an *execution model* that correlates events that are the focus of performance observation;
- a *computation model* that defines the computing environment under which performance measurements are made;
- a *data analysis model* that specifies how measured data is processed;
- a *presentation model* that defines the structure and form of gathered performance data and how it is presented to the user; and

- an *integration model* that describes how diverse performance tool components are configured and integrated in a cohesive form.

In the remainder of this dissertation, we focus on the instrumentation, performance measurement, and execution models. The ability to instrument a program effectively is critical to being able to compose a performance experiment. Instrumentation enables performance observation. As high performance computing environments continually evolve to more complex forms, performance tools will need to provide more effective instrumentation alternatives.

CHAPTER III

MULTI-LEVEL INSTRUMENTATION

3.1 Introduction

As high performance computers grow increasingly complex, parallel programming paradigms and environments must also develop to improve software efficiency. Tools to quantify and observe application performance behavior, however, have lagged behind. Applications that use high-level parallel languages and domain-specific numerical frameworks, heterogeneous execution models and platforms, multiple compilation models and multi-level optimization strategies pose interesting problems for instrumentation. Typically, tools have targeted only a single level of program analysis and transformation (see Figure 3) for instrumentation which greatly reduces access to program information.

In this chapter, we discuss the advantages and disadvantages of each instrumentation level. We then present a multi-level strategy that targets multiple cooperating instrumentation interfaces using a common application programmers interface (API) for coherent, comprehensive performance instrumentation and measurement. Thus, tools can gather performance data appropriate to any instrumentation level and seamlessly move between levels as the focus of performance exploration

changes. Finally, we present a case study that demonstrates the efficacy of a multi-level instrumentation model by simultaneously using instrumentation at three levels.

3.2 Instrumentation Levels

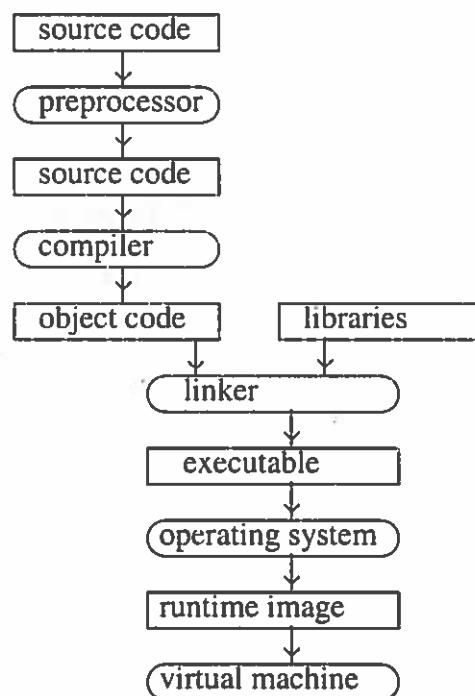


Figure 3. Levels of program transformations

The source code of a program undergoes a set of transformations before it executes, as shown in Figure 3. Instrumentation can be added to the program at any of these levels, each of which imposes different constraints and opportunities for extracting information, as shown in Table 1. As information flows through these levels, different aspects of the program can be revealed. As we move from source code instrumentation

TABLE 1: Relative advantages and disadvantages of each instrumentation level

Level	Examples	Advantages	Disadvantages
Source	Ariadne, JEWEL, TAU	Language portability Domain-specific abstractions Arbitrary points Fine granularity	Re-compilation required Tedious manual task Instrumentation errors Availability of source code Optimizations
Pre-processor	PDT, Sage++, SUIF	Source-to-source translation Automates instrumentation	Re-compilation required Availability of source code Static analysis constraints Language specific
Compiler	CRAY, GNU, lcc, SGI	Access to mapping tables, optimizations Fine granularity	Re-compilation required May not see all routines Presents low-level information
Library	Vampir- Trace	No need to re-compile Interposition, pre- instrumented libraries	Re-linking required Only applicable to libraries
Linker	DITools, Mahler	Inter-module operations Runtime interposition	Modified linker, re-linking Limited coverage (DSOs)
Executable	PAT, Pixie, QPT	No source code required No re-compilation required Language independent	No source-mapping tables Coarse granularity Symbol table required
Runtime	Paradyn, DyninstAPI	No source code, re-compilation, re-linking or re-starting required Instrumentation code can change at runtime	Platform/OS/compiler/ file-format specific Coarse granularity Lacks source-mappings
Virtual machine	HProf, Optimize It!	VM events visible No changes to source, bytecode, or JVM required	Coarse granularity Not applicable to native libraries

techniques to binary instrumentation techniques, our focus shifts from a language specific to a more platform specific instrumentation approach [108]. No one level can satisfy all requirements for performance tools.

3.2.1 Source-Level Instrumentation

At this level, the programmer manually annotates his/her source code with calls to a measurement library. This approach is used in several tools [62][60][113]. During execution, the instrumentation code is activated to record the appropriate aspect of its behavior. Source-level instrumentation can be placed at any point in the program and it allows a direct association between language- and program-level semantics and performance measurements. Since most languages provide some form of cross-language bindings (typically in C), a common measurement API can be called by different languages with minimal changes. Hence, the same features that enable scientists to build cross-language programs can be used for instrumentation as well, enhancing tool portability.

Some languages, however, pose unique challenges for source-level instrumentation. Consider C++ templates. Templates provide for polymorphism at compile-time [118] and can be instantiated into different forms during code generation (e.g., instantiation based on types). This style of programming, commonly called *generic* and *generative programming* [27], has led to the creation of *active libraries* [129] that participate in code generation, as opposed to passive ones that just contain objects. In C++, source instrumentation only sees templates and not their post compilation instantiations, thus it must rely on run-time type information [113][69]. Similar problems exist with other class-based languages and suggests that a multi-level instrumentation approach is necessary.

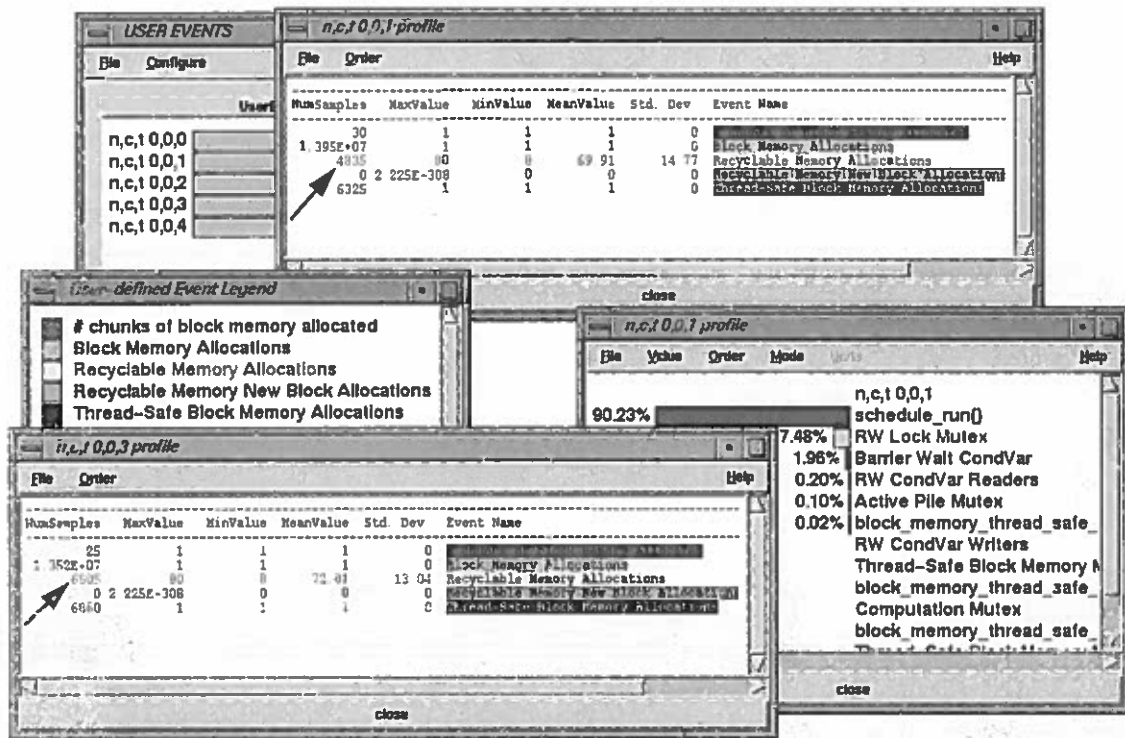


Figure 4. Tracking per-thread memory allocation in PaRP using source-level instrumentation

Source-level instrumentation has several advantages. It allows the programmer to communicate higher-level domain-specific abstractions to the performance tool. This is especially useful if a tool cannot automatically infer such information. For example, Figure 4 shows the use of user-defined events to track different types of memory allocations in a multi-threaded application, an implementation of the parallel dynamic recursion method in PaRP [5]. The figure shows (in highlighted text) that 4835 samples of *Recyclable Memory Allocations* are collected on thread 1, whereas the corresponding number for thread 3 is 6805 (highlighted by arrows). A programmer can communicate such events by annotating the source code at appropriate locations with instrumentation

calls. This is easily done at the source level, but may be significantly more difficult elsewhere. Once the program undergoes a series of transformations to generate the executable code, specifying arbitrary points in the code for instrumentation and understanding program semantics at those points may not be possible.

Another advantage of this level is that once an instrumentation library targets one language, it (theoretically) provides portability across multiple compilers for that language, as well as across multiple platforms. That is, the API is independent of details below the compiler such as operating system dependencies and object file formats.

Unfortunately, source-level instrumentation has several limitations. It requires access to the source code, which may not be available for some libraries. Source code needs to be re-compiled every time the instrumentation is modified. Adding instrumentation is a tedious manual task that introduces the possibility of instrumentation errors that produce erroneous performance data. For example, a user may overlap timers in the source code. While syntactic errors can be caught during compilation, logical errors in instrumentation may be more difficult to detect; overlapping timers, for example, can only be detected at runtime [113]. Also, the presence of instrumentation in the source code can inhibit the application of optimizations as well as lead to the generation of semantically incorrect instrumentation code. These last issues are addressed in Chapter V of this dissertation.

3.2.2 Preprocessor Level

Some of the difficulties with source-level instrumentation can be overcome at the preprocessor level. A preprocessor implemented as a source-to-source translation tool typically expands header files and performs macro substitutions during compilation. Such source-to-source transformations can be used to automatically introduce instrumentation, alleviating the burden on the programmer.

Preprocessor level instrumentation is commonly used to insert performance measurement calls at routine entry and exit points in the source code. To do this, a tool first needs to parse the application source code and locate the semantic constructs to be instrumented, such as routines, loops or individual statements. To insert code, the instrumentor also needs an interface to the parsed internal representation of the source code. Tools such as PDT [69] for C++, C and Fortran90, Sage++ [15] for C++ and SUIF [66][136] for C and Fortran provide an object-oriented class library to access the data structures that represent the parsed intermediate form. Sage++ and SUIF, in addition, provide an API for manipulating the data structures to add or prune internal nodes. The DUCTAPE interface in PDT provides access to the list of routines, classes and templates, and their locations in the source code. We have developed a source-to-source instrumentor [69] using PDT that reads a C++ program, locates the routine and template entry points, inserts instrumentation annotations in the source code and re-writes the instrumented source code; that code is then compiled and linked with a performance measurement library. Sage++ allows an instrumentor to un-parse the annotated internal representation

directly into a C++ program. SUIF produces intermediate files containing instrumentation annotations that can be processed further to produce a binary image, or converted to a C representation. The portability, robustness of front ends, granularity of instrumentation, and the level of detail of program constructs that are visible through the interface is different for all three tools.

Like the source-level instrumentation, the preprocessor-based instrumentation requires the source code for instrumentation. While source code may not be available for some libraries, instrumentation at this level can still be used at call-sites, or points in the application source code where the library routines are called. Typically this is accomplished by a preprocessor that replaces the library routine call with a call to an instrumented version. This has been successfully used in the Pablo [99] system for Fortran and C to target I/O calls. For languages such as C and C++ which have a preprocessor within the compiler, a specialized preprocessor can be avoided. Instead, a header file can be used to define macros that re-define the native library routines (e.g., *open*) with instrumented routines (e.g., *tau_open*). During compilation, the compiler preprocessor replaces calls to proprietary library routines with calls to the instrumented, wrapper libraries which perform performance measurements and call the appropriate library routines. This scheme does not require access to the library source code. However, it does require a minor modification (addition of a header file) to the application sources and requires the instrumented version of the library to be linked with the other object files and libraries. The main limitation of this approach is that it can only capture information about instances of library calls at specific call sites that are re-directed. If a pre-compiled library

routine makes references to a wrapped routine, it is not possible to re-direct such a reference without access to its source code and re-compiling it with the appropriate header file.

Preprocessor-level instrumentation shares many of the same disadvantages as source-level instrumentation including the need for access to source code and re-compilation of source code. Also, some of the problems with manual instrumentation, such as interference with compiler optimizations are equally true for an instrumentor that re-writes the application source code prior to compilation.

3.2.3 Compiler-Based Instrumentation

A compiler can add instrumentation calls in the object code that it generates. Examples of performance tools that use compiler-based instrumentation include the Cray MPP Apprentice [96], SGI Speedshop [114], and Sun Workshop [119]. Since most transformations take place during optimization passes in the compiler, compiler-based instrumentation can tap into a wealth of program information. During compilation, different code transformations, as well as mapping tables are visible. The granularity of instrumentation that is applied at this level ranges from routines all the way down to basic blocks, statements, expressions and instructions. It is also possible to study the effects of code transforming optimizations at this level (see Chapter V).

A compiler can also insert instrumentation code directly into the object code output or use clever breakpointing schemes such as the fast-breakpoints [58], replacing an

instruction with a branch instruction to the instrumentation code. Most compilers include some form of performance instrumentation by providing command line profiling flags that trigger instrumentation code at runtime. Sampling-based profiling instrumentation is most commonly implemented in Fortran, C and C++ compilers [37]. The GNU compiler [36] also provides hooks into routine entry and exit events and can invoke a pre-defined routine at these points. When a source file is compiled with the *-finstrument_functions* command line option, each routine in the source file is instrumented and at its entry and exit points, a user-defined function is invoked with the routine address and call stack information as arguments. The addresses can be mapped back to a routine name with the unix *nm* tool, or a debugging library such as GNU's BFD (binutils) that describes such a mapping.

There are several advantages to instrumentation at the compiler level. The compiler has full access to source-level mapping information. It has the ability to choose the granularity of instrumentation and can include fine-grained instrumentation. The compiler can perform instrumentation with knowledge of source transformations, optimizations and code generation phases. The disadvantage of instrumentation at this level is that a compiler may not see all the routines that produce an executable image. It sees only the set of files that are passed to it. For the same reasons inter-procedural transformations have a limited scope at this level. If object files and libraries from different compilers can be linked together to form an executable file, instrumentation techniques may not cooperate or may be inefficient. This can lead to reduced instrumentation coverage. Another disadvantage of instrumentation at this level is that it may present some program constructs that the user cannot comprehend, thereby increasing

the semantic-gap between the user's understanding of his/her code and its runtime behavior.

3.2.4 Library Level Instrumentation

Wrapper interposition libraries provide a convenient mechanism for adding instrumentation calls to libraries. A good example of this approach is found in the Message Passing Interface (MPI) Profiling Interface [78]. MPI, a standard for inter-process message communication, is commonly used to implement parallel SPMD programs. As part of its definition, it includes alternative entry points for MPI routines. The MPI Profiling Interface allows a tool developer to interface with MPI calls in a portable manner without modifying the application source code and without having access to the proprietary source code of the library implementation. If such profiling hooks were required for all "compliant" implementations of standardized libraries, we would have an excellent basis for developing portable performance tools.

The MPI standard defines the native library routine with weak bindings and a name shifted interface. A weak binding allows two different routines with the same name to co-exist in a binary executable. If a tool re-defines the native call, it takes precedence. In this manner, a performance tool can provide an interposition library layer that intercepts calls to the native MPI library by defining routines with the same name (e.g., *MPI_Send*). These routines wrap performance instrumentation around a call to the name-shifted native library routine provided by the MPI profiling interface (e.g., *PMPI_Send*). The exposure of

routine arguments allows the tool developer to also track the size of messages and message tags. The interposition library can also invoke other native library routines, for example, to track the sender and the size of a received message within a wild-card receive call. Several tools use the MPI profiling interface for tracing message communication events [19]. We have used the profiling interface with TAU [74].

When a library does not provide weak bindings and name shifted interfaces for profiling purposes, it can still be profiled at the library level. One option is to instrument the source using a performance measurement API and provide separate instrumented versions of the same library; we have done this with the POOMA r1 library [113]. Another option is to use a preprocessor-based scheme for generating instrumented versions of the library; we have done this with the POOMA II library [69]. Alternatively, a compiler-based approach can be used; we have done this with the ZPL runtime system (Section 5.4).

Many of the options for providing pre-instrumented versions of a library require access to the library source code. When that is not available (e.g., for proprietary system libraries), wrapper library schemes can be used. All of these mechanisms, however, restrict instrumentation to code in the form of libraries; they cannot be extended to the entire application.

3.2.5 Linker Level Instrumentation

Instrumentation code can also be introduced at the runtime linker level. The linker is the earliest stage where multiple object files, including both statically- and dynamically-linked libraries are merged to form the executable image. Dynamically-linked libraries delay the symbol resolution to runtime, storing un-resolved external references, which are resolved by a dynamic linker when the application executes. The ability to do inter-module operations at link time [117] makes interesting instrumentation schemes possible.

To introduce instrumentation at the linker level requires some modification of the linking process. The Mahler system [134] modifies code at link-time and performs a variety of transformations such as inter-module register allocation, and scheduling of instruction pipeline. It can also add instrumentation for counting of basic blocks and address tracing. In contrast, DITools [105] address the dynamic linking problem. To instrument an un-modified executable at runtime, it replaces the dynamic linker with a system component that re-directs each un-resolved external routine reference to an interposition library layer. The user specifies the instrumentation code with respect to routines and spawns the executable within a special shell. The DI Runtime component modifies the references in the executable's linkage tables and calls the appropriate instrumentation routines before and after the library routine, allowing for the instrumentation of dynamic shared object routines.

The advantage of linker-level instrumentation approach is that it does not require access to the application source code. In the case of dynamic linking, it does not modify

the object code either. This advantage is also a possible limitation. The only program information that can be used is what is contained in the code modules. For dynamic linking, instrumentation can only be applied to routines that are in dynamically linked libraries and need runtime resolution of object references. Also, detailed knowledge of the file formats and linkage tables is required for porting the tool to new platforms, limiting portability.

3.2.6 Executable Level Instrumentation

Executable images can be instrumented using binary code-rewriting techniques, often referred to as binary editing tools [20] or executable editing tools [63]. Systems such as Pixie [134], EEL [63], and PAT [33] include an object code instrumentor that parses an executable and rewrites it with added instrumentation code. The Bytecode Instrumenting Tool (BIT) [25] rewrites the Java bytecode. PAT can be used for call site profiling and instrumentation at the routine level (e.g., for tracing routine calls [33]) as well as gathering routine level hardware performance statistics. Pixie can count basic blocks within a routine, EEL can instrument a routine's control flow graph and instructions, and BIT can be used to re-arrange procedures and re-organize data in the bytecode. In each case, the executable file is re-written with the appropriate instrumentation inserted.

Executable level instrumentation is not a simple matter. For example, consider Pixie. Pixie must deal with several problems. It requires access to the symbol table (which may not be present). Because it operates at basic block boundaries, it needs to perform

address correction for changing the destination of program counter relative branches, and direct and indirect jumps. Performing indirect jumps at the executable level requires a translation table for relating addresses in the original program to addresses in the rewritten program. This translation table is as large as the original code segment! Further, indirect jumps, which require a runtime translation of addresses require instrumentation code for computing the new address. Finally, Pixie needs three registers for its own use which it must take from the program. It chooses the three registers with the least static references and replaces them with virtual "shadow registers" in memory [134]. A reference to the shadow registers requires an indirect translation and consequently, has a runtime overhead.

EEL attempts to circumvent some of these problems by providing an API for analyzing and editing compiled programs. It allows a tool to re-order and instrument routines, control flow graphs within routines, and individual instructions. A control flow graph consists of basic blocks and instructions within basic blocks. Register-transfer level instruction descriptions used in EEL hide the details of the underlying machine architecture and simplify the process of creating portable executable editing tools. QPT2 [64] uses EEL for tracing address references.

The advantages of instrumentation at the binary level are that there is no need to re-compile an application program and rewriting a binary file is mostly independent of the programming language. Also, it is possible to spawn the instrumented parallel program the same way as the original program, without any special modifications as are required for runtime instrumentation [111].

A disadvantage of this approach is that compiler mapping tables that relate source-level statements to instructions are rarely available after a compiler produces an executable. So, a substantive portion of information is unavailable to tools that operate at the binary executable level. This can force the performance measurements to take place at a coarser granularity of instrumentation. Some compilers generate incorrect symbol table information [64] (e.g., when a data table is put in the text segment with an entry that makes it appear as a routine [63]) which can be misleading and further complicate the operation of tools at this level. Similar to link-time instrumentation, there is a lack of knowledge about the source program and any transformation that may have been applied.

3.2.7 Runtime Instrumentation

An extension of executable level instrumentation, dynamic instrumentation is a mechanism for runtime-code patching that modifies a program during execution. DyninstAPI [20] is one such interface for runtime instrumentation. Although debuggers have used runtime instrumentation techniques in the past, DyninstAPI provides an efficient, low-overhead interface that is more suitable for performance instrumentation. A tool that uses this API (also known as a mutator) can insert code snippets into a running program (also known as a mutatee) without re-compiling, re-linking or even re-starting the program. The mutator can either spawn an executable and instrument it prior to its execution or attach to a running program. DyninstAPI inserts instrumentation code snippets in the address space of the mutator. The code snippets can load dynamic shared

objects in the running application, call routines, as well as read and write application data, allowing it to be used in developing debugging and steering applications. DyninstAPI translates code snippets into machine language instructions in the address space of the mutator. It generates code to replace an instruction in the mutatee to a branch instruction to the instrumentation code. It uses a two step approach using short sections of code called *trampolines*. The replaced instruction calls a base trampoline which branches to a mini trampoline. A mini trampoline saves the registers and executes the code snippet(s) with appropriate arguments. Thereafter, it restores the original registers and calls the base trampoline. The base trampoline executes a relocated instruction and returns to the statement after the replaced instruction in the original code as described in [20]. The ability to insert and remove instrumentation makes it valuable for observing the performance of long running programs over a small period of time. It also allows a tool to insert specific instrumentation code that may only be known at runtime. DyninstAPI has been successfully applied in Paradyn [79]. The API is portable and independent of the target platform and language constructs. This makes it suitable for building portable performance tools. DyninstAPI overcomes some limitations of binary editing tools that operate on the static executable level. The main improvement is that the instrumentation code is not required to remain fixed during execution. It can be inserted and removed easily. Also, the instrumentation can be done on a running program instead of requiring the user to re-execute the application.

The disadvantages of runtime instrumentation are similar to those for binary editing. First, the compiler discards low-level mapping information that relates source-

level constructs such as expressions and statements to machine instructions after compilation. This problem is common with any executable code instrumentation tool. Second, although some instrumentation can be done on optimized code, the instrumentation interface requires the presence of symbol table information, typically generated by a compiler in the debugging mode after disabling optimizations. Third, the interface needs to be aware of multiple object file formats, binary interfaces (32 bit and 64 bit), operating system idiosyncrasies, as well as compiler specific information (e.g., to support template name de-mangling in C++ from multiple C++ compilers). To maintain cross language, cross platform, cross file format, cross binary interface portability is a challenging task and requires a continuous porting effort as new computing platforms and programming environments evolve. Again, this problem is also present with any executable code instrumentation tool.

3.2.8 Virtual Machine Level Instrumentation

Instrumentation at the virtual machine level has become an important area of research. A virtual machine acts as an interpreter for program bytecode. This style of execution is commonly found in the Java programming language [4]. The compilation model of Java differs from other languages such as C, C++ and Fortran90 that are compiled to native executables. Instead, a Java compiler produces bytecode. The Java Virtual Machine (JVM) interprets this bytecode to execute the Java application. Optionally, a Just-in-Time (JIT) compiler may be employed in the JVM to convert the

bytecode into native binary code at runtime. A JIT compiler interprets the bytecode and produces a native binary concurrently, switching between the two at runtime for efficiency. Alternately, a hot-spot compiler [122] may be used in the JVM to selectively convert compute-intensive parts of the application to native binary code. This approach requires the JVM to perform measurements to determine what code regions are compute-intensive and provide this feedback to the code-generator.

We use the model of compilation in Java to discuss issues for virtual machine level instrumentation. Any interpreted language system like Java is going to cause difficulties for the instrumentation approaches we have considered. In addition to Java's complex model of compilation, it has a multi-threaded model of computation, and the Java Native Interface (JNI) allows Java applications to invoke routines from libraries written in different languages such as C, C++ and Fortran. This multi-language, multi-threaded, complex compilation and execution model necessitates integrated instrumentation methods. To simplify the creation of performance tools, the Java language designers provided instrumentation capabilities at the virtual machine level.

The Java Virtual Machine Profiler Interface (JVMPi) [132][121] is a portable instrumentation interface that lets a tool to collect detailed profiling information from the virtual machine. This interface loads an in-process profiler agent (implemented as a shared dynamically linked library) and provides a mechanism for the agent to register events that it wishes to track. When these events (such as method entry/exit, thread creation or destruction, mutual exclusion, virtual machine shutdown, object specific events, memory

and garbage collection events) take place, JVMPI notifies the profiler agent. In this manner, an instrumentor can track Java language, application and virtual machine events.

The advantage of this instrumentation level is that tools can access the low-level, language level and virtual machine events that the in-process agent can see. Additionally, this interface provides features such as thread creation and destruction, access to mutual exclusion primitives such as monitors that allows the profiler agent to query and control the state of the virtual machine during execution. The disadvantage of this approach is that it can only see events within the virtual machine. The profiler does not see calls to native system libraries such as message passing libraries, written in C, C++ or Fortran90. This approach may also cause unduly high overheads if it is not possible to selectively disable some of the monitored events.

In the next section, we describe a representational model for inserting instrumentation in the program using a combination of instrumentation interfaces.

3.3 Multi-Level Instrumentation Model

In the previous section, we have seen that as a program source code is transformed to a binary form that executes on the hardware, it presents different instrumentation opportunities to performance tools. Instrumentation interfaces at each level provide detailed information pertinent to that level. Each level has distinct advantages and disadvantages for instrumentation. A performance technologist is thus faced with a set of *constraints* that limit what performance measurements a tool can perform. Typically,

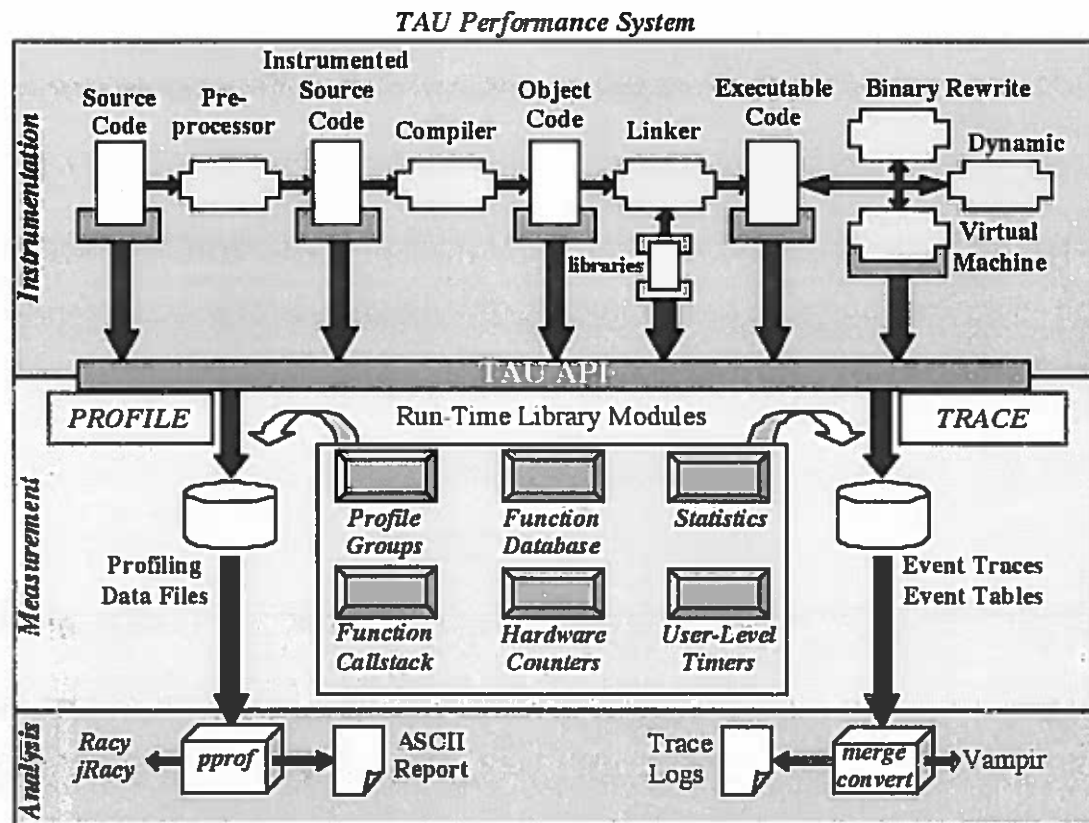


Figure 5. Multi-level instrumentation as implemented in the TAU performance system

monolithic performance tools instrument a program at one instrumentation level and their proponents go to great lengths to justify how that is the most appropriate level for instrumentation. As programming environments evolve towards hybrid modes of operation where multiple languages, compilers, platforms and execution models are used, these monolithic tools are less likely to be applicable. Because they were not designed to operate on hybrid applications, they lack necessary features.

To address these limitations, we propose a model for multi-level instrumentation. In our model, multiple instrumentation APIs are used to instrument a program targeting a common measurement API. This measurement API provides access to the underlying

performance measurement model that specifies the nature of measurements that a tool can perform. An implementation of the measurement model requires access to execution entities which can include such things as the callstack, a database of performance information maintained within each context, node and thread identifiers, low-overhead timers, and hardware performance monitors. To present such diverse performance data in consistent and meaningful ways events triggered from multiple instrumentation interfaces must be integrated using the common measurement API and the performance model embedded in the tools. To achieve this integration, the instrumentation interfaces need to cooperate with each other to provide a unified view of execution: each level and interface makes the information it knows available to other interfaces attempting to uncover missing pieces of information. Semantic entities can be exclusive to one instrumentation interface or they can be partially shared. Each interface must provide mechanisms to control the focus of instrumentation and selectively turn instrumentation on or off. A grouping mechanism can help organize routines into logical partitions that can be addressed collectively for instrumentation (e.g., in turning off all I/O events). Tools also need mechanisms for dealing with missing information that cannot be provided by other interfaces. In some scenarios, needed information may not be available during execution, and might require the post-mortem analysis of performance data. For example, tracking thread identifiers for tracing messages in a mixed mode parallel program is illustrated in the case study (See Section 3.4.5) below.

To demonstrate the efficacy of the multi-level instrumentation model, we built a prototype that targets instrumentation at the source level [113], the preprocessor level [69],

the library level [74], the compiler level, the runtime level [111] and the virtual machine level [110]. It is implemented in the TAU portable profiling and tracing toolkit [74] as shown in Figure 5. At the source level, an instrumentation API allows for manual insertion of instrumentation annotations in the source code. At the preprocessor-level, library routines can be replaced with instrumented ones. A source-to-source translator instruments C++ programs at the preprocessor-level using the Program Database Toolkit (PDT) [69]. Compiler-based instrumentation uses the optimizing lcc compiler, as described in Section 5.3.1. Library-level instrumentation uses an interposition wrapper library for tracking events for the MPI library. DyninstAPI is used for runtime instrumentation and JVMPI is used for virtual machine instrumentation. TAU supports both profiling and tracing performance models and allows third-party tools, such as Vampir [82][94] to be used for visualization of performance data. To demonstrate the use of this prototype, we present a case study.

3.4 Case Study: Integration of Source, Library and JVM Level Instrumentation

Scientific applications written in Java may be implemented using a combination of languages such as Java, C++, C and Fortran. While this defies the pure-Java paradigm, it is often necessary since needed numerical, system, and communication libraries may not be available in Java, or their compiled, native versions offer significant performance advantages [17]. Analyzing such hybrid multi-language programs requires a strategy that leverages instrumentation alternatives and APIs at several levels of compilation, linking,

and execution. To illustrate this point, we consider the profiling and tracing of Java programs that communicate with each other using the Message Passing Interface (MPI). mpiJava [6] is an object-oriented interface to MPI that allows a Java program to access MPI entities such as objects, routines, and constants. While mpiJava relies on the existence of native MPI libraries, its API is implemented as a Java wrapper package that uses C bindings for MPI routines. When a Java application creates an object of the MPI class, mpiJava loads a native dynamic shared object (*libmpijava.so*) in the address space of the Java Virtual Machine (JVM). This Java package is layered atop the native MPI library using the Java Native Interface (JNI) [120]. There is a one-to-one mapping between Java methods and C routines. Applications are invoked using a script file *prunjava* that calls the *mpirun* application for distributing the program to one or more nodes. In contrast, the reference implementation for MPJ [9], the Java Grande Forum's MPI-like message-passing API, will rely heavily on RMI and Jini for finding computational resources, creating slave processes, and handling failures; user-level communication will be implemented efficiently, directly on top of Java sockets, not a native MPI library. The Java execution environment with mpiJava poses several challenges to a performance tool developer. The performance model implemented by the tool must embed the hybrid-execution model of the system consisting of MPI contexts, and Java threads within each of those contexts. To track events through such a system, the tools must be able to expose the thread information to the MPI interface and the MPI context information to the Java interface. Our prototype targets a general computation model initially proposed by the HPC++ consortium [46]. This model consists of shared-memory nodes and a set of

contexts that reside on those nodes, each providing a virtual address space shared by multiple threads of execution. The model is general enough to apply to many high-performance scalable parallel systems and programming paradigms. Performance information, captured at the node/context/thread levels, can be flexibly targeted from a number of parallel software and system execution platforms.

Using this model, different events occur in different software components (e.g., routine transitions, inter-task message communication, thread scheduling, and user-defined events) and performance data must be integrated to highlight the inter-relationship of the software layers. For instance, the event representing a Java thread invoking a message send operation occurs in the JVM, while the actual communication send and receive events take place in compiled native C modules. Ideally, we want the instrumentation inserted in the application, virtual machine, and native language libraries to gather performance data for these events in a uniform and consistent manner. This involves maintaining a common API for performance measurement as well as a common database for multiple sources of performance data within a context of execution.

We apply instrumentation at three levels: the Java virtual machine level, the MPI library level, and the Java source level.

3.4.1 Virtual Machine Level Instrumentation.

Instrumenting a Java program and the JVM poses several difficulties [89] stemming from Java's compilation and execution model in which bytecode generated by

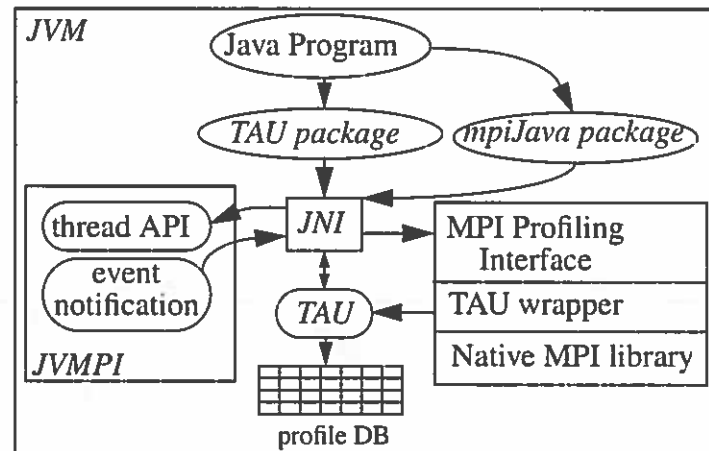


Figure 6. TAU instrumentation for Java and the mpiJava package

the Java compiler is executed by the JVM. Conveniently, Java 2 (JDK1.2+) incorporates the Java Virtual Machine Profiler Interface (JVMPI) [132][121] which provides profiling hooks into the virtual machine and allows an in-process profiler agent to instrument the Java application without any changes to the source code, bytecode, or the executable code of the JVM. JVMPI can notify an agent of a wide range of events including method entry and exit, memory allocation, garbage collection, and thread creation, termination and mutual exclusion events. When the profiler agent is loaded in memory, it registers the events of interest and the address of a callback routine to the virtual machine. When an event takes place, the virtual machine thread that generated the event calls the appropriate callback routine, passing event-specific information. The profiling agent can then use JVMPI to get more detailed information regarding the state of the system and where the event occurred.

Figure 6 shows how JVMPI is used by TAU for performance measurement within a larger system that includes library and source-level instrumentation. Consider a single

context of a distributed parallel mpiJava program. At start-up, the Java program loads the mpiJava and TAU packages and the JVM loads the TAU performance measurement library as a shared object, which acts as a JVMPI profiling agent. A two-way function call interface between the JVM and the profiler agent is established. The JVM notifies the in-process profiling agent of events and it can, in turn, obtain information about and control the behavior of the virtual machine threads using the JVMPI thread primitives (e.g., for mutual exclusion).

When the agent is loaded in the JVM as a shared object, an initialization routine is invoked. It stores the identity of the virtual machine and requests the JVM to notify it when a thread starts or terminates, a class is loaded in memory, a method entry or exit takes place, or the JVM shuts down. When a class is loaded, the agent examines the list of methods in the class and creates an association between the name of the method and its signature, as embedded in the measurement entity, and the method identifier obtained from the TAU Mapping API [124] (further explained in Chapter IV). When a method entry takes place, the agent receives a method identifier from JVMPI, performs the required measurements, and stores the results in the appropriate measurement entity. When a thread is created, it creates a top-level routine that corresponds to the name of the thread, so the lifetime of each user and system level thread can be tracked.

To deal with Java's multi-threaded environment, the instrumentation framework uses a common thread layer for operations such as getting the thread identifier, locking and unlocking the performance database, getting the number of concurrent threads, etc. This thread layer is then used by the multiple instrumentation layers. When a thread is

created, the agent registers it with its thread module and assigns an integer identifier to it. It stores this in a thread-local data structure using the JVMPI thread API described above. It invokes routines from this API to implement mutual exclusion to maintain consistency of performance data. It is important for the profiling agent to use the same thread interface as the virtual machine that executes the multi-threaded Java applications. This allows the agent to lock and unlock performance data in the same way as application level Java threads do with shared global application data. Instrumentation in multiple threads must synchronize effectively to maintain the multi-threaded performance data in a consistent state. The profiling agent maintains a per-thread performance data structure that is updated when a method entry or exit takes place, which results in a low-overhead scalable data structure since it does not require mutual exclusion with other threads. When a thread exits, the agent stores the performance data associated with it to stable storage. When the agent receives a JVM shutdown event, it flushes the performance data for all running threads to the disk.

3.4.2 Library-Level Instrumentation

Instrumentation at the library level is applied using the MPI Profiling Interface to track message communication events. This is accomplished by using a wrapper interposition library layer between the JNI layer and the native MPI library layer as shown in Figure 6. All routines in the MPI are logically grouped together. For profiling, MPI

```

import TAU.*;
import mpi.*;

public class Life {
    static TAU.Profile blocktimer= new TAU.Profile("Life compute local block info",\
    "", "TAU_DEFAULT", TAU.Profile.TAU_DEFAULT);
    static TAU.Profile updatetimer= new TAU.Profile("Life main update loop", "", "\
    TAU_DEFAULT", TAU.Profile.TAU_DEFAULT);

    // .. other static data
    static public void main(String [] args) throws MPIException {
        MPI.Init(args) ;

        Cartcomm p = MPI.COMM_WORLD.Create_cart(dims, periods, false) ;

        /* Compute local `blockSizeX`, `blockBaseX`, `blockSizeY`, `blockBaseY`. */
        blocktimer.Start();
        {
            // Code to compute blockSizeX, blockBaseX, blockSizeY, blockBaseY
        }
        blocktimer.Stop();
        updatetimer.Start();
        for(int iter = 0 ; iter < NITER ; iter++) {
            // Shift this block's upper x edge into next neighbour's lower ghost edge
            p.Sendrecv(block, blockSizeX * sY, 1, edgeXType, dstX[0], 0,
                block, 0, 1, edgeXType, srcX[0], 0) ;

            // other synchronization operations and loops
            dumpBoard() ;
        }
        updatetimer.Stop();

        MPI.Finalize();
    }
}

```

Figure 7. Source-level instrumentation in an mpiJava program

routine entry and exit events are tracked. For event-tracing, inter-process communication events are tracked along with routine entry and exit events.

3.4.3 Source-Level Instrumentation

For other programming languages that our performance system supported (C, C++, Fortran90), standard routine entry and exit instrumentation was supplemented by the

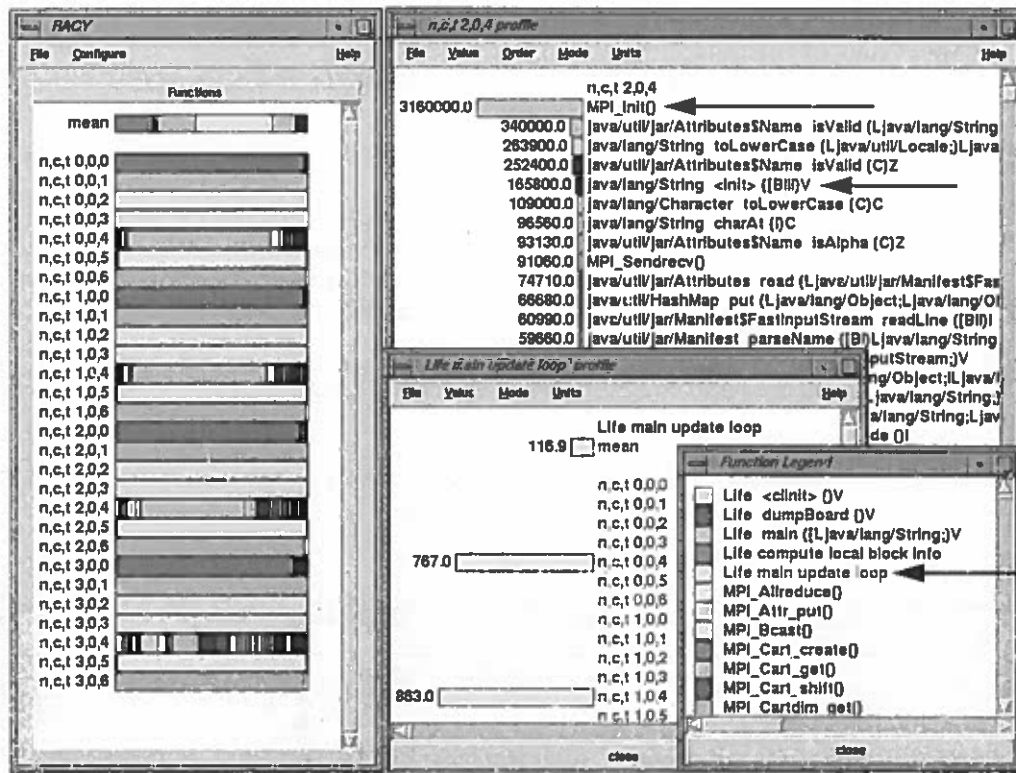


Figure 8. Profile display of an integrated performance view of Life application

ability to specify “user-defined” performance events associated with any code location.

The performance system provided an API to define events, and to start and stop profiling them at specified locations. For Java, we developed a source-level API in the form of a TAU Java package for creating user-level event timers. The user can define event timers of a *TAU.Profile* class and then annotate the source code at desired places to start and stop the timers as shown in Figure 7.

To determine the thread information, the source-level instrumentation needs the cooperation of the virtual machine level instrumentation that sees thread creation, termination and synchronization events. As shown in Figure 6, instrumentation at the

source level cooperates with the instrumentation with the virtual machine level for thread information and with the MPI level for context information. All three instrumentation levels target a common performance database within each context. To demonstrate how program information from these three interfaces can be integrated within performance views, we present performance analysis of an mpiJava benchmark application that simulates the game of Life. This application is run on four processors and Figure 7 shows a portion of its source code that highlights the use of two source-level timers ("*Life compute local block info*," and "*Life main update loop*"). Figure 8 shows a profile display for this application. Note how entities from the source level ("*Life main update loop*") are integrated with MPI library level entities ("*MPI_Init*", "*MPI_Sendrecv*") and Java methods ("*java/lang/String charAt(ICI)*") (shown by the three arrows).

3.4.4 Selective Instrumentation

The performance data from an instrumented Java application contains a significant amount of data about the internal workings of the JVM (e.g., "*java/util/jar/Attributes read*" method in Figure 8). While this may provide a wealth of useful information for the JVM developer, it could inundate the application developer with superfluous details. To avoid this, our multi-level instrumentation is extended to selectively disable the measurement of certain events. Since Java classes are packaged in a hierarchical manner, we allow the user to specify a list of classes to be excluded on the instrumentation command line. For instance, when `-XrunTAU:exclude=java/util,java/lang,sun` is specified

on the command line, all methods of *java/util/**, *java/lang/** and *sun/** classes are excluded from instrumentation [110].

3.4.5 Tracing Hybrid Executions

Inter-thread message communication events in multi-threaded MPI programs pose some challenges for tracing. MPI is unaware of threads (Java threads or otherwise) and communicates solely on the basis of rank information. Each process that participates in synchronization operations has a rank, but all threads within the process share the same rank. We can determine the sender's thread for a send operation and the receiver's thread for a receive operation by querying the underlying thread system through JVMPI. Unfortunately, the sender still won't know the receiver's thread identifier (id) and vice versa. To accurately represent a message on a global timeline, we need to determine the precise node and thread on both sides of the communication. To avoid adding messages to exchange this information at runtime, or supplementing messages with thread ids, we decided to delay matching sends and receives to the post-mortem trace conversion phase. Trace conversion takes place after individual traces from each thread are merged to form a time ordered sequence of events (such as sends, receives, routine transitions, etc.). Each event record has a timestamp, location information (node, thread) as well as event specific data (such as message size, and tags). In our prototype, during trace conversion, each record is examined and converted to the target trace format (such as Vampir, ALOG, SDDF or Dump). When a send is encountered, we search for a corresponding receive by

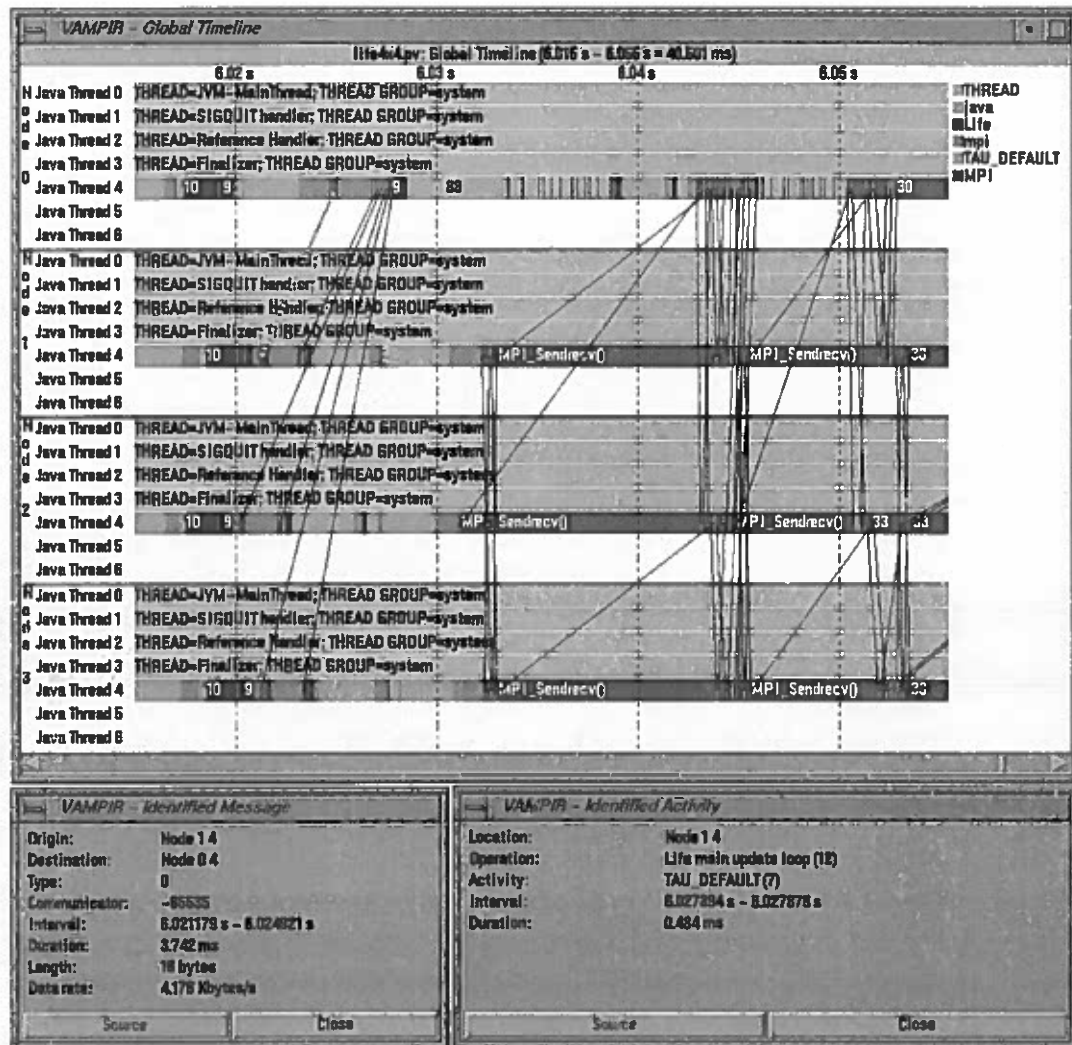


Figure 9. Tracing inter-thread message communication in mpiJava

traversing the remainder of the trace file, matching the receiver's rank, message tag and message length. When a match is found, the receiver's thread id is obtained and a trace record containing the sender and receiver's node, thread ids, message length, and a message tag is generated. The matching works in a similar fashion when we encounter a receive record, except that we traverse the trace file in the opposite direction, looking for the corresponding send event. This technique was used with selective instrumentation

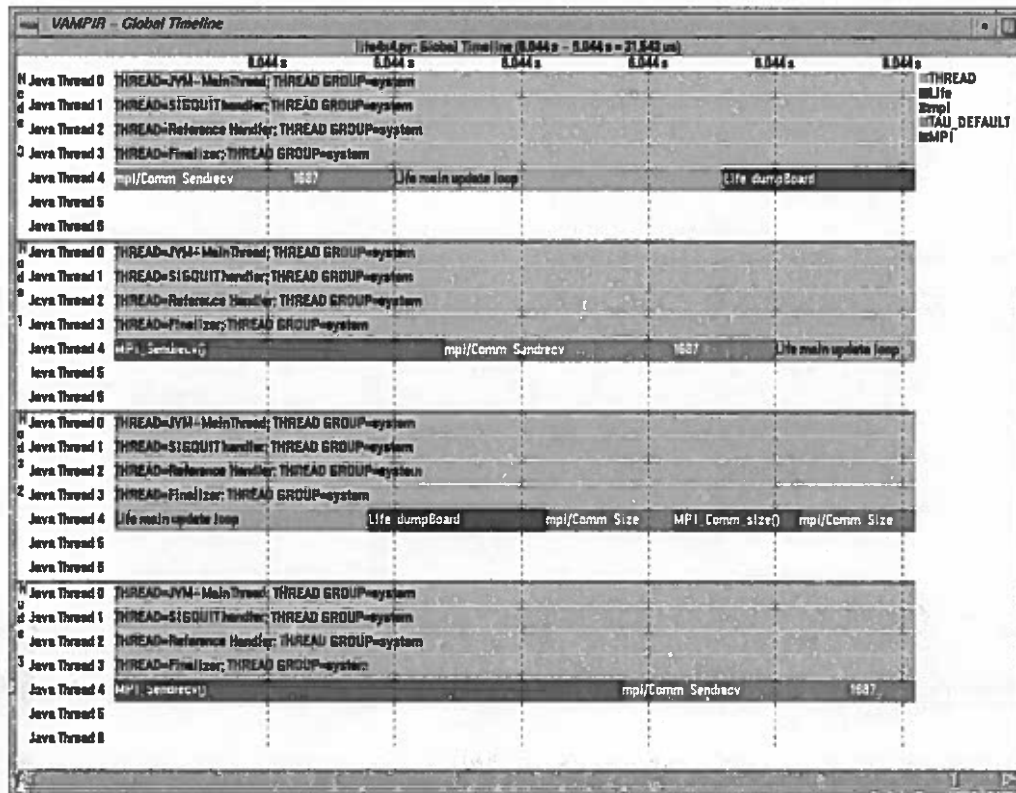


Figure 10. Performance data from source, library and virtual machine levels of a Java application

described earlier, to produce Figures 9 through 11. Figure 9 shows communications as lines from the send event to the corresponding receive. Figure 10 shows a detailed view of the global timeline depicting entities from all three levels in one unified performance view. Figure 11 shows a dynamic call tree display on node 1, thread 4.

The TAU Java package provides the API for these measurements, but utilizes JNI to interface with the profiling library. The library is implemented as a dynamic shared object that is loaded by the JVM or the Java package. It is within the profiling library that the performance measurements are made. However, it captures performance data with respect to nodes and threads of execution. To maintain a common performance data

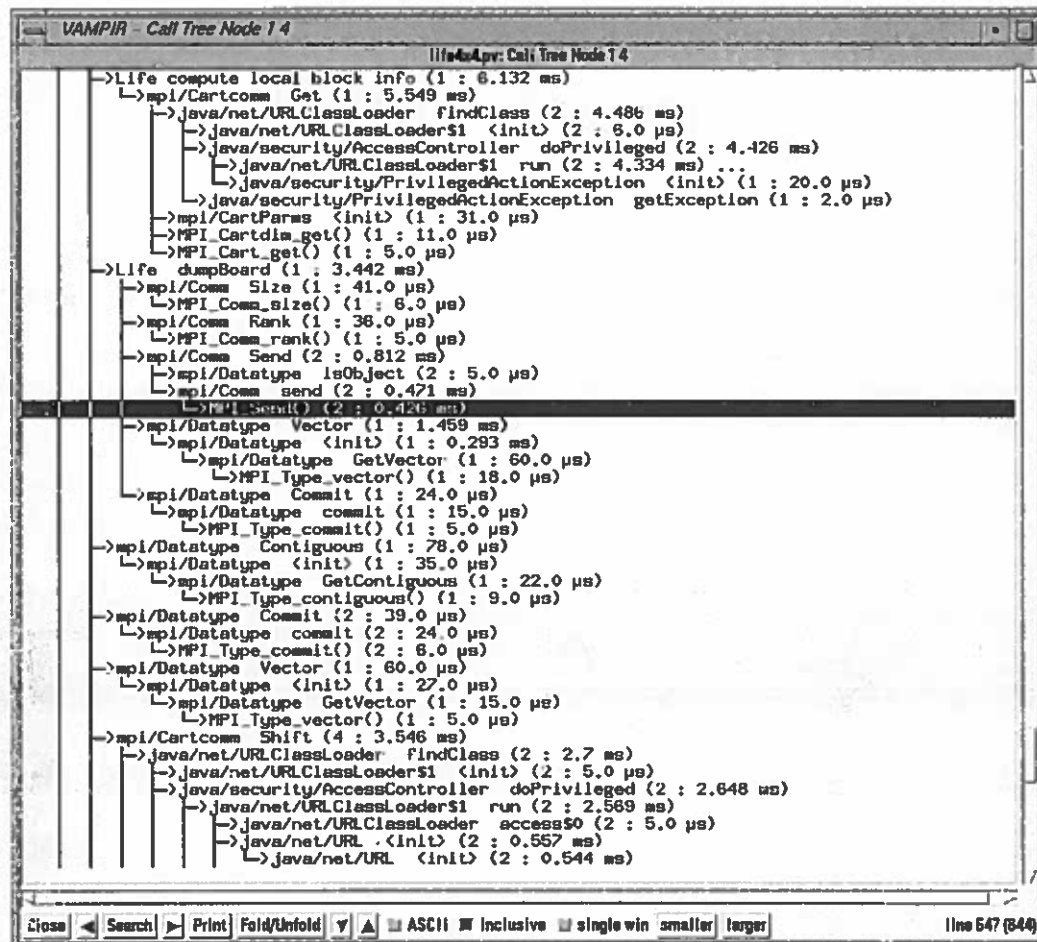


Figure 11. Dynamic call tree display shows Java source, MPI, and virtual machine events

repository in which performance data from multiple “streams” comes together and presents a consistent picture, we need the instrumentation at various levels to cooperate. As shown in Figure 6, the profiling library uses JNI to interface with the JVMPI layer to determine which JVM thread of execution is associated with Java method events and with MPI events. In the same manner, it determines thread information for user-defined events at the source level. For node information, the profiling agent gets process rank from the MPI library.

Thus, instrumentation occurs at the Java source level, at the MPI wrapper library level, and at the virtual machine level.

3.5 Conclusions

In this chapter, we have presented a representational model for multi-level instrumentation to improve the coverage of performance measurement. A prototype of this model is implemented in the TAU portable profiling and tracing toolkit. With the help of a case study, we illustrate how it can be applied to observing the performance of a parallel Java application unifying JVM versus native execution performance measurement, utilizing instrumentation mechanisms at the source, virtual machine, and the library levels.

CHAPTER IV

MAPPING PERFORMANCE DATA

4.1 Introduction

Programming parallel and distributed systems for performance involves a range of runtime issues: overlapping asynchronous inter-node communication with computation using user-level threads, accessing low-latency shared intra-node memory using concurrent threads, exploiting data locality using latency tolerant techniques, partitioning of data among tasks using dynamic load balancing, and generating efficient machine code using sophisticated compilation techniques. To encapsulate such implementation details so that computational scientists can concentrate on science, domain-specific programming environments such as POOMA[101][26], Overture[11][12], and Blitz++ [130], and higher-order parallel languages such as ZPL [21], pC++ [16], CC++ [23], HPC++ [46], and HPF have emerged. They provide higher software layers that support richer programming abstractions for multi-level programming models implemented on multi-layer software architectures.

Understanding the performance characteristics of these “high-level” parallel programs is a challenge because low-level performance data must be mapped back to the

higher-level domain-specific abstractions that the programmer understands. In this chapter, we clarify the need for mapping, present a model for performance data mapping, and show the application of a prototype to complex software scenarios.

4.2 The Need for Mapping

Programming in higher-level parallel languages and domain-specific problem solving environments shields the programmer from low-level implementation details. However, it creates a semantic-gap between user-level abstractions and the entities actually monitored by the performance tool.

Consider a hypothetical example: a scientist writes code to perform some computation on a set of particles distributed across the six faces of a cube as shown in Figure 12. The particles are generated in the routine *GenerateParticles()* and their characteristics depend on which face they reside. *ProcessParticle()* routine takes a particle, as its argument, and performs some computation on it. The main program iterates over the list of all particles and calls the *ProcessParticle()* routine. The scientist thinks about the problem in terms of faces of the cube and the distribution of particles over the cube's surface. However, a typical performance tool reports only on time spent in routines. It might, for example, report that 2% of overall time was spent in *GenerateParticles()* and 98% of the time was spent in *ProcessParticle()*, as shown in Figure 13. This leaves the scientist with no information about the distribution of performance with respect to cube faces: does each face require the same amount of time or not? If the performance tool

described the time spent in *ProcessParticle()* for each face of the cube, as in Figure 14, the scientist might be able to make more informed optimization decisions. We propose a performance mapping model that bridges this gap between a scientist's mental abstractions in the problem domain, and the entities that are tracked by performance tools.

```

Particle* P[MAX]; /* Array of particles */
int GenerateParticles() {
    /* distribute particles over all surfaces of the cube */
    for (int face=0, last=0; face < 6; face++){
        int particles_on_this_face = ... face ; /* particles on this face */
        for (int i=last; i < particles_on_this_face; i++) {
            P[i] = ... f(face); /* properties of each particle are some function f of face */
        }
        last+= particles_on_this_face; /* increment the position of the last particle */
    }
}

int ProcessParticle(Particle *p){
    /* perform some computation on p */
}

int main() {
    GenerateParticles(); /* create a list of particles */
    for (int i = 0; i < N; i++)
        ProcessParticle(P[i]); /* iterates over the list */
}

```

Figure 12. Scientist describes some scientific model

This example is typical: scientific computations often have code segments that execute repeatedly on behalf of different semantic entities (here different cube faces). Such a behavior is commonly found in load balancing or iterative engines where some code region performs “work” and processes data from several abstract entities. Aggregated performance measurements for such code obscure domain-specific detail or

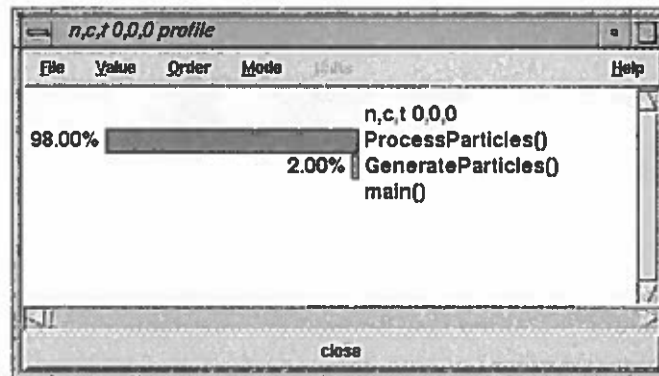


Figure 13. Profile without mapping

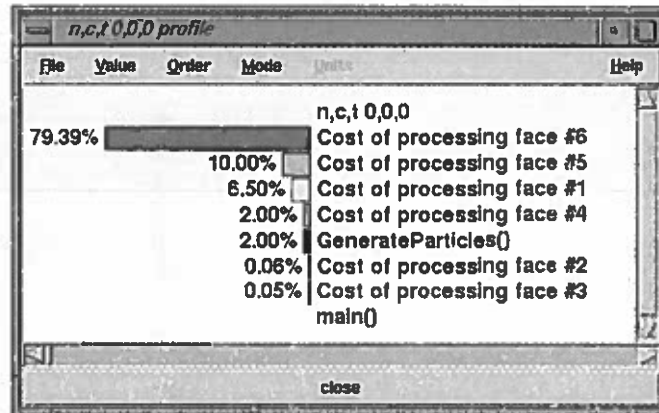


Figure 14. Profile with mapping

“performance knowledge” that may be vital to the scientist. The other extreme in which performance metrics are collected with respect to each instance of the iterative code is also undesirable, leading to an overload of information. In our particle simulation example, we may have to contend with reporting performance metrics for millions of performance entities. The scientist needs to be able to classify performance data for such repetitive code segments in terms of meaningful abstract semantic entities. To do so, we need a mechanism for mapping individual performance measurements at the routine invocation/iteration level to those entities. In terms of repetitive code regions, this amounts to refining

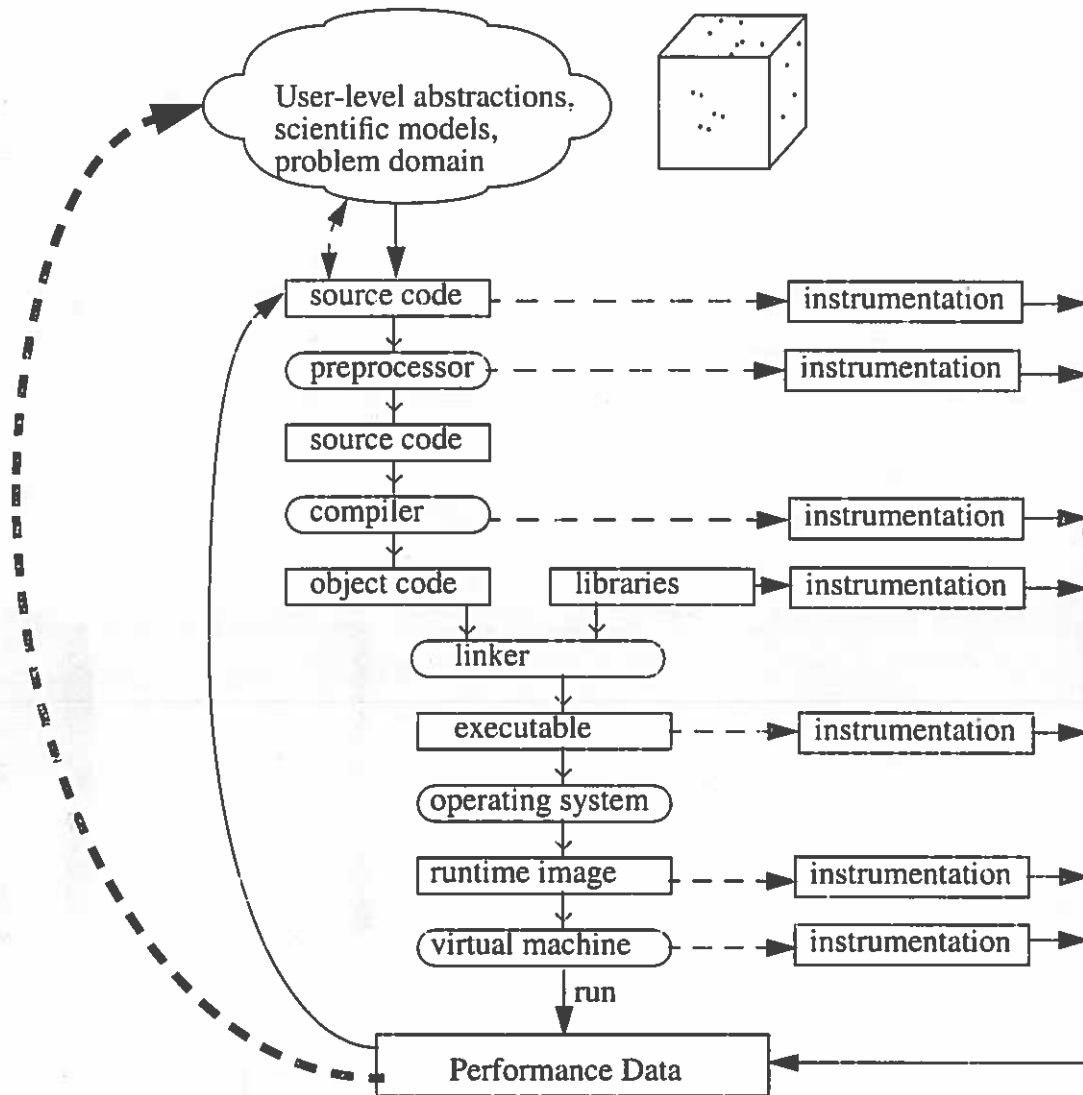


Figure 15. Bridging the semantic gap

upward one-to-many mappings, where a single code segment (where instrumentation is possible at invocation or iteration level) maps to a set of higher level abstractions. To allow a user to do such performance experimentation, a tool must provide flexible mapping abstractions for creating correspondences between different levels of abstraction.

```

Particle P[MAX]; /* Array of particles */
int GenerateParticles() {
    /* distribute particles over all surfaces of the cube */
    for (int face=0, last=0; face < 6; face++) {
        int particles_on_this_face = ... face ; /* particles on this face */
        t = CreateTimer(face); /* information relevant to the domain */
        for (int i=last; i < particles_on_this_face; i++) {
            P[i] = ... f(face); /* properties of each particle are some function of face */
            CreateAssociation(P[i], t); /* Mapping */
        }
        last+= particles_on_this_face; /* increment the position of the last particle */
    }
}

int ProcessParticle(Particle *p){
    /* perform some computation on p */
    t = AccessAssociation(p); /* Get timer associated with particle p */
    Start(t); /* start timer */
    /* perform computation */
    Stop(t); /* stop timer */
}

int main() {
    GenerateParticles(); /* create a list of particles */
    for (int i = 0; i < MAX; i++)
        ProcessParticle(P[i]); /* iterates over the list */
}

```

Figure 16. Mapping instrumentation

Figure 15 shows our approach to mapping performance data to higher-level abstractions by creating an association between some domain-specific information and a performance measurement entity (e.g., a timer that measures inclusive and exclusive elapsed wall-clock time). This association is then accessed and used during the execution of code that requires a more refined instrumentation focus. In our example, it can be done by the code in italics in Figure 16. The performance measurement entity is a simple timer created using the properties of face for each cube face. For instance, the timer for “face 2”

might be semantically identified by the string “cost of processing face #2.” Each face of the cube has one timer. What does “cost of processing face #2” mean in respect to instrumentation? Although this is the performance data we are interested in, it cannot be directly represented in the instrumentation. A more “semantically associated” instrumentation focus is required. When each particle is processed, the appropriate face timer is selected and used to time the computation. Now it is possible to report the time spent processing particles on each face, as was shown in Figure 14. If the scientist decides to modify the geometry, elongating the cube to a cuboid, for example, performance data consistent with this user-level modification can be similarly obtained, helping him/her decide if the effect of such a change on each face of the cube leads to performance improvements.

4.3 Our Approach

Our approach builds upon previous work in this area by Irvin and Miller [49]. They proposed the *noun-verb* (NV) model to aid in mapping performance data from one level of abstraction to another. A noun is any program entity and a verb represents an action performed on a noun [48]. *Sentences* are composed of nouns and verbs. The model considers how sentences at one level of abstraction map to other sentences at a different level of abstraction. An important contribution of Irvin and Miller’s work was that it showed how different mappings can be defined with respect to nouns, verbs, and sentences. Mappings can be *one-to-one*, *one-to-many*, *many-to-one*, or *many-to-many*.

Information about mappings can be determined prior to execution (*static mapping*) or during execution (*dynamic mapping*). Their work further proposed a novel implementation model for dynamic mapping based on the concept of *active sentences* (i.e., sentences that are currently active during execution). Active sentences at the same level of abstraction form a *set of active sentences (SAS)*. Dynamic mapping occurs by identifying which active sentences at one level of abstraction are concurrent with active sentences at another level of abstraction. These concurrently active sentences form the SAS and their entry/exit determines the dynamic mapping. The ParaMap project demonstrated how the NV model and static and dynamic mapping, including the SAS approach, could be implemented in a real parallel language system.

To explain our extension to this work, it is instructive to begin by formalizing the concurrent SAS mapping approach. Let $S = \{S_1, S_2, \dots, S_n\}$ be a set of active sentences at one (lower) level of abstraction and $R = \{R_1, R_2, \dots, R_m\}$ be another set of active sentences at another (higher) level of abstraction. If S and R are concurrent (i.e., their sentences are all active concurrently) (written $S \parallel R$), then S maps to R (written $S \rightarrow R$) by a correspondence between active sentences in S and those in R . That is,

$$S \rightarrow R \text{ when } S \parallel R.$$

Since this is a dynamic mapping, we can categorize the low-to-high (upward) mapping with respect to this correspondence, depending on the cardinality of the sets S and R as shown in Table 2.

TABLE 2: Types of upward mappings $S \rightarrow R$

	$ R = 1$	$ R > 1$
$ S = 1$	one-to-one	one-to-many
$ S > 1$	many-to-one	many-to-many

Consider the case of a one-to-many mapping where S consists of one sentence, $S = \{S_1\}$. How should the performance of S be related to the performance for R ? Two approaches are proposed in Irvin's work:

Aggregation: $T(R) = T(S_1)$ where $S = \{S_1\} \rightarrow R$ (T represents execution time)

Amortization: $T(R_1) = T(R_2) = \dots = T(R_m) = 1/m * T(S_1)$

In aggregation, the cost of executing S_1 ($T(S_1)$) is reported as the cost of executing the active sentences in R as a whole ($T(R)$). This mapping lacks the resolution to characterize performance for each sentence of R individually. In contrast, amortization splits the cost of S_1 across all sentences of the SAS R evenly, under the assumption that each sentence is considered to contribute equally to the performance. This assumption may not be accurate if different sentences affect performance differently. Similar cost analysis applies to the other mapping categories.

There are three problems that were not fully addressed in the ParaMap work. First, the NV model offers a rich conceptual framework for describing program entities at different levels of abstraction. However, the model should also allow for the definition of semantic entities, both as annotations to program entities as well as independent sentences

(in NV terms). This would aid in supporting user-level abstractions. Second, for the one-to-many SAS mapping formalism, it is difficult to refine the resolution and accuracy of mapping performance costs without some means to access additional information (attributes) that can further qualify sentence semantics (“sentence attributes”) and mapping associations (“mapping attributes”). In addition to helping to address the “sentence ordering problem” [49], the notion of attributes could provide a means for representing semantic contexts for mapping control. Finally, the SAS approach did not handle the mapping of asynchronous sentences. These are sentences at different abstraction levels that are logically associated with each other, but are not concurrently active

4.4 Semantic Entities, Attributes, and Associations (SEAA)

As shown in the hypothetical example above (and in the case studies that follow), we are interested in mapping low-level performance data to higher-level abstract entities (i.e., upwards mapping), primarily in cases of one-to-many and many-to-one mapping, and, where asynchronous execution is present. Consider applying SAS to the example where we want to map low-level *particle* execution time to the high-level *face* abstraction. Semantically, each face has associated to it particles at runtime, a many-to-one mapping case. On the other hand, the *ProcessParticle()* routine is applied to all particles and, transitively, to all faces, each a one-to-many mapping case. All mappings are dynamic because they are determined at runtime. However, the *face* entity is a semantic entity that

is not specifically definable in the NV because it is not directly associated with some program construct, except when *particle* properties are being determined. Moreover, even if it could be defined, the notion of a *face* entity being “active” is unclear. If it is considered active in the SAS sense, this occurs only in the setting up of the particles; otherwise, it is inactive. Only by explicitly setting “face” attributes in the *particle* object can the performance mapping inquire about face activity. Otherwise, SAS would have to assume that all faces are active, choosing aggregation or amortization to map performance costs, or that none are, because the faces are not concurrently active (the face and *ProcessParticle()* entities are asynchronously active). If we assume sequential execution, we can imagine how SAS allows us to map performance costs of *Particle()* execution to *particle* entities, since only a one-to-one mapping is present at any time, but parallelizing the particle processing loop begins to stress the approach if there is only one SAS.

We propose a new dynamic mapping technique, *Semantic Entities, Attributes, and Associations (SEAA)*. SEAA embodies three jointly related ideas: 1) entities can be defined based solely on semantics at any level of abstraction, 2) any entity (sentence) can be attributed with semantic information, and 3) mapping associations (entity-to-entity) can be directly represented. The first idea (*semantic entities*) allows entities to be created and used in mapping without having explicit relation to program constructs. The second idea (*semantic attributes*) provides a way to encode semantic information into entities for use in qualifying mapping relationships. The third idea (*semantic associations*) provides the link to dynamic performance cost mapping based on semantic attributes.

Applying SEAA to the example above, we see that an abstract *face* entity is “created” in the form of a “face timer.” Each particle, then, is logically attributed with a face identifier. We say “logically” because in this example an association is directly constructed. With the association in place, the link to the face timer can be accessed at runtime and used to qualify how the performance mapping should be performed (i.e., execution time of *ProcessParticle()* will be accumulated for the properly associated face). With the SEAA concept, problems of SAS concurrency, inaccurate cost accounting, and asynchronous execution can be averted.

An association is a tuple composed of an application data object and a performance measurement entity (PME) (e.g., a timer). We distinguish two types of associations: embedded (internal) and external.

An embedded association extends the data structure (class/struct) of the data object with which it is associated, and stores the address of the PME in it. In Figure 16, *AccessAssociation()* can access a timer stored as a data member in the object using the object address. This is an inexpensive operation, but it requires an ability to extend the class structure to define a new data member, possibly affecting space constraints. Sometimes, it is not possible to create such an association, either due to constraints on class (e.g., due to unavailability of the source code for modification), or the instrumentation interface may not provide such facilities (e.g., it may not be possible to extend classes using dynamic instrumentation, as the code is already compiled). In such cases, an external association may be used.

An external association, as the name suggests, creates an external look-up table (often implemented as a hash table) using the address of each object as its key to look up the timer associated with it. There is a higher runtime cost associated with each lookup, but this scheme does not modify the definition of the class/struct, and hence, is more flexible.

In the remainder of this chapter, we present two case studies showing how embedded and external associations can help describe the application performance in terms of domain-specific abstractions. We compare this mapping approach with traditional schemes.

4.5 Case Study: POOMA II, PETE, and SMARTS

Parallel Object-Oriented Methods and Applications (POOMA) is a C++ framework that provides a high-level interface for developing parallel and portable computational Physics applications. It is a C++ library composed of classes that provide domain-specific abstractions that computational physicists understand. While presenting this easy-to-use high-level interface, POOMA encapsulates details of how data is distributed across multiple processors and how efficient parallel operation, including computational kernels for evaluation of expressions are generated and scheduled. Thus, POOMA shields the scientist from the details of parallel programming, compile-time, and runtime optimizations.

POOMA has a multi-layer software framework that makes analyzing the performance of applications problematic due to the large semantic gap. The high-level data-parallel statements are decomposed into low-level routines that execute on multiple processors. If we measure the performance of these routines, we need some way of mapping the collected data back to constructs that are compatible with the user's understanding of the computation. To better illustrate the gravity of this performance instrumentation and mapping problem, we explain the workings of two POOMA components: PETE and SMARTS in the next two subsections. After that we discuss instrumentation issues, then mapping issues. Finally, we demonstrate our mapping approach and its output.

4.5.1 PETE

The Portable Expression Template Engine (PETE) [41] is a library that generates custom computational kernels for efficient evaluation of expressions of container classes. Based on the expression template technique developed by Todd Veldhuizen and David Vandevoorde [127], POOMA uses PETE for evaluation of array expressions.

An object-oriented language, such as C++, provides abstract-data types, templates, and overloaded-operators. By overloading the meaning of algebraic operators such as +, -, *, and =, the programmer can specify how abstract data types can be added, subtracted, multiplied, and assigned respectively. This allows users to develop high-level container classes, such as multi-dimensional arrays, with an easy-to-understand syntax for creating

arithmetic (array) expressions. Thus, for example, the user can build array expressions such as "A=B+C-2.0*D" or "A(I) = B(I) + C(I) - 2.0*D(I)" where A, B, C, and D are multi-dimensional arrays and I is an interval that specifies a portion of the entire array. Figure 17 shows a simple POOMA program that consists of a set of array statements.

A naive implementation of such an array class may use binary-overloaded operators for implementing the arithmetic operations described above, introducing temporary variables during expression evaluation. The above expression would be evaluated in stages as

```
t1=2.0*D;  
t2=C-t1;  
A=B+t2
```

where t1 and t2 are temporary arrays generated by the compiler. Creating and copying multi-dimensional arrays at each stage of pair-wise evaluation of an array expression is an expensive operation. In contrast, if the same computation is coded in C or Fortran, it would probably resemble Figure 18, where the iteration space spans each dimension of the array and no temporary variables are created. Expression templates allow POOMA array expressions to retain their high-level form, while being transformed by the compiler into C++ code that is comparable to C or Fortran in efficiency [130][103].

Expression templates provide a mechanism for passing expressions as arguments to functions [127]. In C++, an expression is parsed by the compiler and its form, or structure, is stored as a nested template argument. Thus, the form of an expression is

```

xterm
#include "Pooma/Arrays.h"
#include <iostream.h>

// The size of each side of the domain.
const int N = 3*1024;

int
main(
  int          argc,          // argument count
  char *      argv[]        // argument list
){
  // Initialize Pooma.
  Pooma::initialize(argc, argv);

  // The array we'll be solving for
  Array<2> A(N, N), B(N,N), C(N,N), D(N,N), E(N,N);

  // Must block since we're doing some scalar code (see tutorial 4).
  Pooma::blockAndEvaluate();

  A = 1.0;
  B = 2.0;
  C = 3.0;
  D = 4.0;
  E = 5.0;

  A = B + C + D;
  C = E - A + 2.0 * B;
  D = A + C;
  C = D + A - B;
  A = 2.0 * D + E;
  E = 1.5 * B - A;

  Pooma::blockAndEvaluate();

  cout << "D(1,1) = " << D(1,1) << endl;
  cout << "D(9,9) = " << D(9,9) << endl;

  // Clean up Pooma and report success.
  Pooma::finalize();
  return 0;
}

```

Figure 17. Parallel multi-dimensional array operations in POOMA.

embedded in its type. Figure 19 shows how an expression, such as “ $B+C-2.0*D$ ” can be represented as a parse tree and written in a prefix notation, which resembles the form of the equivalent template expression. During the template instantiation process, a compiler generates specialized code for satisfying instantiation requirements. Expression templates scheme takes advantage of this property of a C++ compiler that resembles a partial evaluator [127][128].

Thus, the C++ compiler generates application-specific computation kernels during the template instantiation and specialization phase, an approach known as *generative programming* [27]. Expression objects such as those shown in Figure 19, contain

```

const int N = 1024;
double A[N][N], B[N][N], C[N][N], D[N][N];
int i, j;
/* initialization ...*/
/* computation */
for (i=0; i < N; i++)
  for (j=0; j < N; j++)
    A[i][j] = B[i][j] + C[i][j] - 2.0 * D[i][j];

/* Instead of simply
   A = B + C - 2.0 * D;
   if coded in POOMA */

```

Figure 18. Equivalent evaluation of array expressions in C

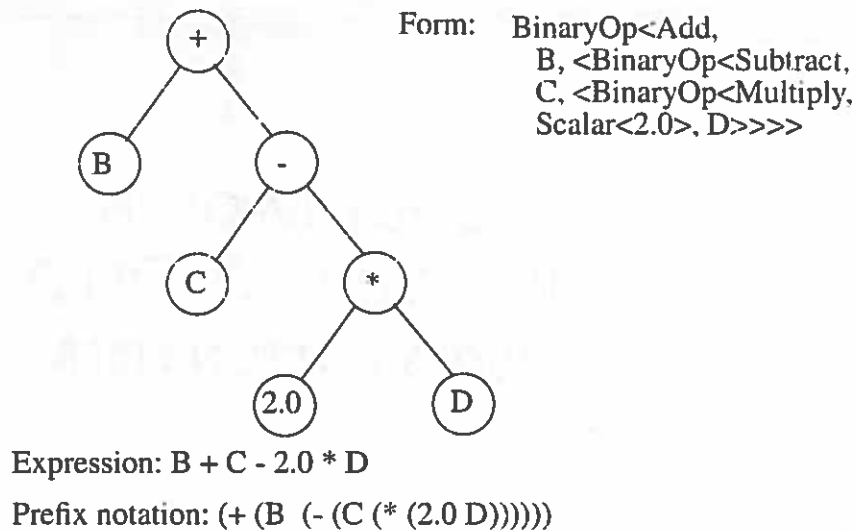


Figure 19. Parse tree for an expression (form taken from [41])

references to all scalars and arrays that are present in the expression. The structure of the expression is embedded in the type of the object. A compiler that is able to inline functions aggressively [103] can generate efficient inlined code for the assignment of the evaluator object to the array on the left hand side of an expression. For the above expression, Figure

20 shows code that can be generated by a good C++ compiler. In this manner, PETE can generate efficient C++ code that does not copy or create temporary arrays in the evaluation of an array expression.

```

/* For a one dimensional case, A=B+C-2.0*D would look like: */
vector<double>::iterator itA = A.begin();
vector<double>::iterator itB = B.begin();
vector<double>::iterator itC = C.begin();
vector<double>::iterator itD = D.begin();

while (itA != A.end())
{ /* A = B + C - 2.0 * D would be transformed as : */
  *itA = *itB + *itC - 2.0 * (*itD);
  *itA++; *itB++; *itC++; *itD++;
}

```

Figure 20. PETE can generate code that is comparable in performance to C

Given PETE's transformations, there is clearly a need to figure out a way to track the transformations for performance mapping purposes. Unfortunately, POOMA takes the problem one step further. Instead of executing each array statement in its entirety before the next, POOMA delays the evaluation of array expressions, and groups work packets (*iterates*), from multiple array statements (loop jamming). These packets are then scheduled on available threads of execution.

4.5.2 SMARTS

SMARTS [126] is a run-time system that exploits loop-level parallelism in data-parallel statements through runtime-data dependence analysis and execution on shared

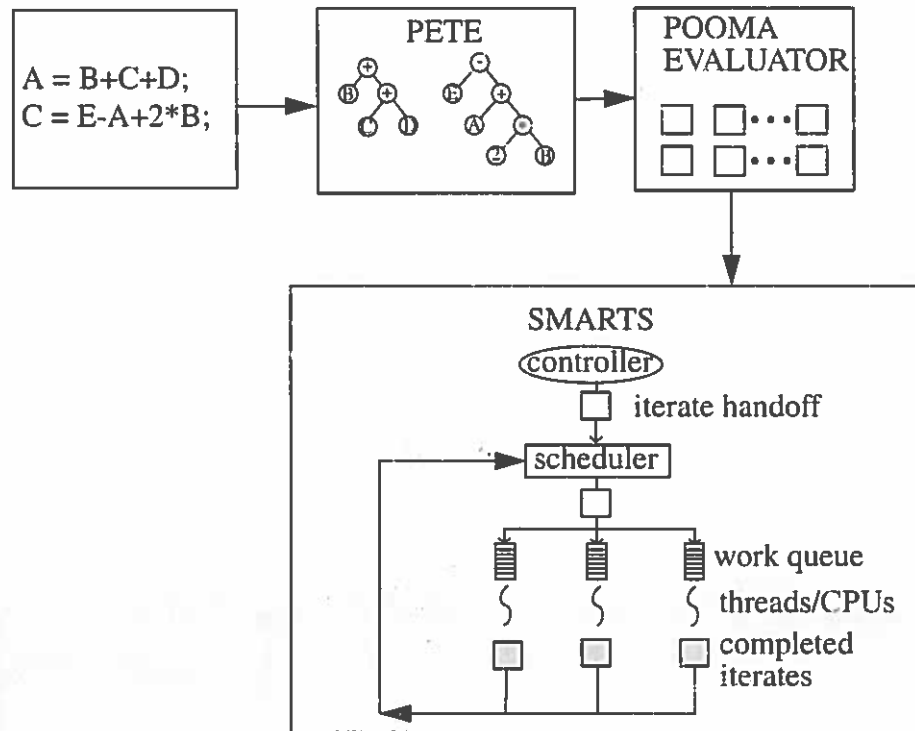


Figure 21. Compile-time (PETE) and run-time (SMARTS) optimizations employed by POOMA

memory multiprocessors with hierarchical memories. It schedules independent loop iterations of data-parallel (POOMA array) statements to maximize cache re-use.

POOMA statements are decomposed into iterates, or work packets, as shown in Figure 21, after compile-time expression template optimizations are applied in the PETE layer. Iterates are scheduled for asynchronous execution by the SMARTS controller, onto one or more user-level SMARTS threads. Evaluating interleaved work packets that originate from multiple statements is often referred to as vertical execution [125]. This is in contrast to horizontal execution where all work packets from a single statement must be processed in a lock step fashion with, perhaps, a barrier synchronization operation

between statements. The “vertical” direction here refers to the direction in which chunks of the set of high-level data-parallel statements in a program are roughly traversed (evaluated). The actual order of statement evaluation depends on how the dependence graph, representing the data dependencies among the iterates, is partitioned and evaluated. SMARTS applies multiple operations corresponding to different statements to the same data before it leaves the cache. Locality-aware compiler optimizations such as strip-mining, tiling, loop fusion, and loop interchange [93][138] achieve the same effect. However, these may not be successfully applied when inter-statement data dependencies are complex. SMARTS operates at a library level, instead of the compiler level, and can see beyond the compiler optimizations. Several scheduling algorithms are incorporated in SMARTS and a comparison of their relative parallel speedups is presented in [126]. The choice of a scheduler determines the scheduling policy, its overhead, and the extent of cache re-use. Thus, SMARTS tries to improve temporal locality and increase parallelism of data-parallel statements.

4.5.3 Instrumentation Challenges Created by PETE and SMARTS

Both PETE and SMARTS present challenging performance instrumentation problems. To highlight these, consider the execution of the high-level POOMA array statements shown in Figure 17. Using an off-the-shelf vendor supplied performance tool, we obtain a sequential profile as shown in Figure 22, which gives the percentage of cumulative and exclusive time spent in each routine. Templates are translated into their

```

emacs@pyros.cs.uoregon.edu
Buffers Files Tools Edit Search Mule Help
Function list, in descending order by exclusive ideal time
-----
[index]  excl.secs  excl.%  cum.%  cycles  instructions  calls  function (dso\
: file, line)
-----
[1]      0.731    23.7%   23.7%   141895935   118364520     5  run_161Expres\
sionKernel_tm_136_59Array_tm_46_XCiL_1_2d34BrickView_tm_17_XCiL_1_2XCbL_1_180pAssign46\
ConstArray_tm_28_XCiL_1_2d16ConstantFunction15InlineKernelTagFv_v (SimpleJacobi: Expressio\
nKernel.h, 220; compiled in UCCicAAAa04JHF.int.c)
[2]      0.487    15.8%   39.5%   94497844    113418314     1  run_484Expres\
sionKernel_tm_459_59Array_tm_46_XCiL_1_2d34BrickView_tm_17_XCiL_1_2XCbL_1_180pAssign36\
8ConstArray_tm_349_XCiL_1_2d336ExpressionTag_tm_314_310BinaryNode_tm_291_50pAdd164Bina\
ryNode_tm_145_100pSubtract64ConstArray_tm_46_XCiL_1_2d34BrickView_tm_17_XCiL_1_2XCbL_1\
_164ConstArray_tm_46_XCiL_1_2d34BrickView_tm_17_XCiL_1_2XCbL_1_1114BinaryNode_tm_96_10\
0pMultiply15Scalar_tm_2_d64ConstArray_tm_46_XCiL_1_2d34BrickView_tm_17_XCiL_1_2XCbL_1\
115InlineKernelTagFv_v (SimpleJacobi: ExpressionKernel.h, 220; compiled in UCCicAAAa04JHF.in\
t.c)
[3]      0.390    12.6%   52.1%   75629619    80403528     1  run_427Expres\
sionKernel_tm_402_59Array_tm_46_XCiL_1_2d34BrickView_tm_17_XCiL_1_2XCbL_1_180pAssign31\
1ConstArray_tm_292_XCiL_1_2d279ExpressionTag_tm_257_253BinaryNode_tm_234_50pAdd158Bina\
ryNode_tm_139_50pAdd64ConstArray_tm_46_XCiL_1_2d34BrickView_tm_17_XCiL_1_2XCbL_1_164Co\
nstArray_tm_46_XCiL_1_2d34BrickView_tm_17_XCiL_1_2XCbL_1_164ConstArray_tm_46_XCiL_1_2d\
34BrickView_tm_17_XCiL_1_2XCbL_1_115InlineKernelTagFv_v (SimpleJacobi: ExpressionKernel.h,\
220; compiled in UCCicAAAa04JHF.int.c)
[4]      0.390    12.6%   64.7%   75623475    80400456     1  run_433Expres\
sionKernel_tm_408_59Array_tm_46_XCiL_1_2d34BrickView_tm_17_XCiL_1_2XCbL_1_180pAssign31\
7ConstArray_tm_298_XCiL_1_2d285ExpressionTag_tm_263_259BinaryNode_tm_240_100pSubtract1\
58BinaryNode_tm_139_50pAdd64ConstArray_tm_46_XCiL_1_2d34BrickView_tm_17_XCiL_1_2XCbL_1\
_164ConstArray_tm_46_XCiL_1_2d34BrickView_tm_17_XCiL_1_2XCbL_1_164ConstArray_tm_46_XCi\
L_1_2d34BrickView_tm_17_XCiL_1_2XCbL_1_115InlineKernelTagFv_v (SimpleJacobi: ExpressionKer\
nel.h, 220; compiled in UCCicAAAa04JHF.int.c)
[5]      0.390    12.6%   77.4%   75608115    66213960     1  run_383Expres\
sionKernel_tm_358_59Array_tm_46_XCiL_1_2d34BrickView_tm_17_XCiL_1_2XCbL_1_180pAssign26\
7ConstArray_tm_248_XCiL_1_2d235ExpressionTag_tm_213_209BinaryNode_tm_190_50pAdd114Bina\
ryNode_tm_96_100pMultiply15Scalar_tm_2_d64ConstArray_tm_46_XCiL_1_2d34BrickView_tm_1\
7_XCiL_1_2XCbL_1_164ConstArray_tm_45_XCiL_1_2d34BrickView_tm_17_XCiL_1_2XCbL_1_115Inlin\
eKernelTagFv_v (SimpleJacobi: ExpressionKernel.h, 220; compiled in UCCicAAAa04JHF.int.c)
[6]      0.390    12.6%   90.0%   75601971    66210888     1  run_389Expres\
sionKernel_tm_364_59Array_tm_46_XCiL_1_2d34BrickView_tm_17_XCiL_1_2XCbL_1_180pAssign27\
3ConstArray_tm_254_XCiL_1_2d241ExpressionTag_tm_219_215BinaryNode_tm_196_100pSubtract1\
14BinaryNode_tm_96_100pMultiply15Scalar_tm_2_d64ConstArray_tm_46_XCiL_1_2d34BrickView_\
_tm_17_XCiL_1_2XCbL_1_164ConstArray_tm_46_XCiL_1_2d34BrickView_tm_17_XCiL_1_2XCbL_1_115\
InlineKernelTagFv_v (SimpleJacobi: ExpressionKernel.h, 220; compiled in UCCicAAAa04JHF.int.c)
[7]      0.292     9.5%   99.5%   56727603    56773704     1  run_332Expres\
sionKernel_tm_307_59Array_tm_46_XCiL_1_2d34BrickView_tm_17_XCiL_1_2XCbL_1_180pAssign21\
6ConstArray_tm_197_XCiL_1_2d184ExpressionTag_tm_162_158BinaryNode_tm_139_50pAdd64Const\
Array_tm_46_XCiL_1_2d34BrickView_tm_17_XCiL_1_2XCbL_1_164ConstArray_tm_46_XCiL_1_2d34B\
rickView_tm_17_XCiL_1_2XCbL_1_115InlineKernelTagFv_v (SimpleJacobi: ExpressionKernel.h, 22\
0; compiled in UCCicAAAa04JHF.int.c)
-----
-- prof_output (Fundamental)--L26-- 1%-----
Minibuffer window is not active

```

Figure 22. Profile generated by a vendor-supplied performance tool.

instantiated entities in forms not known to the user. None of the entities in the output relate directly to code that the user writes. On closer inspection, we find that the routines profiled are templates, but the names are mangled by the C++ compiler. There are two ways to deal with this: employ a demangling tool that translates between each demangled and mangled name, or to instrument it so that the application generates de-mangled template names. As

our current focus is not on the demangling problem, but rather on the mapping issue, we use the latter approach and apply routine-based source-level template instrumentation.

Each template instantiation can be uniquely described using runtime-type information (RTTI) of objects [113] using C++ language features [118] that are independent of compiler implementations and system specific constraints. With this support, we can construct the name of an instantiated template based on the method name, class name, and a string that represents the types of its arguments and its return type. Once we can construct the name, we can define a semantic entity that represents the instantiated template class. Performing source-level instrumentation of templates is now possible and results in a profile that shows the template names as illustrated in Figure 23. This figure shows the complete template name and includes the instantiations in terms of types at the language level, instead of showing the names in the internal compiler name mangling format, that the user does not understand.

While the observed program entities are presented at a higher language level, it may still not be at a high enough level of abstraction. Although, name mangling problems are ameliorated, only the structure of the expression template is given, which can be fairly difficult to interpret, even for someone who understands the inner workings of PETE. For the performance data to be useful, it must be converted into a form that the user can understand. Ideally, this would be in terms of actual source-level data-parallel POOMA statements. Suppose the library that implements the expression templates describes the form of the expression template to the tool environment. For instance, the form can be a string like "Array=Array+scalar*Array." A semantic entity can again be defined, but in

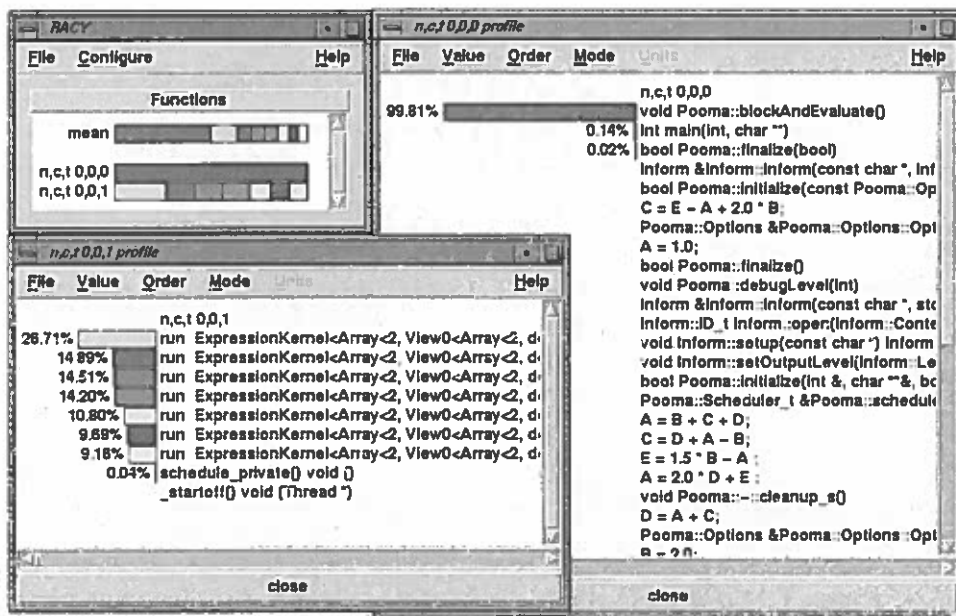


Figure 24. Synchronous timers cannot track the execution of POOMA array statements.

4.5.4 Mapping Challenges in POOMA

Standard routine-based instrumentation of the parallel POOMA application (optimized using PETE and SMARTS) aggregates the time spent in all expression templates that have the form “Array=scalar,” as shown in Figure 23. To get a statement-level profile of the POOMA application, one might try to use conventional timers. Here, the name of the timer could be composed of the statement text, and the cost of each statement could be measured by surrounding it with start and stop timing routines. However, using “synchronous” timers for tracking the execution of POOMA statements leads to a profile, as shown in Figure 24. It shows that there are two threads of execution; on thread 0 (the controller thread), 99.81% of time is spent in *Pooma::blockAndEvaluate()*

```

A = 1.0;
B = 2.0;
A = B + C + D;
C = E - A + 2.0 * B;
...
template<class LHS, class RHS, class Op,
        class EvalTag>
void ExpressionKernel<LHS, RHS, Op,
                    EvalTag>::run()
{ /* iterate execution */
}

```

Figure 25. One-to-many mappings in POOMA

routine, and negligible time (< 1%) is spent in the execution of actual array statements. On thread 1 we see a profile similar to Figure 22 that consists of template expressions.

Clearly, there is something happening in the execution that the user does not expect. This again, highlights the semantic-gap between the user's perceptions and the result of transformations that tools cannot track effectively. The reason for this gap here is due to the asynchronous execution of iterates in SMARTS.

As discussed above, POOMA breaks up each array statement into iterates. All iterates are executed by the *run()* method of the *ExpressionKernel* class template, as shown in Figure 25. To accurately measure and correlate the cost of executing each array statement, we need to measure the cost of creating the iterates (the synchronous component of executing a statement on thread 0), perform measurements in the *ExpressionKernel::run()* method, and map these costs to the appropriate POOMA statement. To make matters worse, iterate execution takes place on threads other than thread 0 (controller thread), as described in Figure 21. Furthermore, iterates are executed out-of-order and asynchronously, to maximize cache re-use.

4.5.5 Our Mapping Approach

Beyond the use above of semantic entities to track template instantiations and capture statement identification, the problem of iteration mapping and asynchronous execution requires more sophisticated and aggressive mapping support. Using our association based approach to mapping performance data, we can create a unique association between a (low-level) *ExpressionKernel* object and its corresponding (high-level) POOMA statement. We do this using semantic attributes and semantic association. As before, for each high-level POOMA statement (e.g., $A=B+C+D$;) we create a timer (a semantic entity) with the statement string (e.g., "A=B+C+D;") as the timer name (semantic attribute). During the creation of each *ExpressionKernel* object that represents an iterate, the timer address is stored in an STL map [118] with a key (a hash attribute). When the statement executes, the constructor of the *ExpressionKernel* object is invoked and the map (which acts as a hash table) is searched with the same key to reveal the timer associated with the statement (i.e., a dynamic semantic association). The *ExpressionKernel* class is extended to include a data member that holds the address of the timer. In this manner, each iterate is assigned to the statement-level timer through a process that encodes the dynamic semantic association into the iterate object. When the *run()* method is invoked asynchronously during iterate evaluation, the "embedded association" (i.e., the per-object statement timer) is accessed and timing measurements in this method are performed with the statement-level timer. Hence, low-level cost of executing the *run()* method for each statement iterate can be precisely measured and

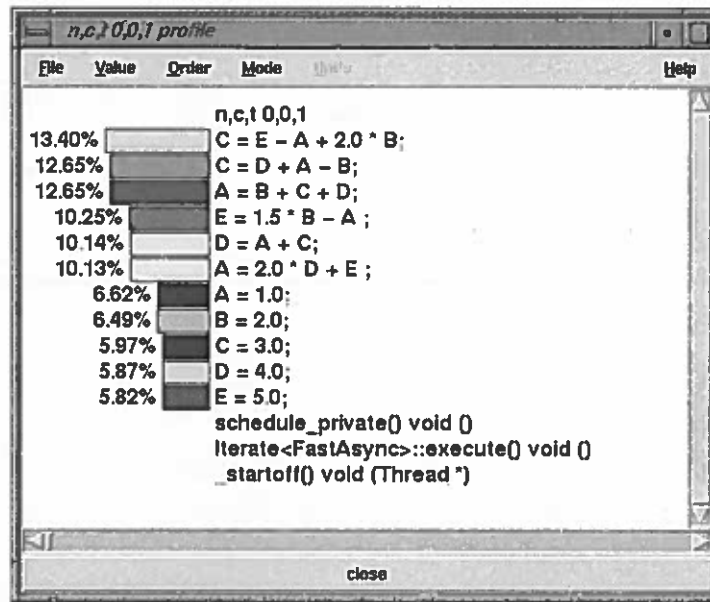


Figure 26. Mapping costs of individual POOMA statements

accumulated for the statement. The performance data can then be shown in a more meaningful manner, as shown in Figure 26.

Moreover, no limitations are imposed on the choice of performance measurement. We could just as easily generate event traces, to highlight the temporal aspect of iterate scheduling. Figure 27 shows the scheduling of iterates on threads 1, and 2, while the *Pooma::blockAndEvaluate()* (controller code) awaits their completion on thread 0 for a thousand iterations of the array statements.

Following the SEAA approach, we can partition the performance data from one-layer of abstraction, based on application data (iterate objects) and correlate it to another, accurately accounting for the cost of individual statement execution, even in the presence of asynchronous execution. Mapping performance data based on embedded associations

provides a better view of the performance data in POOMA, one that corresponds to source-level constructs.

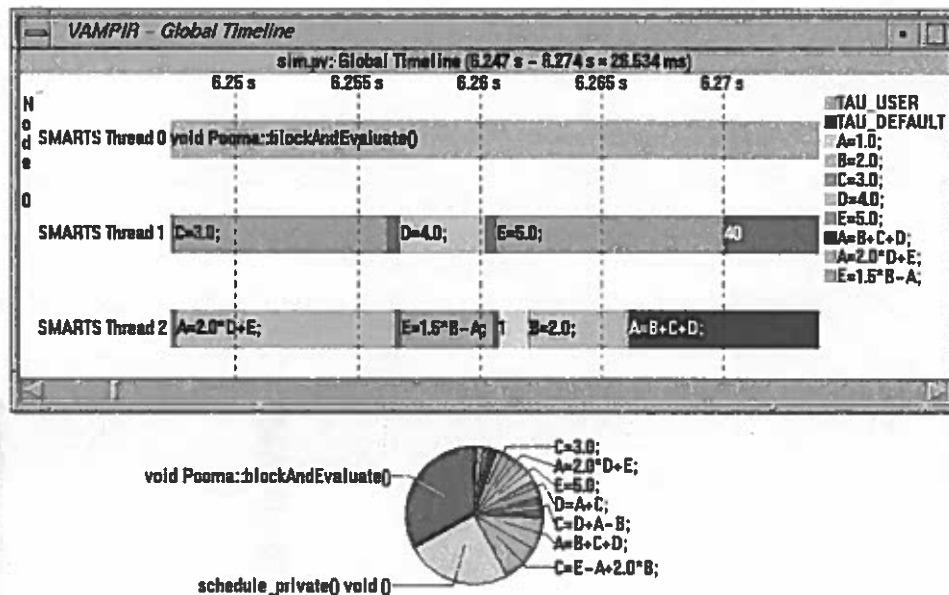


Figure 27. Event traces of mapped iterates show the contribution of array statements

4.6 Case Study: Uintah

Admittedly, POOMA represents a complex programming system, but that does not mean that the nature of mapping problems it presents are uncommon. Uintah [35] is a component-based framework for modeling and simulation of the interactions between hydrocarbon fires and high-energy explosive materials and propellants. Similar to POOMA's programming objective, Uintah encapsulates the complexities of parallelism, but uses the DOE Common Component Architecture (CCA) [57] as the programming approach and the SCIRun framework [54] for interactive steering and visualization.

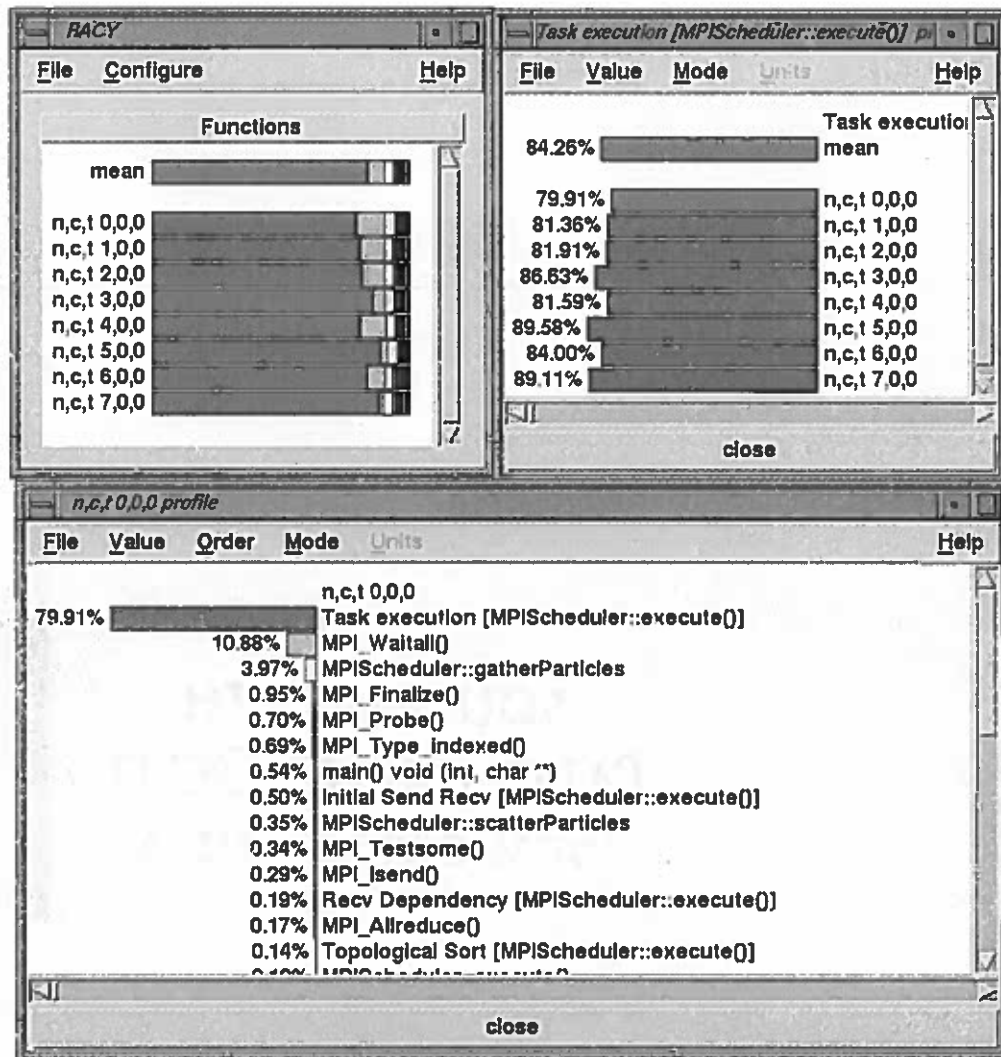


Figure 28. Task execution needs to be explored further in Uintah

Uintah is a domain-specific interactive scientific Problem Solving Environment (PSE) that targets tera-scale high performance computing. In [35], the authors describe a typical simulation using the PSE: a cylinder containing combustible, high-energy materials is suspended above a flame. As the simulation progresses, the high-energy material explodes, the cylinder deforms and cracks as the pressure builds up, and the material finally detonates. To simulate such a system, a computational fluid dynamics (CFD)

time	msec	total msec	#call	#subrs	usec/call	name
89.3	1:29.712	1:35.091	225	28	422625	MPI Waitall()
6.8	7.202	7.202	49	0	146993	MPI Scheduler::gatherParticles
4.7	3.735	5.026	14	294	359036	MPI Scheduler::gatherParticles
1.2	1.287	1.287	99	0	13141	MPI Finalize()
1.0	1.012	1.012	1	0	1012782	MPI Allreduce()
0.7	719	719	99	0	7268	MPI Type Indexed()
0.5	564	564	598.5	0	943	MPI Type Indexed()
100.0	472	1:46.469	1	19	106469389	main() void(int, char **)
0.3	350	352	14	98	25178	MPI Scheduler::gatherParticles
1.2	297	1.303	15	2688.5	86895	MPI Scheduler::gatherParticles
79.5	195	1:44.924	15	1189	6994941	MPI Scheduler::execute()
0.1	151	151	498.75	0	303	MPI Send()
0.1	149	149	15	0	9994	MPI Scheduler::gatherParticles
0.1	133	133	131.25	0	1020	MPI Recv()
6.8	106	7.197	211	2165	34111	MPI Scheduler::gatherParticles
0.1	102	102	85.75	0	1190	MPI Scheduler::gatherParticles
0.1	86	86	498.75	0	173	MPI Recv()
0.2	64	248	211	2455.25	1180	Send Dependency [MPI Scheduler::execute()]
0.1	60	60	1	0	6410	MPI Init()
0.0	25	49	73.5	73.5	679	Test Some [MPI Scheduler::execute()]
0.0	9	11	15	30	798	MPI Scheduler::gatherParticles
0.0	6	6	1197	0	6	MPI Type hvector()
0.7	4	723	99	99	7312	MPI Scheduler::gatherParticles
0.0	3	3	1795.5	0	2	MPI Scheduler::gatherParticles
0.0	2	2	1030.75	0	2	MPI Pack size()
0.0	2	2	1000.5	0	2	MPI Scheduler::gatherParticles
0.0	1	1	15	0	126	MPI Scheduler::gatherParticles
0.0	1	1	498.75	0	3	MPI Type size()
0.0	1	1	98	0	16	MPI Scheduler::gatherParticles
0.0	1	1	354	0	4	Task Graph Output [MPI Scheduler::execute()]
0.0	1	1	33.25	0	34	MPI Send()
0.0	0.284	0.284	30	0	9	MPI Buffer detach()
0.0	0.244	0.244	98	0	2	MPI Get count()
0.0	0.228	0.228	3	0	76	MPI Type vector()
0.0	0.0778	0.0778	15	0	5	MPI Buffer attach()
0.0	0.0565	0.0565	1	0	57	MPI Comm size()
0.0	0.019	0.019	1	0	2	MPI Scheduler::gatherParticles

Figure 29. Uintah node profile shows that task execution is a computationally intensive activity.

component, simulating hydrocarbon combustion, is coupled with a material point method (MPM) component, to study the mechanics of solid deformation and energy transport within the cylinder.

To evaluate the performance of a Uintah application, we selectively instrument at the source level and the message passing library level. This two-level instrumentation, is explained in the previous chapter. It enables us to see the details of message communication as well as application level routines. Figure 28 shows a high-level profile of the execution of different tasks within the parallel scheduler. We see that *Task execution* takes up a significant chunk of the overall execution time. It takes up 79.91% of exclusive time on node 0 and Figure 29 shows that it takes 89.3% of inclusive time averaged over all

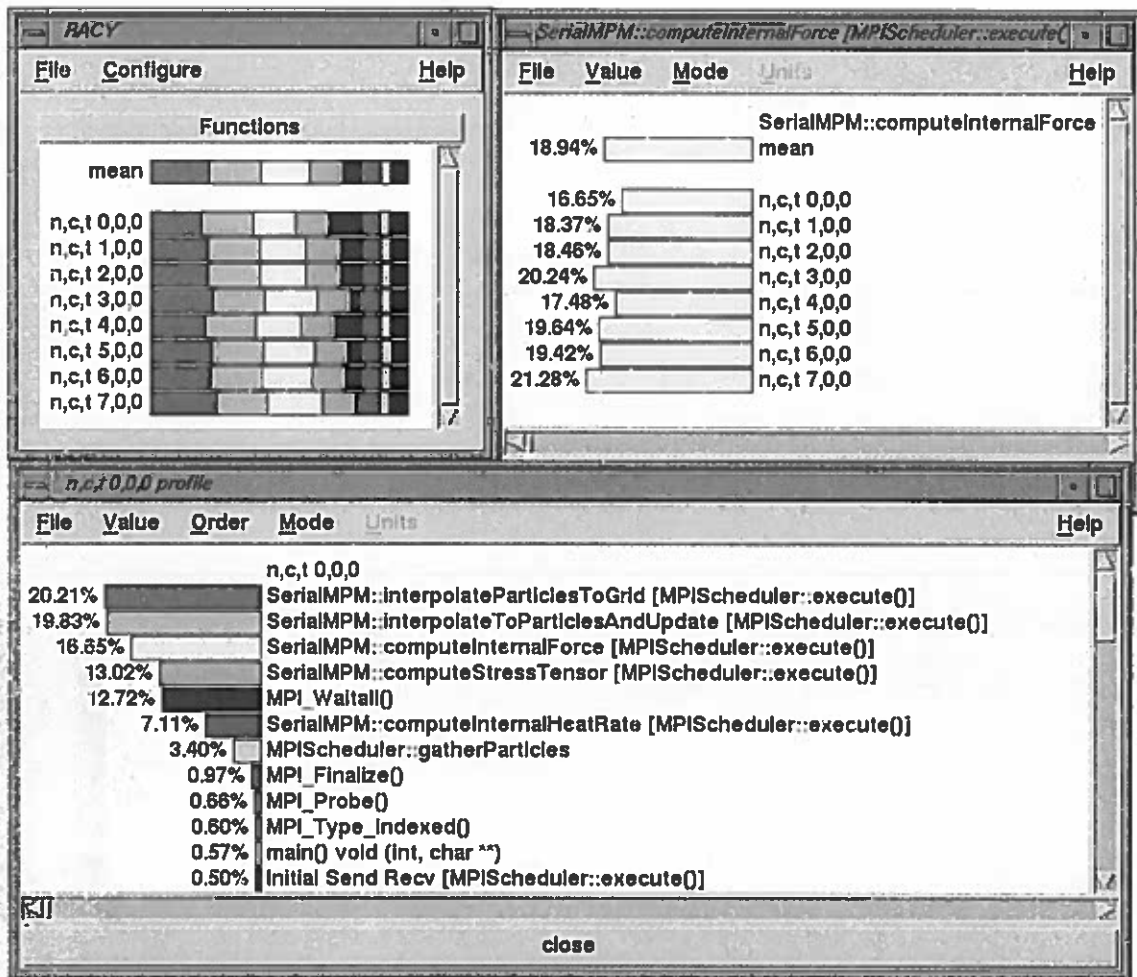


Figure 30. Mapping reveals the relative contribution of different tasks

nodes. While more detailed instrumentation can show us each instance of task execution, it does not highlight the nature of tasks that execute (i.e., the task semantics: “what does the task do?”). The computation on the individual particles generates work packets. Each work packet belongs to a task, and each task does something that the computational scientist understands (such as interpolating particles to a grid in the serial multi-point method). Tasks can be given a name (e.g., *SerialMPM::interpolateParticlesToGrid*) and

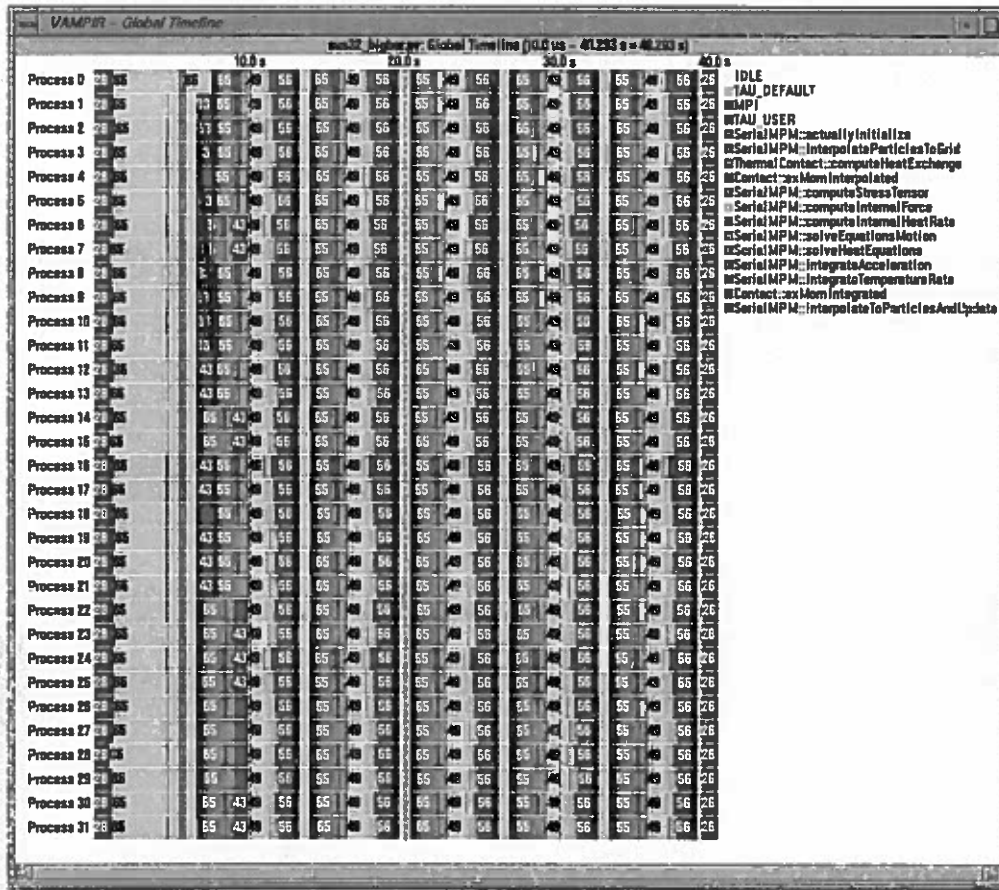


Figure 31. Global timeline shows the distinct phases of task execution

these names correspond to domain-specific abstractions that are not directly represented in the execution.

What the scientist wants to see is the partition of the overall task execution time among different tasks. The similarities to the POOMA mapping problem are apparent. The number of tasks is finite and is typically less than twenty. However, there are several million particles (or work packets) that are distributed across processors and executed. To relate the performance of each particle based on the task to which it belongs, defines the performance mapping problem.

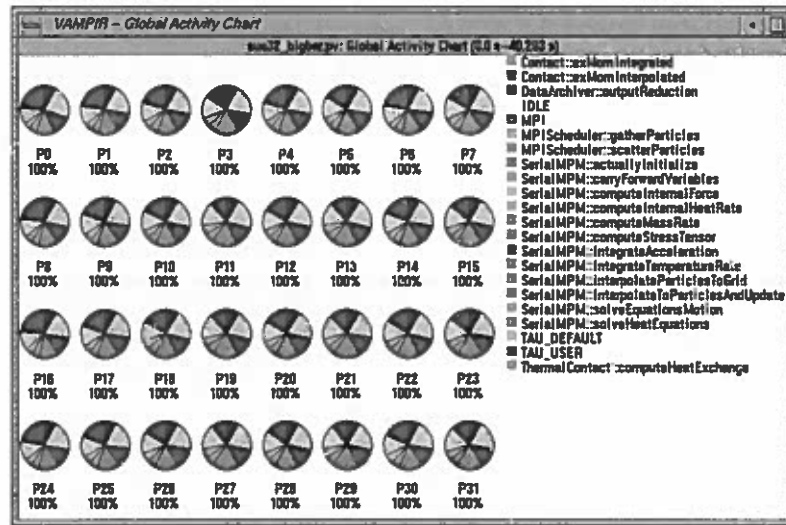


Figure 32. Color-coded activity chart highlights the relative contribution of different tasks

Using our SEAA model of mapping, we form an association, during initialization, between a timer for each task (the task semantic entity) and the task name (its semantic attribute). While processing each work packet in the scheduler, a method to query the task name is invoked and the address of the static character string is returned. Using this address, we do an external map lookup (implemented as a hash-table) and retrieve the address of the timer object (i.e., a runtime semantic association). Next, this timer is started and stopped around the code segment that executes an individual task.

Figure 30 shows the results of this work packet-task mapping performance analysis in Uintah. Again we see the benefit of the SEAA approach in presenting performance data with respect to high-level semantics.

When event-traces are generated, it lets us track the distinct phases of computation. Figure 31 shows a global timeline where each task is color-coded. That is, although we are

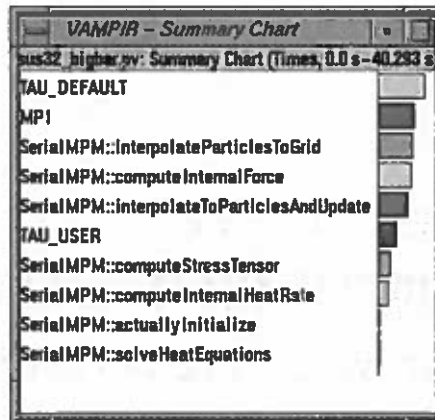


Figure 33. Summary chart

looking at individual work packets being executed, the mapping allows us to “see” their performance data at a high-level. The activity and summary charts for a 32 processor run of the program are shown in Figure 32 and Figure 33 respectively.

4.7 Conclusions

In the above case studies, we have tried to illustrate the application of the SEAA mapping approach to applications where high-level operations are decomposed into low-level work units that are separately executed and (asynchronously) scheduled. We have shown how upward one-to-many mappings involve attributing the cost of one low-level statement to one or more higher-level entities and can be addressed by SEAA. Previous work in this area has focussed on aggregating the cost of multiple instances of the mapped statement to higher-level statements/entities. Our work is able to look at each instance, isolating the per-instance cost and correlate the measurements performed at each instance

to its associated higher level form. This allows a more accurate attribution of measured low-level costs.

In each of the above cases, we are faced with the problem of resolving one-to-many mappings for an iterative computation. Traditional approaches of aggregation or amortization over all iterations do not provide the level of detail or accuracy that we desire. So, we measure each iteration, or invocation of the low-level routine and correlate its cost to higher-level abstractions using application data in conjunction with static semantic entities (routine, statement execution).

Although not the explicit focus of this chapter, in order to perform these experiments we used instrumentation at multiple levels. In POOMA, source-level instrumentation for mapping is combined with preprocessor level instrumentation using an instrumentor based on PDT [69] and instrumentation at the library level (SMARTS). Together these levels cooperate with each other and target a common performance measurement API. In contrast, in Uintah, we used instrumentation at the source level for mapping together with instrumentation at the MPI wrapper interposition library level for composing the performance experiments. Thus, a multi-level instrumentation strategy helps generate performance data in these mapping scenarios.

CHAPTER V

INSTRUMENTATION AWARE COMPILATION

5.1 Introduction

Instrumentation at the source level of a program can focus on the statement, expression, and basic-block levels, allowing language-level semantics to be associated with performance measurements [111]. Typically this is done by inserting calls to a measurement library at appropriate locations. However, because compilers do not distinguish between measurement routines from application routines, the instrumentation and code restructuring transformations performed by many optimizing compilers may mutually interfere:

The tacit assumption underlying source code instrumentation is that the organization and structure of the compiler-generated code are similar to that in the source code. When this assumption is false, instrumentation may either inhibit or change the normal optimizations or it may measure something other than what might be expected when examining the source code.

- Daniel A. Reed [98] pg. 487

This has been an open problem for source-level instrumentation.

In the next section, we describe the issues in more detail. We then introduce the concept of instrumentation-aware compilation that addresses the problem, allowing for accurate source-level instrumentation despite aggressive optimizations. We finish this chapter by discussing our prototype of an instrumentation-aware compiler and its use with both C and ZPL application code.

5.2 Issues in Instrumentation

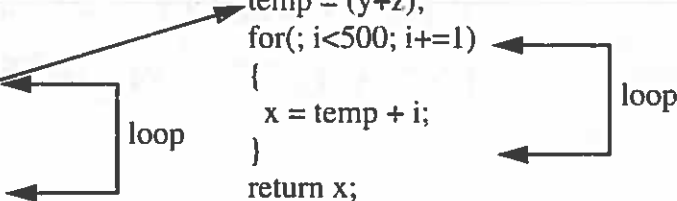
Hypothesis: There exists instrumentation that can inhibit compiler optimizations

Timing instrumentation is often introduced as start timer and stop timer calls placed around a section of code, as shown in Figure 34. Figure 34 (a) shows the original source code for a simple application without instrumentation and Figure 34 (c) shows the same code with instrumentation for a timer inserted around a single statement. Consider what happens as a result of optimization. Figure 34 (b) shows the un-instrumented code after optimization. Loop invariant code motion [2] has been applied to move the subexpression $(y+z)$ out of the loop as shown by the arrow. In the case of the instrumented code of Figure 34 (c), however, x and y are global variables and the compiler conservatively assumes that their values could be changed in the *stop* and *start* routines. As shown in Figure 34 (d), it does not perform the optimization.

This example demonstrates that the presence of instrumentation can inhibit optimizations even for simple cases. What implications does this have for performance

```
void start(void);
void stop(void);
extern int y,z;
int foo()
{
  int i, x;
```

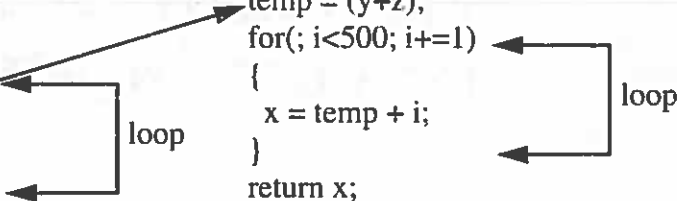
```
  for (i=0; i < 500; i++)
  { /* the block */
    x = y+z+i;
  }
  return x;
}
```



(a) Unoptimized code without instrumentation

```
extern int foo(void);
extern int y; extern int z;
int foo(void){
  auto int i, x, temp;
```

```
  i=0;
  temp = (y+z);
  for(; i<500; i+=1)
  {
    x = temp + i;
  }
  return x;
}
```



(b) Optimized code without instrumentation

```
void start(void);
void stop(void);
extern int y,z;
int foo()
{
  int i, x;
```

```
  for (i=0; i < 500; i++)
  { /* the block */
    start();
    x = y+z+i;
    stop();
  }
  return x;
}
```

(c) Unoptimized code with instrumentation

```
extern void start(void);
extern void stop(void);
extern int y,z;
extern int foo(void);
int foo(void) {
  auto int i; auto int x;
```

```
  i = 0;
  for(; i < 500; i+=1)
  {
    start();
    x = ((y+z)+i);
    stop();
  }
  return x;
}
```

(d) Optimized code with instrumentation

Figure 34. Four versions of a program, demonstrating the effect of code restructuring optimizations on the accuracy of a timer. Uninstrumented code is on the top; instrumented code is on the bottom. Unoptimized code is on the left; optimized code is on the right.¹

¹ To generate this figure, we used a commercial optimizing C++ compiler to generate the equivalent C code by lowering the intermediate representation of the program after optimization. The compiler generated symbols in the code were translated to a more readable, equivalent form by hand.

```

1: #include <stdio.h>
2: #include <Profile/Profiler.h>
3:
4: int x, y;
5: int f(int a, int b, int c, int d)
6: {
7:     int w, z ;
8:     TAU_PROFILE_TIMER(ft, "Contribution of x,y and z", "[f()]",
                       TAU_DEFAULT)
9:
10:    TAU_PROFILE_START(ft);
11:    x = a + a * (b - c);
12:    y = (b - c) * d;
13:    z = x + y;
14:    TAU_PROFILE_STOP(ft);
15:    w = 5*z+6;
16:
17:    return w;
18: }
19:

```

Figure 35. Using a timer to measure the cost of executing three statements

measurement? Clearly, making measurements of unoptimized code can lead to distorted views of performance with respect to its optimized form.

Hypothesis: There exist optimizations that can lead to incorrect measurements

Consider the program in Figure 35, which again uses a simple timer positioning *start* and *stop* methods around a group of statements. The timer, *ft* (declared at line 8), measures the cost of executing the statements at lines 11-13. In contrast with the previous example, the variables *a*, *b*, *c*, and *d* used in the measured code are not global and, hence, the timer calls will not interfere with optimization. Figure 36 shows the unoptimized assembly code for the routine *f* (generated for the Sparc processor). In starting routine *f*

```

1: .section ".data"
2: .size .3,4
3: .align 4
4: .3:
5: .word 0x0
6: .global f
7: .section ".text"
8: .align 4
9: .proc 4
10: f:
11: save %sp,-96,%sp
12: st %i0,[%fp+68]
13: st %i1,[%fp+72]
14: st %i2,[%fp+76]
15: st %i3,[%fp+80]
16: set .3,%o0
17: mov %o0,%o0
18: set .L4,%o1
19: mov %o1,%o1
20: set .L5,%o2
21: mov %o2,%o2
22: set 0xffffffff,%o3
23: mov %o3,%o3
24: call tau_profile_c_timer; nop
25: set .3,%l7
26: ld [%l7],%o0
27: mov %o0,%o0
28: call tau_start_timer; nop
29: set x,%l7
30: ld [%fp+68],%l6
31: ld [%fp+68],%l5
32: ld [%fp+72],%l4
33: ld [%fp+76],%l3
34: sub %l4,%l3,%l4
35: smul %l4,%l5,%l5
36: add %l6,%l5,%l6
37: st %l6,[%l7]
38: set y,%l7
39: ld [%fp+72],%l6
40: ld [%fp+76],%l5
41: sub %l6,%l5,%l6
42: ld [%fp+80],%l5
43: smul %l5,%l6,%l6
44: st %l6,[%l7]
45: set x,%l7
46: ld [%l7],%l7
47: set y,%l6
48: ld [%l6],%l6
49: add %l7,%l6,%i4
50: set .3,%l7
51: ld [%l7],%o0
52: mov %o0,%o0
53: call tau_stop_timer; nop
54: .L2:
55: smul %i4,5,%l7
56: add %l7,6,%i5
57: mov %i5,%i0
58: ret; restore
59: .type f,#function
60: .size f,.-f

```

Timer on

Timer off

Figure 36. Unoptimized assembly code for routine f

(line 10), the stack pointer is saved (line 11) and the four arguments (a , b , c , d) are stored at appropriate locations using the address of the frame pointer (lines 12-15). Next, the output registers $o1$ through $o3$ are set up for the call to the timer registration routine (lines 16-23), the timer is created (line 24), and it is started (lines 25-28). After the timer is started, x is calculated (lines 29, 34-37) by evaluating the subexpression $(b-c)$ (line 34),

multiplying the result by a (line 35), adding a to that (line 36), and storing the final value in x (line 37). Similarly, y is calculated (lines 38-47) by loading b and c (lines 39-40), evaluating $(b-c)$ (line 41), multiplying it by d (lines 42-43), and storing the result in y (line 44). After x , and y are calculated, z is computed by adding x and y (line 49). This is followed by stopping the timer (lines 50-53). After the timer is stopped, w is computed (lines 54-56) by multiplying z with 5 (line 55) and adding 6 to z (line 56), before returning from the routine (lines 57-58) as was the case in our previous example (Figure 34), the unoptimized code measures exactly what the user would expect his/her instrumentation to measure, in this case, the assignment to x , y , and z but not the assignment to w .

When the same program is optimized, however, the code transformations lead to a violation of the user's measurement expectations. Consider the optimized code in Figure 37. There are two problems. In the first case, the compiler performs common subexpression elimination, computing $(b-c)$ in line 18 and using it in lines 20 and 40. This means that the computation of $(b-c)$ and its use in computing $a*(b-c)$ at line 20 both occur *before the timer is created* in lines 21-29; *or started* in lines 30-33. The result is that the timer does not measure the cost of computing $a*(b-c)$! In the second case, $(z*5)$ is computed in line 47 *before stopping the timer* (lines 48-51). Again the optimization is legal since the value of z cannot be affected by the stop timer call. The consequence is that the cost of $(z*5)$ is incorrectly attributed to the timer. Thus, the cost of $a*(b-c)$ is not counted when it should be and the cost of $(z*5)$ is counted when it shouldn't be. This is clearly inconsistent with the user's intentions.


```

1: .section ".data"
2: .size .3,4
3: .align 4
4: .3:
5: .word 0x0
6: .global f
7: .section ".text"
8: .align 4
9: .proc 4
10: f:
11: save %sp,-112,%sp
12: st %i0,[%fp+68]
13: st %i1,[%fp+72]
14: st %i2,[%fp+76]
15: st %i3,[%fp+80]
16: ld [%fp+72],%i5
17: ld [%fp+76],%i4
18: sub %i5,%i4,%i0
19: ld [%fp+68],%i5
20: smul %i0,%i5,%i1
21: set .3,%o0
22: mov %o0,%o0
23: set .L4,%o1
24: mov %o1,%o1
25: set .L5,%o2
26: mov %o2,%o2
27: set 0xffffffff,%o3
28: mov %o3,%o3
29: call tau_profile_c_timer; nop
30: set .3,%i5
31: ld [%i5],%o0
32: mov %o0,%o0
33: call tau_start_timer; nop
34: set x,%i5
35: ld [%fp+68],%i4
36: add %i4,%i1,%i4
37: st %i4,[%i5]
38: set y,%i5
39: ld [%fp+80],%i4
40: smul %i4,%i0,%i4
41: st %i4,[%i5]
42: set x,%i5
43: ld [%i5],%i5
44: set y,%i4
45: ld [%i4],%i4
46: add %i5,%i4,%i0
47: smul %i0,5,%i0
48: set .3,%i5
49: ld [%i5],%o0
50: mov %o0,%o0
51: call tau_stop_timer; nop
52: .L2:
53: add %i0,6,%i0
54: ret; restore
55: .type f,#function
56: .size f,-f

```

(b-c)
 a*(b-c)
 Timer on
 (b-c)*d
 (z*5)
 Timer off
 6+(z*5)

Figure 37. Optimized assembly code for routine f

Thus, we have demonstrated the two key problems in source-level instrumentation in the presence of optimization: performance instrumentation may inhibit the application of optimizations, and optimizations may not preserve the semantics of the instrumentation. Existing tools typically avoid these problems by allowing instrumentation only at the routine level. We next propose a mechanism that would allow

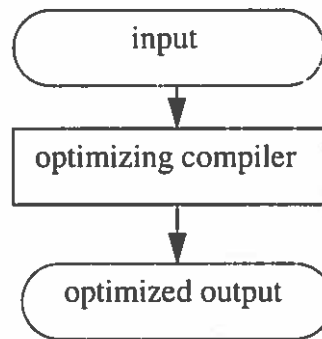


Figure 38. Traditional optimizing compiler

accurate instrumentation down to the granularity of source-level statements despite aggressive optimizations.

5.3 Instrumentation-Aware Compilation

An instrumentation-aware compiler preserves the semantics of instrumentation in the presence of optimizations. Figure 38 shows the model of a simple optimizing compiler. Figure 39 shows the extensions needed for an instrumentation-aware compiler. In that figure, the source code annotated with instrumentation is first fed into a de-instrumentor that strips the annotations and extracts an instrumentation specification. The de-instrumented program is fed into the compiler. Since all instrumentation has been removed, it can not affect optimization. During compilation, the de-instrumented code is also analyzed to produce mapping tables that are updated to reflect code transformations. The mapping information relates the output of the optimizer to the de-instrumented input. It is fed, along with the optimized output and the extracted instrumentation specification to the instrumentor which re-introduces the required instrumentation, attributing the costs of

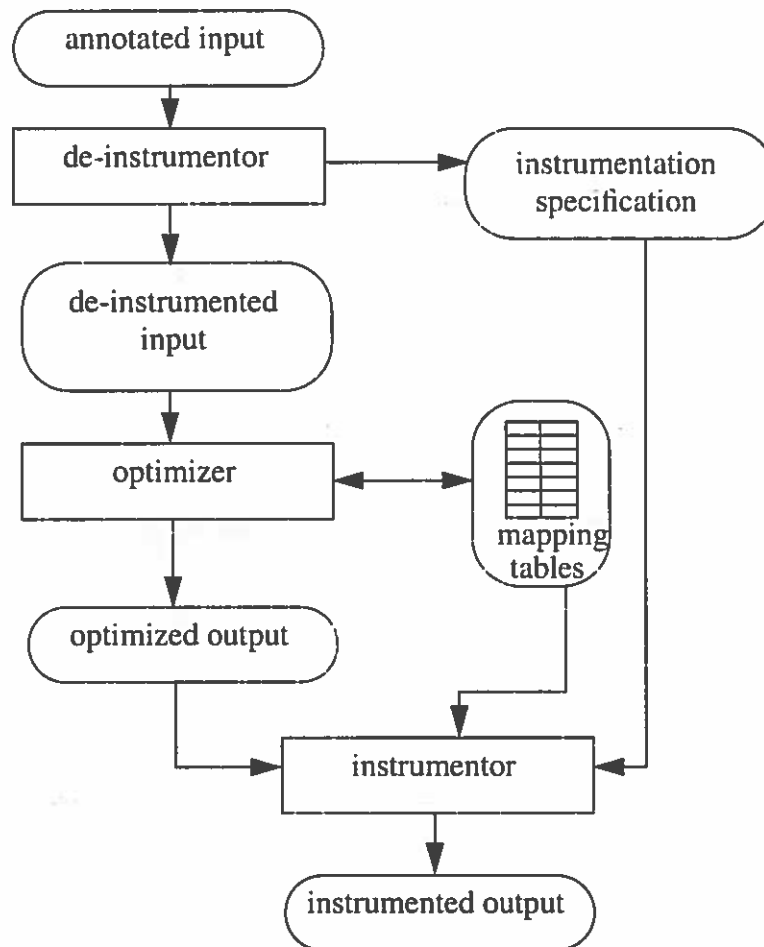


Figure 39. Instrumentation-aware compilation model is cognizant of optimizations

source statements to individual instructions appropriately. The output of the instrumentor is then sent to the back end which generates object code for the target platform. The result of these transformations is that the instrumentation-aware compiler can now generate optimized object code that has better placement of instrumentation annotations for performance measurement.

The maintenance of mapping information during compilation is key. An optimizing compiler has three distinct phases: parsing, optimization, and code generation as shown in Figure 40. A front end parses the source code to build an internal representation of the code. This code list may be in the form of an abstract syntax tree or a forest of directed acyclic graphs. During parsing, as tokens are processed, valuable source information flows through the front end. To build mappings, the first step is to extract source-level information from the parsing phase and store it in the internal representation. The internal codelists are fed to the optimizer. Typically, optimizers prefer input in the form of a flow graph. A flow graph describes the flow of control and data in the program. So, the code lists need to be transformed before the optimizer can perform a sequence of code transformations. The key to building source-level mappings is to preserve the source information during these transformation phases. Appendix I shows examples of different optimizations and how a compiler can maintain correspondences between unoptimized code and optimized code during each transformation phase. During transformation, instructions in the original code list are re-ordered, copied, and deleted. It is important for routines that modify the code list to also keep track of source-level information to prevent any loss of mappings. The mapping correlates individual instructions in the optimized output to the original source-level statements. For further detail, we describe a prototype implementation of an instrumentation-aware compiler for the C language.

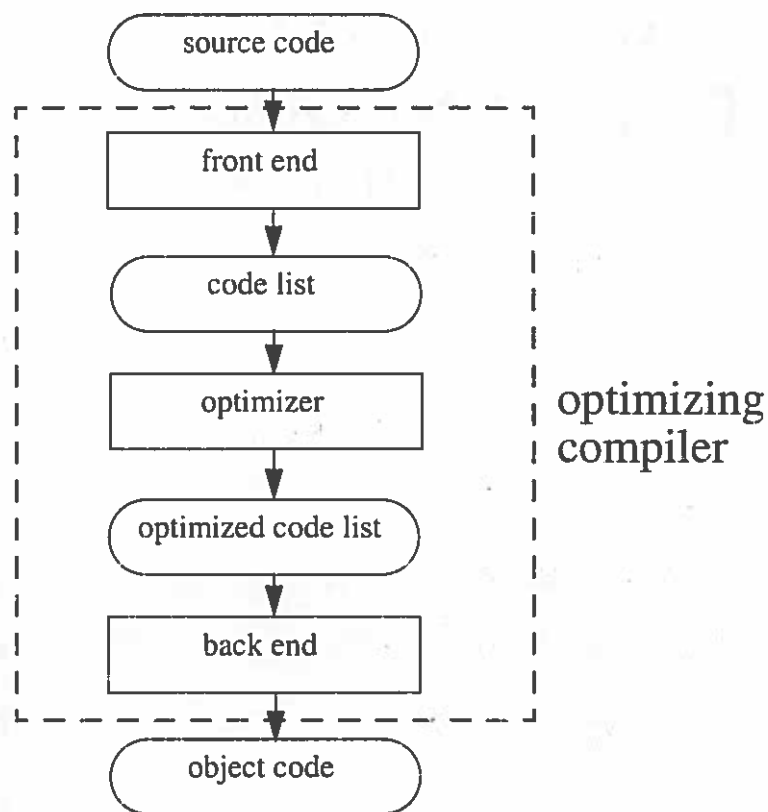


Figure 40. Structure of a typical optimizing compiler

5.3.1 Prototype Implementation

Our prototype of an instrumentation-aware compiler supports performance instrumentation in the form of timer registration, start and stop calls. We started with the source code of the compiler used in the FULLDOC [50] project at the University of Pittsburgh. That compiler extended the publicly available and portable lcc [31] C compiler to include optimizations and support for debugging. It performs statement and loop level optimizations such as constant propagation, loop invariant code motion, dead code

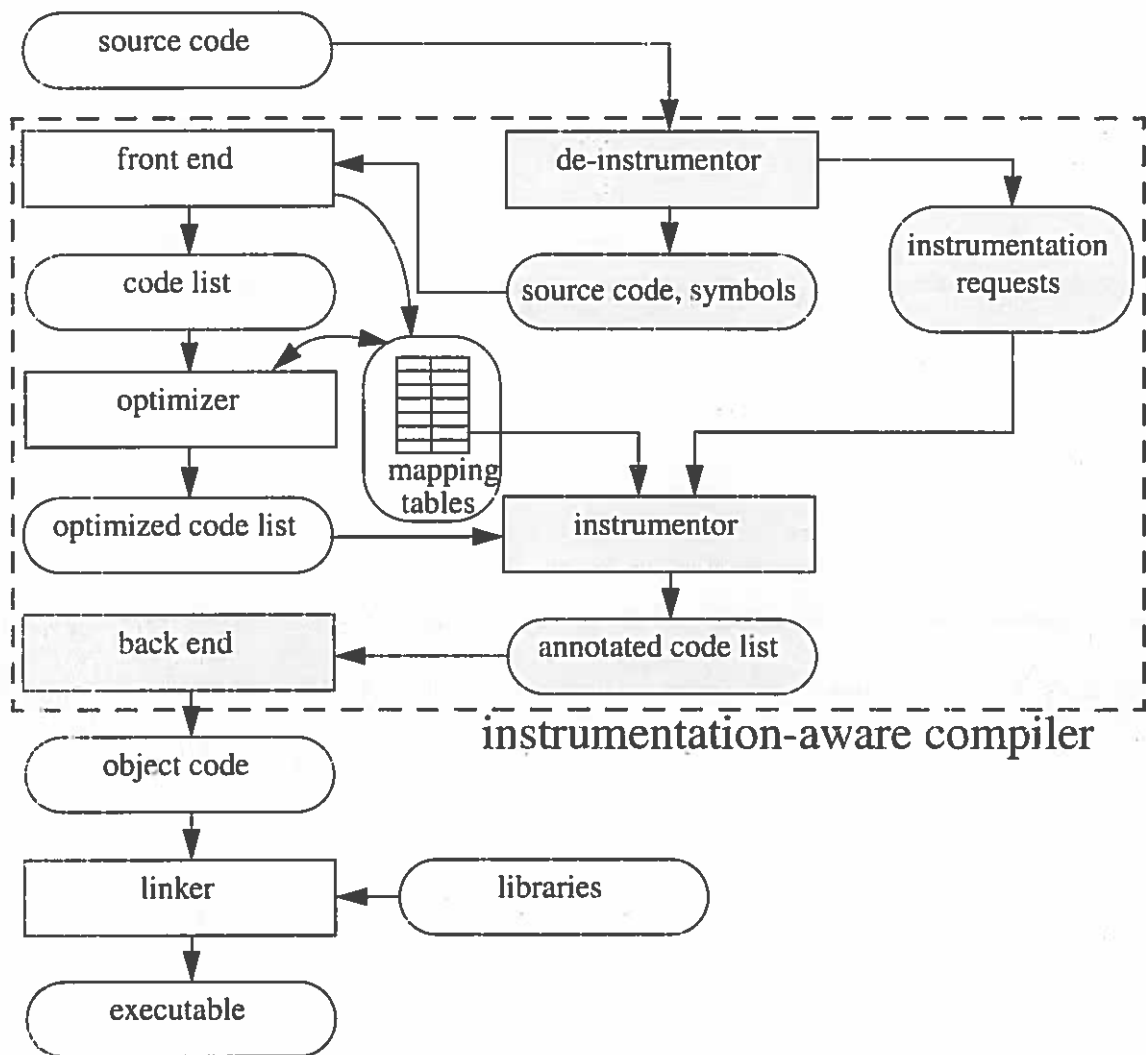


Figure 41. Structure of an instrumentation-aware compiler

elimination, partial redundancy elimination, register allocation, and folding. In addition, the compiler maintains mappings [52] that relate source code locations to corresponding locations in the optimized code through multiple phases of code transformations. This enhanced compiler was used to implement a comparison checker (COP) [51] that compares the computed values of executions of optimized and unoptimized codes; when these differ, it can point out the source location where values differed and which

optimizations were responsible for the differences. This technique is used to debug and validate an optimizer. It was also used to build a full reporting debugger (FULLDOC) that can stop a program between any two source statements and report values of all variables in the optimized program using a combination of static and dynamic information.

While the enhanced optimizing lcc compiler had mappings and optimizations, it was geared towards setting breakpoints between any two statements in the source code and recreating the values of variables at those points. For performance instrumentation, we needed to track the contribution of statements between a pair of *start* and *stop* instrumentation calls in the source code, but we were not interested in exploring the state of the program by querying the values of the variables. The information we needed for fine-grained performance instrumentation was not available in the compiler, so we made a number of straightforward extensions to the compiler.

Figure 41 shows the structure of our prototype instrumentation-aware compiler. The following describes each of the added components needed to build our instrumentation-aware compiler as indicated by the shaded rectangles in Figure 41.

5.3.1.1 De-instrumentor

As shown in Figure 42, the de-instrumentor removes all instrumentation from the code, building the de-instrumented source code, the instrumentation specification, and the symbol mappings. We implement it as a source-to-source translator that recognizes instrumentation annotations by their *TAU_* prefix. As an example, typical timer

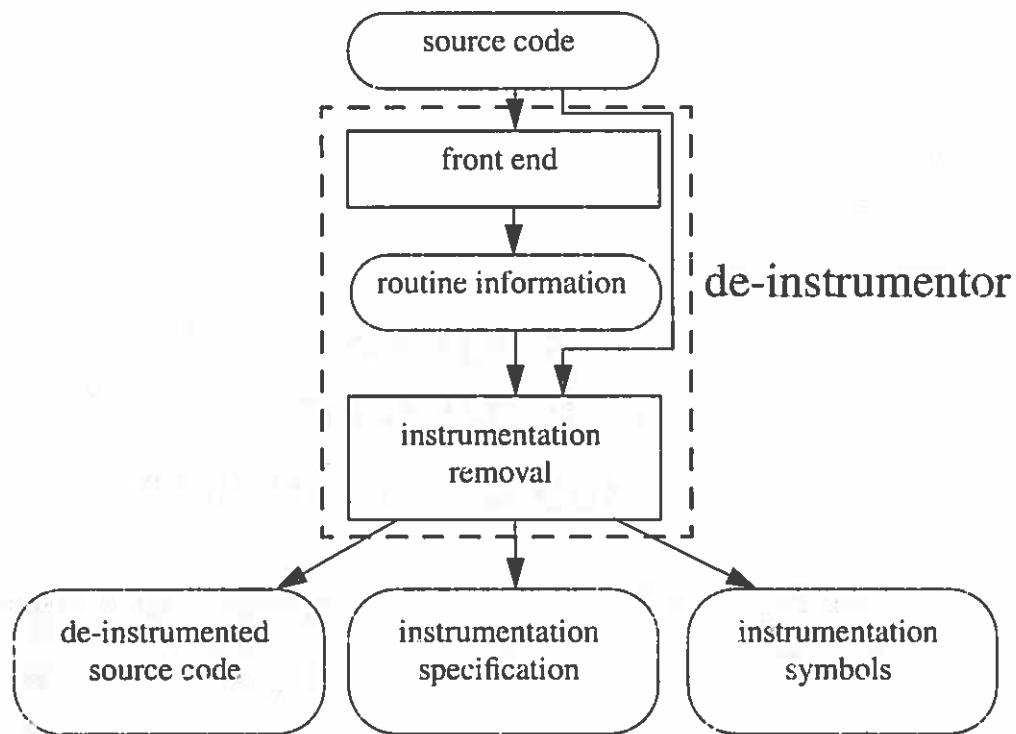


Figure 42: Structure of a de-instrumentor

declarations and timer start and stop calls were shown in Figure 35. Note that they are created inside a routine. Thus, the same timer symbol (*ft* in Figure 35) may be used in different routines with different scope. The instrumentation specification produced by our de-instrumentor, records both location and scope. Each location is a tuple comprised of the source file name, the row (or y co-ordinate) and the column number (or x co-ordinate) in the source file. The properties of timers (the group to which they belong, name, symbol and their scope) are stored separately in an instrumentation symbols file which is compiled along with the de-instrumented source file. Entities in the symbols file are static variables with global scope. They are used as arguments for timer declarations and thus do not interfere with any optimizations in the compiler. The de-instrumentor replaces each

instrumentation annotation with a blank line, retaining the original line numbering scheme of the annotated source code.

The de-instrumentor for our prototype implementation consists of a compiler front end phase that passes through the source to produce routine location information (routine name and extent of routine in terms of location of its starting and ending position). It was convenient to use the front end from the optimizing compiler for this to maintain compatibility in routine naming schemes. The routine information, along with the original source code is fed to the instrumentation removal phase. In this phase, instrumentation constructs are recognized and separated from the source code. This is achieved with a simple token parser based on lex and yacc unix tools. Instrumentation annotations are then combined with routine information to construct timer names with global scope; this is done by appending the local name to its routine name (e.g., in our case, a timer *ft* in routine *f* is called *f_ft* in the global scope). This design permits the presence of multiple timers within a routine as well as in different routines in the compilation unit.

5.3.1.2 Instrumentor

In our implementation, the instrumentor handles three timer instrumentation calls: register, start and stop timer. It annotates the code list with appropriate calls and passes it to the code generating back end.

Start and stop timer annotations around a section of code are interpreted by the instrumentor to mean that all optimized instructions with source correspondences in the

```

Instrument:
for each timer t
  for each source range r of t
    state = INACTIVE; previous = null;
    for each instruction i
      if i belongs to r then
        if state is INACTIVE then
          allocate instrumentation to start t before executing i
          previous = i; state = ACTIVE;
        end if
        if state is ACTIVE then
          previous = i;
        end if
      end if

      if i does not belong to r then
        if state is ACTIVE then
          allocate instrumentation to stop t after executing previous
          previous = null; state = INACTIVE;
        end if
      end if
    end for
  end for
end for

```

Figure 43. Algorithm for fine-grained source-level performance instrumentation

delimited region of code are to be instrumented. This interpretation is tool-specific and may vary for different kinds of instrumentation. The algorithm we use here is shown in Figure 43. The accuracy of instrumentation produced by this algorithm depends on the presence of complete source-level mapping information for each instruction. (We did not have complete information as discussed in Section 5.3.4.)

5.3.1.3 Back end

The back end converts the instructions to assembly code which is passed to the assembler. The back end is responsible for interpreting the instrumentation annotations in the code list, according to our prototype implementation. Each instruction in the code list has an ordered list of timers associated with it. The properties of each timer include its qualified name, operation (start or stop) and location (before or after the instruction executes). The back end examines this list and generates instrumentation accordingly. To instrument the application, we used the fast breakpoint scheme proposed by Peter Kessler [58]. This scheme allows us to implement low-overhead breakpoints for performance instrumentation. The prototype implements fast breakpoints for the Sparc processor, which includes register windows [115].

To plant a fast breakpoint before an instruction that calls a measurement library routine, the scheme replaces the instruction with a branch instruction to the breakpoint. At the breakpoint, state registers are saved and the measurement routine is called. After it executes, registers are restored and the replaced instruction is executed and normal execution resumes, as described in [58]. Although storing the execution state normally includes saving the floating point registers, saving them would involve allocation of space which is expensive. To avoid this, we exploit the fact that our measurement routines do not perform floating point operations and consequently we do not save and restore them.

This scheme is light weight as compared to unix level breakpoints that involve the use of a signal (SIGTRAP) and have an overhead of the order of milliseconds. Using our

prototype, the overhead of a pair of TAU start and stop calls is just 2.16 *microseconds*. (This cost was determined by amortizing over ten million calls to a profiled routine and includes both the TAU overhead for profiling as well as the fast breakpoint overhead as observed on a Sun UltraSPARC-III processor running at 440 MHz under the Solaris 8 operating system.)

5.3.2 Perturbation Issues

Instrumentation can perturb the performance of an application. No matter how low the instrumentation overhead, the presence of instrumentation can change the computation, creating a *probe-effect* [32]. Because of the small relative difference between the execution time of a code segment and the instrumentation overhead needed to measure it, the fine-grained instrumentation that we support can in fact have greater impact on the perturbation than previous routine-based instrumentation. The pressure on tools to generate fine-grained instrumentation conflicts with the pressure to reduce the overhead of instrumentation, forcing trade-offs. Our approach allows the user to selectively request fine-grained instrumentation which will have a lower overhead than instrumenting all statements. We also provide access to the low-overhead free running timers and hardware performance counters found on most microprocessors, further minimizing measurement overhead. Such a hybrid instrumentation approach, using both non-intrusive hardware measurement options and intrusive software-based instrumentation allows us to balance the measurement overhead. However, it is beyond the scope of this work to eliminate or

compensate for this overhead for parallel programs; that is best addressed by other research in this area. Previous work, for example, has produced techniques for perturbation compensation in order to minimize the distortion of performance data [71][72]. Although, we do not directly implement a perturbation compensation scheme in our work, we believe the interface with the compiler will help to identify points where there may be significant perturbation effects. This will be important for future perturbation control techniques.

5.3.3 Results

The instrumentation-aware compiler prototype generates accurate, fine-grained measurements in the presence of aggressive optimizations. Figure 44 shows, for example, the code produced for the statements computing the contribution of x , y and z from the code previously seen in Figure 35. Instrumentation calls were inserted to time the entire routine as well as statement-level timers (in this case to evaluate the contribution of evaluation of x , y and z). Note that after entering routine f (line 5), the control branches to a fast breakpoint ($FB1_f$ at line 36) where the registers are saved (line 38) and two timers are created (lines 40-60) – one for the routine f and another for the timer ft (called f_ft here as its scope is within routine f). The computation of $(b-c)$ occurs at the breakpoint $FB2_0$ (line 75) after the timer ft is started (line 73). Dashed arrows in the figure show the path of execution. The flow returns to the routine (line 15) and x is computed (lines 17-22) by multiplying $(b-c)$ to a (line 20) and adding the result to a (line 21). Next, y is computed

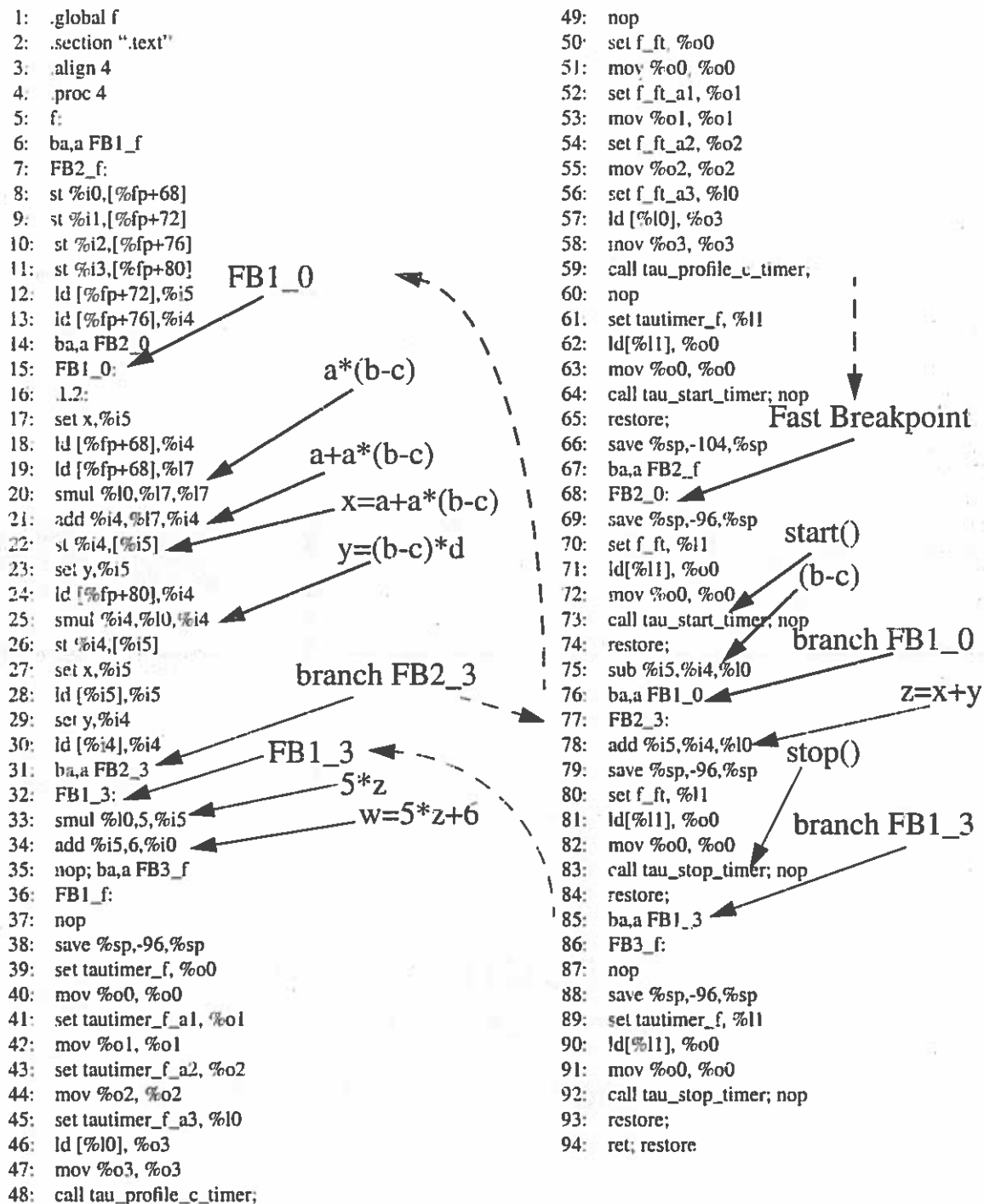


Figure 44. Optimized code generated by the instrumentation-aware compiler

(lines 28-29) by *re-using* $(b-c)$ and multiplying it with d (line 25). To compute z , x and y are loaded (lines 27-30) and the flow branches to the fast breakpoint `FB2_3` (line 31) where z is computed as $(x+y)$ (line 78) before stopping the timer `ft` (line 83) and returning to the routine at label `FBI_3`. This is followed by computing w (lines 33-34) by multiplying z with 5 (line 33) and adding the result to 6 (line 34). So, the timing of `ft` includes the computation of x , y , and z and excludes the computation of w as the instrumentation specifies; the time for computing $(b-c)$ is included in the measurement.

The assembly file, as shown in Figure 44 is then converted to binary code by the assembler. When the binary file is linked with measurement libraries and executed, it generates performance data as shown in Figure 45, showing the time spent in the timer `ft` ("*Contribution of x , y and z [$f()$]*").

5.3.4 Incomplete Source-Level Information

In order to accurately measure all code, an instrumentation-aware compiler needs complete mappings from all generated instructions to the source level. Because such complete mappings were not needed for debugging, they were not available from the `FULLDOC` compiler. Specifically, we needed additional information on arguments to routines, jump instructions, routine invocations, and compiler generated variables. We were able to get some of this information by examining the generated instructions that did not map to the source level. If such an instruction were the definition of a variable (def point) created by the compiler, for example, the variable was probably a temporary that

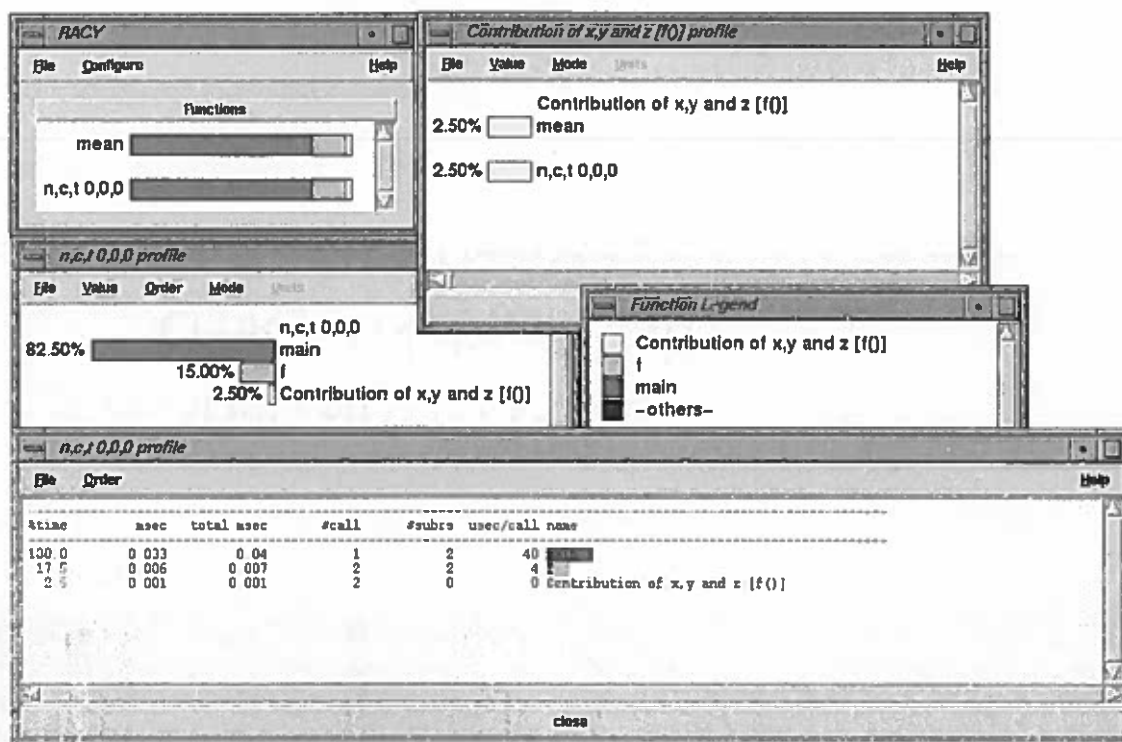


Figure 45. Performance data from an optimized program compiled with the instrumentation-aware compiler

was used elsewhere. In such a case, we employed the compiler's def-use analysis to compute the set of instructions that use the temporary such that there is a path in the control flow graph from the define point to the use that does not re-define the variable [2]. We then recursively computed the source location of all instructions in that set to form the mapping. To include the mapping of routine invocations and their arguments, we extended the parser to record the needed routine and argument information in the instruction data structure and ensured that all compiler routines subsequently copy the contents of this data structure as well. With these techniques, we were able to improve the coverage of our mappings.

TABLE 3: Mapping information for three benchmarks

	SIMPLE	NAS 2.3 IS	ZPL Jacobi
Total optimized instructions in codelist	4637	3924	11074
Instructions missing source mappings	1170 (25.2%)	837 (21.3%)	1778 (16%)
JUMP instructions	519 (6.8%)	383 (9.7%)	619 (5.5%)
Instructions with source mappings	3467 (74.8%)	3087 (78.7%)	9296 (84%)

To determine the extent of missing source-level mappings, we took three benchmarks: the NAS Parallel Benchmark 2.3 IS suite from NASA [8], the MPI version of the SIMPLE hydrodynamics benchmark from LLNL [67], and the Jacobi ZPL program from University of Washington [68][21]. We ran each of these programs through our prototype and calculated the number of optimized instructions in the codelist that were missing source-level mappings. Table 3 shows that the percentage of instructions that did not have source-level mappings ranged from 25.2% for the SIMPLE benchmark to 16% for the ZPL Jacobi benchmark. It also shows the contribution of JUMP instructions to this missing information. Thus, these benchmarks show that 74% to 84% of optimized instructions in these specific benchmarks had source-level mappings associated with them. We believe that this missing information is available from the optimizer and we are continuing our implementation efforts to produce it.

If for a particular implementation complete mappings are not available, however, it is possible for the instrumentor to initiate a dialogue with the user to help bridge the gap. It can show the source code in the *vicinity* of the instrumentation request location, (Figure 46 shows a portion of source code for the SIMPLE Benchmark) and show the available

```
404  if (irecv) {
405      MPI_Irecv(recvbuff,2,MPI_DOUBLE,src,tag,MPI_COMM_WORLD,
                                     &recv_request);
406  }
407  if (isend) {
408      sendbuff[0] = r[xlo+xdeltax][ylo+xdeltay];
409      sendbuff[1] = z[xlo+xdeltax][ylo+xdeltay];
410      MPI_Isend(sendbuff,2,MPI_DOUBLE,dst,tag, MPI_COMM_WORLD,
                                     &send_request);
411  }
```

Figure 46. Source code of an MPI application

the user to specify the location in the optimized code list where an instrumentation call should be inserted. While this approach goes against the goal of transparency of instrumentation, it may be very useful to the sophisticated user during the performance experimentation process. The instrumentor might also want to initiate a dialogue with the user in the case where it determines that the instrumentation request is not feasible. For example, in a message vectorization optimization, several small message communication requests are replaced by one message communication that appends several small buffers in a vector and performs the communication request. If the user places fine-grained instrumentation around one such message synchronization operation, and the instrumentation in the optimized code can only account for the time for the aggregated message operation, the instrumentor may classify the instrumentation request as infeasible and communicate this to the user. The other alternative in this case may be to partition the observed latency and/or bandwidth for the large message in terms of smaller messages, but this could lead to erroneous results. Thus, there are some cases, when the instrumentor

```

FUNCTION: reflect_boundary
LABEL 392:
OPT: $611=1          SRC:{ek-simple.c.tau.c, x=1, y=400}
OPT: $610=myindex-numscols SRC:{ek-simple.c.tau.c, x=1, y=401}
  LABEL 393:
  LABEL 385:
OPT: $613 == 0 L411  SRC:{ek-simple.c.tau.c, x=8, y=404}
OPT: ARG recvbuff    SRC:{ek-simple.c.tau.c, x=6, y=405}
OPT: ARG 2           SRC:{ek-simple.c.tau.c, x=6, y=405}
OPT: ARG 11          SRC:{ek-simple.c.tau.c, x=6, y=405}
OPT: ARG $612        SRC:{ek-simple.c.tau.c, x=6, y=405}
OPT: ARG $614        SRC:{ek-simple.c.tau.c, x=6, y=405}
OPT: ARG 91          SRC:{ek-simple.c.tau.c, x=6, y=405}
OPT: ARG recv_request SRC:{ek-simple.c.tau.c, x=6, y=405}
OPT: CALL MPI_Irecv  SRC:{ek-simple.c.tau.c, x=15, y=405}
  LABEL 411:
  LABEL 394:
OPT: $611 == 0 L412  SRC:{ek-simple.c.tau.c, x=8, y=407}
OPT: $609=ylo+xdeltay<<3 SRC:{ek-simple.c.tau.c, x=6, y=409} SRC:{ek-simple.c.tau.c, x=6, y=408}
OPT: $608=192*xlo+xdeltax SRC:{ek-simple.c.tau.c, x=6, y=409} SRC:{ek-simple.c.tau.c, x=6, y=408}
OPT: sendbuff=r[$608][$609] SRC:{ek-simple.c.tau.c, x=6, y=408}
OPT: sendbuff[8]=z[$608][$609] SRC:{ek-simple.c.tau.c, x=6, y=409}
OPT: ARG sendbuff      SRC:{ek-simple.c.tau.c, x=6, y=410}
OPT: ARG 2             SRC:{ek-simple.c.tau.c, x=6, y=410}
OPT: ARG 11           SRC:{ek-simple.c.tau.c, x=6, y=410}
OPT: ARG $610         SRC:{ek-simple.c.tau.c, x=6, y=410}
OPT: ARG $614         SRC:{ek-simple.c.tau.c, x=6, y=410}
OPT: ARG 91           SRC:{ek-simple.c.tau.c, x=6, y=410}
OPT: ARG send_request SRC:{ek-simple.c.tau.c, x=6, y=410}
OPT: CALL MPI_Isend   SRC:{ek-simple.c.tau.c, x=15, y=410}

```

Figure 47. Mapping table for a small portion of the code

may choose to deny the instrumentation request and report the reasons for doing this to the user.

The notion of cooperation, in the form of a dialogue, between the compiler and the user is not new. It has been used in vector compilers [138] and Parallelizing compilers [66]

where a compiler reports on parts of the program that cannot be effectively vectorized or transformed into concurrent code segments and the reasons why it cannot do so. This helps the user restructure the code and achieve better performance.

5.4 Case Study: Successive Program Transformations in ZPL

Some high level languages such as ZPL or HPF are typically compiled in multiple stages using successive optimizing transformations. A program written in ZPL, for example, is converted by the ZPL optimizing compiler to a C program, which is then compiled and further optimized by the C compiler. From there, ZPL programs are linked with the runtime system libraries to produce a parallel executable image, as shown in Figure 48. Instrumentation-aware compilation will need to be applied in both compilers as shown in Figure 49. For this case study, we did not implement a full version of the instrumentation-aware ZPL compiler and are thus limited to instrumenting each statement in ZPL (that is, we do not have a de-instrumentor/instrumentor that would allow selective measurements). The first transformation from ZPL to C produces mapping tables that relate the optimized C code to the ZPL source code. Those mappings were originally generated for the ZEE debugger [76]. Figure 50 shows the code for the ZPL Jacobi benchmark. Figure 51 shows the mapping that relates the ZPL source lines (in column 1) to corresponding C computation (in column 3); columns 2 (pre) and 4 (post) show the extent of loops involved in the computation of the ZPL statement. We then instrumented the C code, as shown in Figure 52, manually, using only the information generated by the

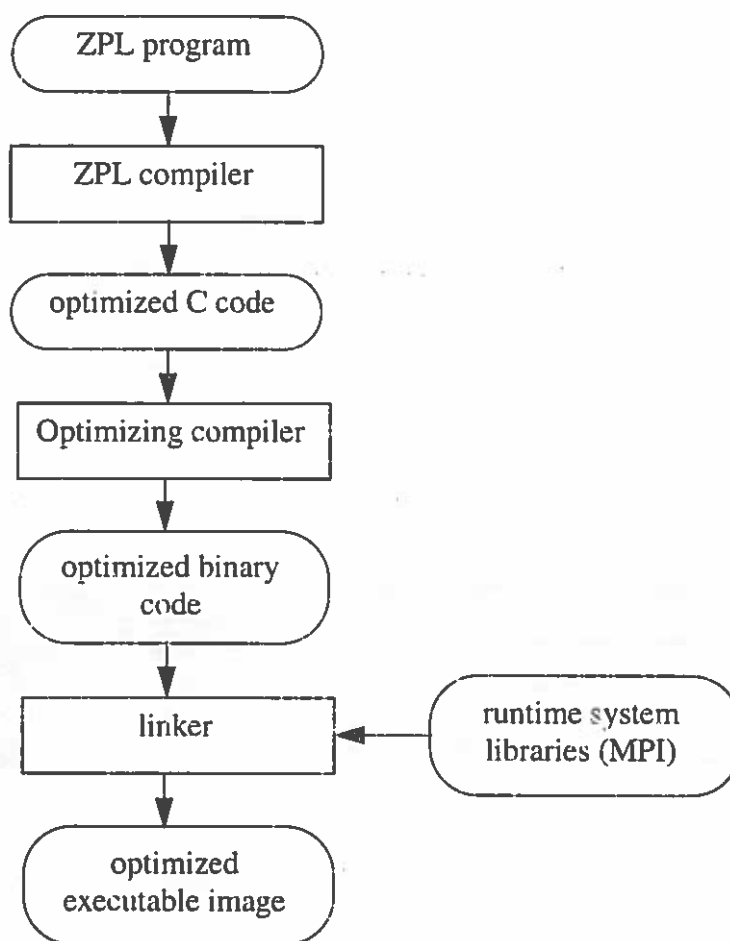


Figure 48. ZPL's compilation model

ZPL compiler for mapping. The hand instrumentation phase generates an instrumented C program as shown in Figure 53. We generate optimized code for it by compiling it with our prototype C compiler. Thus, the original ZPL code undergoes two stages of optimizations – one by the ZPL compiler and the other by the C compiler before generating the instrumented code.

This process provided good coverage for the source code, but ZPL uses a complex runtime layer for achieving portability, using both a portable layer, IronMan [22] that

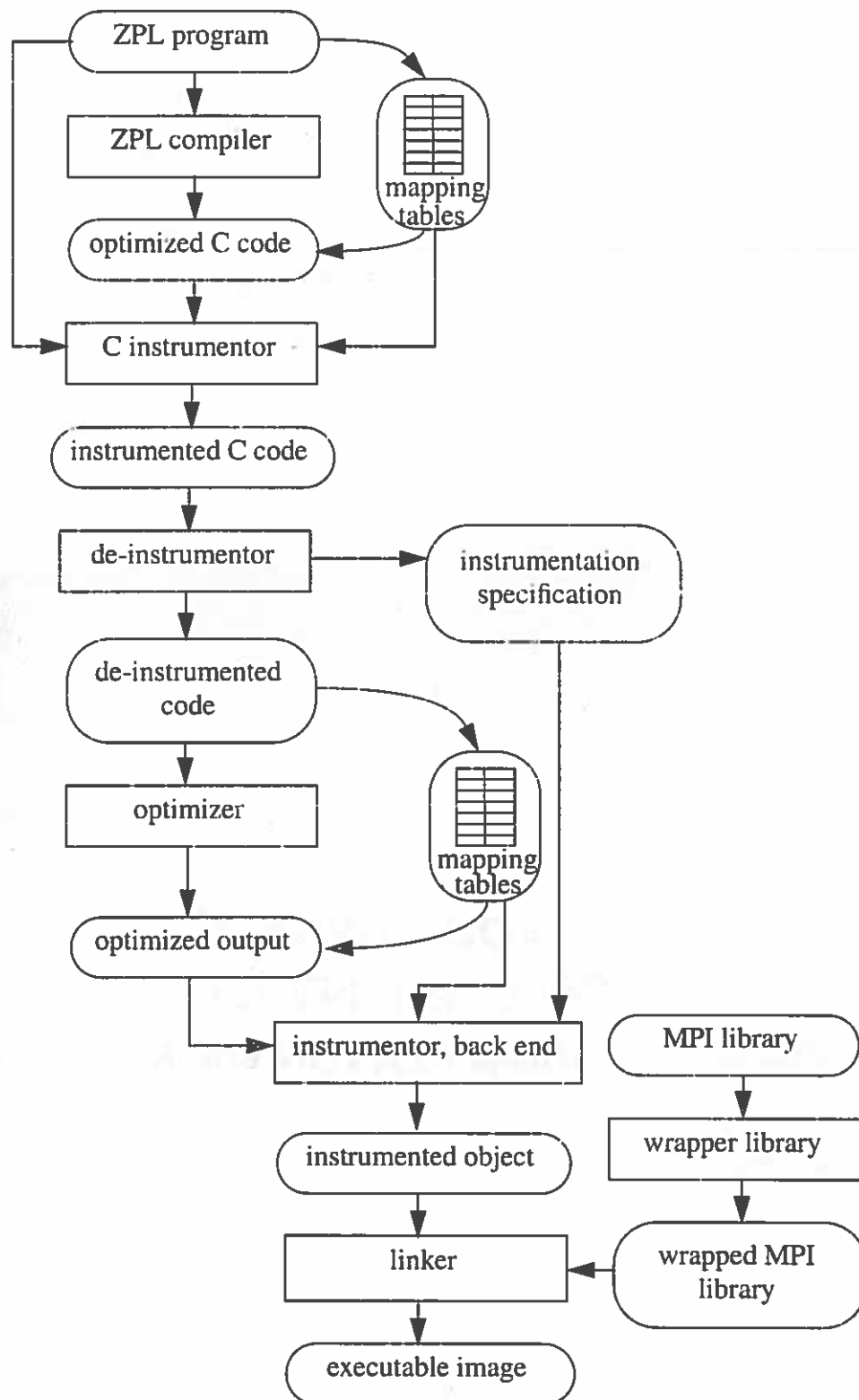


Figure 49. Multi-stage instrumentation-aware compilation in ZPL

```

17 procedure Jacobi();
18 begin
19   [R]      A := 0.0;           -- Initialization
20   [north of R] A := 0.0;
21   [east of R]  A := 0.0;
22   [west of R]  A := 0.0;
23   [south of R] A := 1.0;
24
25   [R]      repeat           -- Body
26             Temp := (A@north+A@east+A@west+A@south)/4.0;
27             err := max<< fabs(A-Temp);
28             A := Temp;
29             until err < delta;
30
31   [R]      writeln(A);
32 end:

```

Figure 50. Source code of a ZPL program

ZPL	C-pre	C	C-post
17	-1	-1	-1
18	70	67	74
19	91	88	95
20	112	109	116
21	133	130	137
22	154	151	158
23	-1	-1	-1
24	171	-1	-1
25	252	249	256
26	257	-1	-1
27	287	284	291
28	-1	-1	-1

Figure 51. Mapping Table from the ZPL compiler

targets different modes of inter-process communication, and the MPI library. To measure time spent in the runtime system, we used the multi-level instrumentation strategy presented in Chapter III. To target the contribution of IronMan, we again use the

```

108  {
109  /* begin MLOOP *** line 21 of jacobi.z */
110  for (_i0 = _REG_MYLO(_R_east_of_R,0); _i0 <=
_REG_MYHI(_R_east_of_R,0); _i0 += _REG_STRIDE(_R_east_of_R,0)) {
111  for (_i1 = _REG_MYLO(_R_east_of_R,1); _i1 <=
_REG_MYHI(_R_east_of_R,1); _i1 += _REG_STRIDE(_R_east_of_R,1)) {
112  (*((double *)_F_ACCESS_2D(A,_i0,_i1)) = 0.0;
113  }
114  }
115  }
116  /* end MLOOP */

```

Figure 52. Optimized C program generated by the ZPL compiler

instrumentation-aware compiler; this time to instrument IronMan routines. To target the contributions from MPI, we use the MPI Profiling interface, and generate a wrapper interposition library to target its instrumentation.

This multi-level instrumentation framework targets one common measurement API, and streams of performance data flow into one common performance data repository to present a seamlessly integrated picture. The interfaces cooperate with each other: MPI wrapper library knows information about the rank of the MPI process and the compiler-based instrumentation interface uses this information to store data in appropriate files after the top level routine exits from the program. Figure 54 shows a profile of the optimized ZPL program. Note that ZPL program statements, such as “[R] A:=0;” and “err := max<<fabs(A-Temp)” are integrated in the profile with the ZPL runtime system entities, such as “_SetupRegionWithDirections,” and MPI level entities, such as “MPI_Recv()”.

This ZPL case study shows us how to apply multi-level instrumentation for wider coverage, and how to integrate multiple cooperating instrumentation interfaces. It also


```

#include "jacobi_usr.h"
void Jacobi() {
    TAU_PROFILE_TIMER(t19, "[R]      A := 0.0;"," ", TAU_USER);
    TAU_PROFILE_TIMER(t20, "[north of R] A := 0.0;"," ", TAU_USER);
    TAU_PROFILE_TIMER(t21, "[east of R] A := 0.0;"," ", TAU_USER);
    TAU_PROFILE_TIMER(t22, "[west of R] A := 0.0;"," ", TAU_USER);
    TAU_PROFILE_TIMER(t23, "[south of R] A := 1.0;"," ", TAU_USER);
    TAU_PROFILE_TIMER(t25, "[R]      repeat"," ", TAU_USER);
    TAU_PROFILE_TIMER(t26, "Temp := (A@north+A@east+A@west+A@south)/", " ", TAU_USER);
    TAU_PROFILE_TIMER(t27, "err := max<< fabs(A-Temp);"," ", TAU_USER);
    TAU_PROFILE_TIMER(t28, "A := Temp;"," ", TAU_USER);
    TAU_PROFILE_TIMER(t31, "[R]      writeln(A);"," ", TAU_USER);

    {
        /* begin MLOOP *** line 21 of jacobi.z */
        for (_i0 = _REG_MYLO(_R_east_of_R,0); _i0 <= _REG_MYHI(_R_east_of_R,0);
            _i0 += _REG_STRIDE(_R_east_of_R,0)) {
            for (_i1 = _REG_MYLO(_R_east_of_R,1); _i1 <=
                _REG_MYHI(_R_east_of_R,1); _i1 += _REG_STRIDE(_R_east_of_R,1)) {
                TAU_PROFILE_START(t21);
                *((double*)_F_ACCESS_2D(A,_i0,_i1)) = 0.0;
                TAU_PROFILE_STOP(t21);
            }
        }
    }
    /* end MLOOP */
}

```

Figure 53. Instrumented C code the ZPL application

shows the viability of applying our instrumentation model to multiple stages of optimizations from two different compilers, cascaded to form an optimized executable with both fine-grained statement and coarse-grained routine level instrumentation.

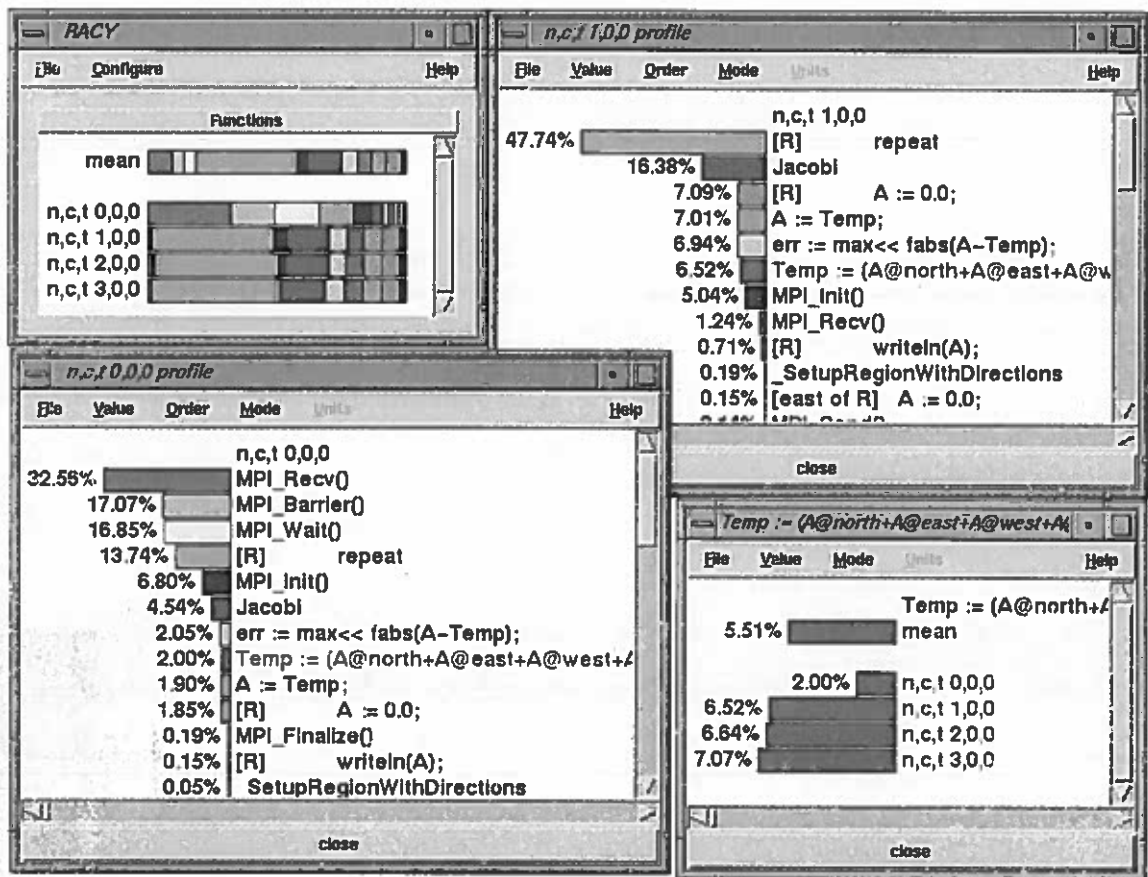


Figure 54. Performance data from the optimized ZPL program compiled by the instrumentation-aware compiler

5.5 Conclusions

Performance tools that limit the choices of instrumentation consequently limit performance exploration. Typically, compilers don't support fine-grained instrumentation because it can interfere with optimizations. In this work, we developed a model for instrumenting a program in the presence of compiler transformations, allowing users to explore the performance characteristics of their code with much finer resolution. Rapidly

evolving programming environments that use diverse language, execution model, compiler, and runtime system technologies [13] often make it difficult to perform even simple performance experiments. We have illustrated here that it is possible to map data from multiple levels of instrumentation back to the source level, in order to present a coherent, and unified view of an execution.

CHAPTER VI

CONCLUSIONS

Empirical performance evaluation is a process to characterize, understand and improve a program's performance based on the performance observation of real systems. Instrumentation is the heart of performance observation as it determines when, where, and how performance measurement can be performed. In this dissertation, we address three of the most egregious limitations of instrumentation and measurement in existing performance evaluation tools: limited execution coverage, lack of abstraction of low-level data, and the granularity of observation.

Existing performance tools typically focus on just one level of program representation, introducing instrumentation at a single stage of program transformation, from source to compilation to execution, resulting in gaps in coverage. The limited observation scope becomes a serious problem in complex environments where a combination of languages, compilation models, multi-level software optimization strategies, and runtime systems is used. We present

- *a framework for multi-level instrumentation that supports multiple cooperating instrumentation interfaces for performance observation.*

Such a multi-level strategy is ultimately more powerful than instrumentation strategies based on single levels of program focus. We describe a performance observation framework that simultaneously supports instrumentation at the source, preprocessor, compiler, library, runtime, and virtual machine level. We show the efficacy of the multi-level instrumentation approach through demonstrations of the framework in case studies where sophisticated, integrated performance views are produced. These results indicate that performance observation coverage can be improved when limitations of one instrumentation level can be overcome through knowledge sharing and support across levels.

With better coverage comes an increased need to interpret performance data.

While a tool may generate a plethora of low-level performance data, that data does not help the user understand the program's performance at his/her conceptual level, unless it is somehow converted into meaningful performance knowledge. Measurement data must be related back to the abstractions understood by the programmer. We introduce

- *a performance mapping technique called Semantic Entities, Attributes, and Associations (SEAA) that allows us to bridge the semantic gap between the programmer's conceptualizations and performance data.*

SEAA advances dynamic mapping work in several ways. First, it allows semantic entities to be created and used in mapping without having explicit relation to program constructs. Second, it provides a way to encode semantic information into entities for use in qualifying mapping relationships. Third, it provides the link to dynamic performance cost mapping based on semantic attributes. Our performance observation framework includes a

a prototype of the SEAA performance mapping concept and we show its application to two case studies.

Flexibility of instrumentation allows for examining the performance of a program to an arbitrary level of detail. Fine-grained instrumentation at the source level, however, interacts with compiler optimizations: instrumentation may inhibit compiler optimizations, and optimizations may corrupt measurement code. We show how to avoid this by

- *extending traditional compilers to instrumentation-aware compilers that can preserve the semantics of fine-grained performance instrumentation despite aggressive program restructuring.*

We also show, with the help of a prototype implementation, how the instrumentation-aware compilation model can be applied to the compilation of high-level parallel languages that require more than one compilation (and consequently optimization) phase. With the help of a case study for ZPL compilation, we demonstrate how low-level performance data can be mapped to higher-level ZPL statements using a multi-level instrumentation strategy.

The models and techniques presented in this dissertation, lay the ground work for building an extensible, empirical performance evaluation infrastructure. Such an infrastructure would allow the configuration of diverse performance observation tools within a framework. It would support fine-grained instrumentation of source code despite aggressive optimizations, and it would support multi-level instrumentation, and the

mapping of low-level performance data to high-level abstractions. It would provide unprecedented flexibility in empirical performance observation to fully explore a program's performance space.

APPENDIX

MAINTAINING SOURCE-LEVEL MAPPINGS

In Chapter V, we describe how source-level mappings are used by the instrumentor of an instrumentation-aware compiler to track the contribution of different fragments of code. Mappings embed a cost model for associating the cost of executing instructions with source-level entities. Here, with the help of examples, we describe how we can correlate execution costs and create mappings during statement level optimizations. We study a variety of optimizations and show how to maintain correspondences between optimized code and source-level information. Maintaining such correspondences is not new. It has been studied extensively in the context of debugging optimized code [53] where mappings relate statement instances between optimized and unoptimized code. Since a set of transformations may be applied successively on a group of statements, it is important to maintain the source-level mappings at each stage.

Terminology

In the examples that follow, we represent the unoptimized code on the left and optimized code on the right. Unoptimized code statements are numbered successively as

U_1, U_2 , etc. and optimized code statements are numbered O_1, O_2 , etc. $S(X)$ represents a set of source locations of a statement X . Each source location is a tuple of the form $\langle \text{filename}, x, y \rangle$ that represents the location of the row (y) and column (x) number in the source file. We show below how we can maintain the set of source locations for an optimized instruction for commonly applied optimizations. We use optimizations as described in [135].

1. **Elimination of Global Common Subexpression:** duplicate expressions are eliminated so that the expression is computed once [135].

$U1: x = a+b;$

$O1: u = a+b;$

$U2: y = a+b;$

$O2: x = u;$

$O3: y = u;$

$S(O_1) = S(U_1) \cup S(U_2)$

$S(O_2) = S(U_1)$

$S(O_3) = S(U_2)$

2. **Copy Propagation:** replace the copy of a variable with the original variable [2][135].

$U1: x = a;$

$O1: x = a;$

$U2: y = x + 5;$

$O2: y = a + 5;$

$S(O_1) = S(U_1)$

$S(O_2) = S(U_2)$

3. **Dead Code Elimination:** remove statements that define values for variables that are not used [135].

$U1: x = a;$

Since there is no optimized code (O_1), assigning source locations does not apply here.

4. Constant Folding: replace mathematical expressions involving constants with their equivalent values [135].

U1: $x = 3 + 5;$

O1: $x = 8;$

$S(O_1) = S(U_1)$

5. Loop Circulation: interchange more than two perfectly nested loops [135].

U1: for ($i=0; i < N; i++$)

O1: for ($k=0; k < P; k++$)

U2: for ($j=0; j < M; j++$)

O2: for ($j=0; j < M; j++$)

U3: for ($k=0; k < P; k++$)

O3: for ($i=0; i < N; i++$)

U4: $A[i][j][k] = c;$

O4: $A[i][j][k] = c;$

$S(O_1) = S(U_3)$

$S(O_2) = S(U_2)$

$S(O_3) = S(U_1)$

$S(O_4) = S(U_4)$

6. Loop Invariant Code Motion: remove statements from within loops where computed values do not change within the loop [135].

U1: for ($i=0; i < N; i++$)

O1: $x = c;$

U2: {

O2: for ($i = 0; i < N; i++$)

U3: $x = c;$

O3: {

U4: $b[i] = a + i + 3;$

O4: $b[i] = a + i + 3;$

U5: }

O5: }

$$S(O_1) = S(U_3)$$

$$S(O_2) = S(U_1)$$

$$S(O_4) = S(U_4)$$

7. Loop Fusion: combine loops with the same headers [135].

U1: for (i=0; i<N;i++)

O1: for (i = 0; i < N; i++)

U2: {

O2: {

U3: a[i] = i + 2;

O3: a[i] = i + 2;

U4: }

O4: b[i] = 3 * i + 2;

U5: for (i = 0; i < N; i++)

O5: }

U6: {

U7: b[i] = 3 * i + 2;

U8: }

$$S(O_1) = S(U_1) \cup S(U_5)$$

$$S(O_3) = S(U_3)$$

$$S(O_4) = S(U_7)$$

8. Loop Unrolling: duplicate the body of a loop [135].

U1: for (i=0; i<100; i++)

O1: a[0] = 54;

U2: {

O2: for (i = 1; i < 100; i++)

U3: a[i] = i + 54;

O3: {

U4: }

O4: a[i] = i + 54;

O5: }

$$S(O_1) = S(U_3)$$

$$S(O_2) = S(U_1)$$

$$S(O_4) = S(U_3)$$

This example illustrates how we can handle a one-to-many mapping where one statement of an unoptimized program maps to more than one statement in the optimized code.

9. Strip Mining: modify loop to utilize hierarchical memory or vector architecture to reduce cache misses [138][135].

U1: for (i=0; i<N;i++)

U2: {

U3: a[i] = i * 3;

U4: b[i] = a[i] + 5;

U5: }

O1: for (j = 0; j < N; j = j+SZ)

O2: for (i = j, i < MIN(N, J+SZ-1); i++)

O3: {

O4: a[i] = i * 3;

O5: b[i] = a[i] + 5;

O6: }

$$S(O_1) = S(U_1)$$

$$S(O_2) = S(U_1)$$

$$S(O_4) = S(U_3)$$

$$S(O_5) = S(U_4)$$

10. Loop Unswitching: modify a loop that contains an if statement to an if statement that contains a loop [135].

U1: for (i=0; i<N;i++)

U2: {

U3: if (k > 24)

O1: if (k > 24)

O2: {

O3: for (i = 0; i < N; i++)

U4: {	O4: {
U5: a[i] = 2 * i + 3;	O5: a[i] = 2 * i + 3;
U6: }	O6: }
U7: else	O7: }
U8: {	O8: else
U9: a[i] = 2*i - 4;	O9: {
U10: }	O10: for (i = 0; i < N; i++)
U11: }	O11: {
	O12: a[i] = 2*i - 4;
	O13: }

$S(O_1) = S(U_3)$

$S(O_8) = S(U_7)$

$S(O_5) = S(U_5)$

$S(O_{12}) = S(U_9)$

$S(O_3) = S(U_1)$

$S(O_{10}) = S(U_1)$

11. Bumping: modify the loop iterations by bumping the index by a preset value [135].

U1: for (i=-2; i < 98; i++)	O1: for (i = 0; i < 100; i++)
U2: {	O2: {
U3: a[i] = i + 54;	O3: a[i] = i + 54 - 2;
U4: }	O4: }
$S(O_1) = S(U_1)$	

BIBLIOGRAPHY

- [1] V.S. Adve, J. Mellor-Crummey, M. Anderson, K. Kennedy, J. C. Wang, and D. A. Reed, "Integrating Compilation and Performance Analysis for Data-Parallel Programs," Proceedings of the Workshop on Debugging and Performance Tuning for Parallel Computing Systems, IEEE Computer Society Press, (M. L. Simmons, A. H. Hayes, D. A. Reed, and J. Brown, Eds.), January 1996.

- [2] A. Aho, R. Sethi, and J. Ullman, *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1985.

- [3] J. M. Anderson, L. M. Berc, J. Dean, S. Ghemawat, M. R. Henzinger, S. A. Leung, R. L. Sites, M. T. Vandevoorde, C. A. Waldspurger, and W. E. Weihl, "Continuous Profiling: Where Have All the Cycles Gone?," Proceedings 16th ACM Symposium on Operating Systems Principles, October 5-8, 1997, St. Malo, France.

- [4] K. Arnold, and J. Gosling, *The Java Programming Language*, Addison-Wesley, 1996.

- [5] W. Arnold and R. Haydock, "A Parallel Object-Oriented Implementation of the Dynamic Recursion Method," Proceedings ISCOPE '98 Conference, LNCS Series, Springer-Verlag, 1998.

- [6] D. Bailey, E. Barszcz, L. Dagum, and H. Simon, "NAS parallel benchmark results," Technical Report RNR-93016, NASA Ames Res. Center, Moffet Field, 1993.

- [7] M. Baker, B. Carpenter, G. Fox, S. Ko, and S. Lim, "mpiJava: An Object-Oriented Java Interface to MPI," Proceedings of International Workshop on Java for Parallel and Distributed Computing, IPPS/SPDP 1999, April 1999.

- [8] M. Baker and B. Carpenter, "Thoughts on the structure of an MPJ reference implementation," URL:<http://www.npac.syr.edu/projects/pcrc/HPJava/mpiJava.html>.
- [9] A. Bakic, M. Mutka, and D. Rover, "BRISK: A Portable and Flexible Distributed Instrumentation System," Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 148, ACM, Aug 1998.
- [10] T. Ball and J. Larus, "Optimally Profiling and Tracing Programs," *ACM Transactions on Programming Languages and Systems*, 16(4) pp. 1319-1360, July 1994.
- [11] F. Bassetti, K. Davis, and D. Quinlan, "A Comparison of Performance-Enhancing Strategies for Parallel Numerical Object-Oriented Frameworks," Proceedings ISCOPE '97 Conference, LNCS Vol. 1343, Springer-Verlag, December 1997.
- [12] F. Bassetti, D. Brown, K. Davis, W. Henshaw, and D. Quinlan, "OVERTURE: An Object-oriented Framework for High Performance Scientific Computing," Proceedings of SC '98: High Performance Networking and Computing. IEEE Computer Society, 1998.
- [13] P. Beckman and D. Gannon, "Tulip: A Portable Run-Time System for Object Parallel Systems," Proceedings of the 10th International Parallel Processing Symposium, August 1996.
- [14] R. Berrendorf and H. Ziegler, "PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors," Technical Report FZJ-ZAM-IB-9816, Forschungszentrum Jülich, GmbH, Germany, October 1998.
- [15] F. Bodin, P. Beckman, D. Gannon, J. Gotwals, S. Narayana, S. Srinivas, and B. Winnicka, "Sage++: an Object-oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools," In Proceedings of Second Annual Object-Oriented Numerics Conference, 1994.
- [16] F. Bodin, P. Beckman, D. Gannon, S. Yang, S. Kesavan, A. Malony, and B. Mohr, "Implementing a Parallel C++ Runtime System for Scalable Parallel Systems," Proceedings of Supercomputing'93, pp. 588-597, November 1993.

- [17] R. Boisvert, J. Moreira, M. Philippsen, and R. Pozo, "Java and Numerical Computing," *Computing in Science and Engineering*, 3(2) pp. 18-24, 2001.
- [18] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci, "A Portable Programming Interface for Performance Evaluation on Modern Processors," *The International Journal of High Performance Computing Applications*, 14:3, pp. 189-204, 2000.
- [19] S. Browne, J. Dongarra, and K. London, "Review of Performance Analysis Tools for MPI Parallel Programs," URL:<http://www.cs.utk.edu/~browne/perftools-review/>.
- [20] B. Buck and J. Hollingsworth, "An API for Runtime Code Patching," *Journal of High Performance Computing Applications*, 14(4):317-329, Winter 2000.
- [21] B. Chamberlain, S. Choi, E. Lewis, C. Lin, L. Snyder, W. Weathersby, "Factor-Join: A Unique Approach to Compiling Array Languages for Parallel Machines," Workshop on Languages and Compilers for Parallel Computing, Aug. 1996.
- [22] B. Chamberlain, S. Choi, L. Snyder, "A Compiler Abstraction for Machine Independent Parallel Communication Generation," Proceedings of Workshop on Languages and Compilers for Parallel Computing, Aug. 1997.
- [23] K. Chandy and C. Kesselman, "CC++: A Declarative Concurrent Object-Oriented Programming Notation," In G. Agha, P. Wegner, A. Yonezawa, (editors), *Research Directions in Concurrent Object Oriented Programming*, MIT press, 1993.
- [24] C. Cl  men  on, J. Fritscher, M. Meehan, and Roland R  hl, "An Implementation of Race Detection and Deterministic Replay with MPI," Technical Report CSCS TR-95-01, Centro Svizzero di Calcolo Scientifico, Switzerland, 1995.
- [25] B. Cooper, H. Lee, and B. Zorn, "ProfBuilder: A Package for Rapidly Building Java Execution Profilers," University of Colorado, Boulder, Technical Report CU-CS-853-98, April 1998.

- [26] J. Cummings, J. Crotinger, S. Haney, W. Humphrey, S. Karmesin, J. Reynders, S. Smith, T. Williams, "Rapid Application Development and Enhanced Code Interoperability using the POOMA Framework," Proceedings of OO'98: SIAM Workshop on Object-Oriented Methods and Code Inter-operability in Scientific and Engineering Computing, 1998.
- [27] K. Czarnecki, U. Eisenecker, R. Glück, D. Vandevoorde, T. Veldhuizen, "Generative Programming and Active Libraries," in Proceedings of the 1998 Dagstuhl-Seminar on Generic Programming, LNCS, Springer-Verlag, 1998.
- [28] L. DeRose, Y. Zhang, and D. Reed, "SvPablo: A Multi-Language Performance Analysis System." in Proceedings of 10th International Conference on Computer Performance Evaluation - Modelling Techniques and Tools - Performance Tools '98, pp. 352-355, Sept. 1998.
- [29] T. Fahringer and H. Zima, "A Static Parameter based Performance Prediction Tool for Parallel Programs," Proceedings of the 7 th ACM International Conference on Supercomputing, July 1993.
- [30] T. Fahringer, "Estimating Cache Performance for Sequential and Data Parallel Programs," Proceedings of HPCN'97, LNCS, Springer-Verlag, April 1997.
- [31] C. Fraser, and D. Hanson, "A Retargetable C Compiler: Design and Implementation," Benjamin/Cummings Publishing Company, Redwood City, CA, 1995.
- [32] J. Gait, "A Probe Effect in Concurrent Programs," *Software Practice and Experience*, 16(3), March 1986.
- [33] J. Galarowicz, B. Mohr, "Analyzing Message Passing Programs on the CRAY T3E with PAT and VAMPIR," Technical Report TR IB-9809, ZAM Forschungszentrum Juelich, Germany, May 1998.
- [34] G. Geist, M. Heath, B. Peyton, and P. Worley. "A Users' Guide to PICL: A portable instrumented communication library," Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, Oak Ridge, TN, Sept. 1990.

- [35] J. Germain, J. McCorquodale, S. Parker, C. Johnson, "Uintah: A Massively Parallel Problem Solving Environment," Proceedings HPDC'00: Ninth IEEE International Symposium on High Performance and Distributed Computing, August 2000.
- [36] GNU Project, "Using and Porting the GNU Compiler Collection (GCC): Invoking GCC," URL:http://gcc.gnu.org/onlinedocs/gcc_3.html 2001.
- [37] S. Graham, P. Kessler, and M. McKusick, "gprof: a Call Graph Execution Profiler," SIGPLAN '82 Symposium on Compiler Construction, *SIGPLAN Notices*, 17(6), June 1982.
- [38] S. Graham, P. Kessler, and M. McKusick, "An Execution Profiler for Modular Programs," *Software Practice and Experience*, 13, pp. 671-685, 1983.
- [39] B. Haake, K. E. Schauer, C. J. Scheiman, "Profiling a Parallel Language Based on Fine-Grained Communication," Proceedings of Supercomputing '96, 1996.
- [40] R. Hall, "Call Path Refinement Profiles," *IEEE Transactions on Software Engineering*, 21(6), June 1995.
- [41] S. Haney, J. Crotinger, S. Karmesin, S. Smith, "Easy Expression Templates Using PETE, the Portable Expression Template Engine," Los Alamos National Laboratory Technical Report LA-UR-99-777, Los Alamos, NM, 1999.
- [42] M. Heath and J. Etheridge. "Visualizing the Performance of Parallel Programs", *IEEE Software*, 8(5), 1991.
- [43] J. Hollingsworth, "Finding Bottlenecks in Large Scale Parallel Programs," Ph.D. Thesis, University of Wisconsin, Madison, 1994.
- [44] J. Hollingsworth, and B. Buck, "Dyninst API Programmer's Guide," Computer Science Department, University of Maryland. URL:<http://www.cs.umd.edu/projects/dyninstAPI> 2001.

- [45] J. Hollingsworth, and B. Miller, "Instrumentation and Measurement," in I. Foster, and C. Kesselman (editors), *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers Inc, San Francisco, 1999, pp. 339-365.
- [46] HPC++ Working Group, "HPC++ White Papers", Technical Report TR 95633, Center for Research on Parallel Computation, 1995.
- [47] Intel Corporation, "Intel VTuneTM Performance Analyzer," URL: <http://developer.intel.com/software/products/vtune/>, 2001.
- [48] R. B. Irvin, and B. P. Miller, "Mapping Performance Data for High-Level and Data Views of Parallel Program Performance," Proceedings of International Conference on Supercomputing, May 1996.
- [49] R. B. Irvin, "Performance Measurement Tools for High-Level Parallel Programming Languages," Ph.D. Thesis, University of Wisconsin, Madison, 1995.
- [50] C. Jaramillo, R. Gupta, and M.L. Soffa, "FULLDOC: A Full Reporting Debugger for Optimized Code," in Proceedings of 7th International Static Analysis Symposium, LNCS 1824, Springer-Verlag, pp. 240-259, June/July 2000.
- [51] C. Jaramillo, R. Gupta, and M.L. Soffa, "Comparison Checking: An Approach to Avoid Debugging of Optimized Code," ACM SIGSOFT Symposium on Foundations of Software Engineering and European Software Engineering Conference, LNCS 1687, Springer Verlag, pp. 268-284, Sept. 1999.
- [52] C. Jaramillo, R. Gupta, and M.L. Soffa, "Capturing the Effects of Code Improving Transformations," in Proceedings of International Conference on Parallel Architectures and Compilation Techniques, pp. 118-123, Oct. 1998.
- [53] C. Jaramillo, "Source Level Debugging Techniques and Tools for Optimized Code," Ph.D. Dissertation, University of Pittsburgh, 2000.

- [54] C. Johnson, S. Parker, and D. Weinstein, "Large-Scale Computational Science Applications Using the SCIRun Problem Solving Environment," Proceedings of Supercomputer 2000.
- [55] G. Judd, "Design Issues for Efficient Implementation of MPI in Java," Proceedings of ACM JavaGrande Conference, pp. 37-46, 1999.
- [56] K. Karavanic, "Experiment Management Support for Parallel Performance Tuning," Ph.D. Thesis, University of Wisconsin, Madison, Dec. 1999.
- [57] K. Keahey, P. Beckman, and J. Ahrens, "Component Architecture for HighPerformance Applications," Proceedings of the First NASA Workshop on Performance-Engineered Information Systems, September, 1998.
- [58] P. Kessler, "Fast breakpoints: Design and implementation," Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation, Appeared as *SIGPLAN Notices* 25(6), pp. 78-84, June 1990.
- [59] J. Kohn, and W. Williams, "ATExpert," *Journal of Parallel and Distributed Computing*, 18, pp. 205-222, 1993.
- [60] J. Kundu, "Integrating Event- and State-based approaches to the Debugging of Parallel Programs," Ph.D. Thesis, University of Massachusetts, Amherst, 1996.
- [61] L. Lamport, "Time, Clocks and the Ordering of Events in a Distributed System," *Communications of the ACM*, 21, pp. 558-565, July 1978.
- [62] F. Lange, R. Kroeger, M. Gergeleit, "JEWEL: Design and Implementation of a Distributed Measurement System," *IEEE Transactions on Parallel and Distributed Systems*, 3(6), pp. 657-671, November 1992.

- [63] J. Larus, and E. Schnarr, "EEL: Machine-Independent Executable Editing," Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 1995
- [64] J. Larus, T. Ball, "Rewriting Executable Files to Measure Program Behavior," *Software Practice and Experience*, 24(2) pp. 197-218. February 1994.
- [65] T. LeBlanc, and J. Mellor-Crummey, "Debugging Parallel Programs with Instant Replay," *IEEE Transactions on Computers*, Vol. C-36(4), pp. 471-482, April 1987.
- [66] S. Liao, "SUIF Explorer: an Interactive and Interprocedural Parallelizer," Ph.D. Thesis, Stanford University Technical Report CSL-TR-00-807, August 2000.
- [67] C. Lin and L. Snyder, "A portable implementation of SIMPLE," *International Journal of Parallel Programming*, 20(5), pp. 363-401, 1991.
- [68] C. Lin and L. Snyder, "ZPL: An Array Sublanguage," in U. Banerjee, D. Gelernter, A. Nicolau and D. Padua, editors, *Languages and Compilers for Parallel Computing*, pp. 96-114, 1993.
- [69] K. A. Lindlan, J. Cuny, A. D. Malony, S. Shende, B. Mohr, R. Rivenburgh, C. Rasmussen. "A Tool Framework for Static and Dynamic Analysis of Object-Oriented Software with Templates." Proceedings of SC2000: High Performance Networking and Computing Conference, November 2000.
- [70] T. Madhyastha, and D. Reed, "Data Sonification: Do You See What I Hear?" *IEEE Software*, 12(2) pp. 45-56, March 1995.
- [71] A. Malony, "Program Tracing in Cedar," University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, Technical Report CS-660, 1987.
- [72] A. Malony, "Performance Observability," Ph.D. Thesis, University of Illinois, Urbana Champaign, 1990 (Also Available as CSRD Report No. 1034 UILU-ENG-90-8030) Sept. 1990.

- [73] A. Malony, "Tools for Parallel Computing: A Performance Evaluation Perspective," in J. Bazewicz et al. (Editors), *Handbook on Parallel and Distributed Processing*, Springer Verlag, pp. 342-363, 2000.
- [74] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, F. Bodin, "Performance Analysis of pC++: A Portable Data-Parallel Programming System for Scalable Parallel Computers," Proceedings of the 8th International Parallel Processing Symposium (IPPS), pp. 75-85, April 1994.
- [75] A. Malony and S. Shende, "Performance Technology for Complex Parallel and Distributed Systems," in P. Kacsuk and G. Kotsis (editors), *Distributed and Parallel Systems: From Instruction Parallelism to Cluster Computing*, Kluwer, Norwell, MA, pp. 37-46, 2000.
- [76] S. McLaughry, "Debugging Optimized Parallel Programs," Directed Research Project Report, Department of Computer and Information Science, University of Oregon, 1997. URL: <http://www.cs.uoregon.edu/research/paracomp/publ>.
- [77] C. Mendes, "Performance Scalability Prediction on Multicomputers," Ph.D. Dissertation, University of Illinois at Urbana-Champaign, 1997.
- [78] Message Passing Interface Forum, "MPI: A Message Passing Interface Standard," *International Journal of Supercomputer Applications* (Special Issue on MPI), 8(3/4), 1994. URL: <http://www.mcs.anl.gov>.
- [79] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K. Kunchithapadam, and T. Newhall, "The Paradyn Parallel Performance Measurement Tools", *IEEE Computer* 28(11), Special issue on Performance Evaluation Tools for Parallel and Distributed Computer Systems, Nov. 1995.
- [80] B. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S. Lim, and T. Torzewski "IPS-2: The Second Generation of a Parallel Program Measurement System," *IEEE Transactions on Parallel and Distributed Systems*, 1(2) pp. 206 - 217, April 1990.

- [81] B. Mohr, A. Malony, and J. Cuny, "TAU," in G.V. Wilson and P. Lu (editors), *Parallel Programming using C++*, MIT Press, 1996.
- [82] W. Nagel, A. Arnold, M. Weber, H. C. Hoppe, and K. Solchenbach, "VAMPIR: Visualization and Analysis of MPI Resources," *Supercomputer*, 12(1) pp. 69-80, 1996.
- [83] R. Netzer, "Trace Size vs. Parallelism in Trace-and-Replay Debugging of Shared-Memory Programs," *Languages and Compilers for Parallel Computing*, LNCS, Springer-Verlag, 1993.
- [84] R. Netzer, "Race Condition Detection for Debugging Shared-Memory Parallel Programs," Ph.D. Thesis, University of Wisconsin, Madison, 1991.
- [85] R. Netzer, "Optimal Tracing and Replay for Debugging Shared-Memory Parallel Programs," in Proceedings of ACM/ONR Workshop on Parallel and Distributed Debugging, May 1993.
- [86] R. Netzer, "Optimal Tracing and Replay for Debugging Message-Passing Parallel Programs," *The Journal of Supercomputing*, 8(4) 371-388. 1994.
- [87] R. Netzer, S. Subramanian, J. Xu, "Critical-Path-Based Message Logging for Incremental Replay of Message-Passing Programs," Proceedings of International Conference on Distributed Computing Systems, Poland, June 1994.
- [88] R. Netzer, J. Xu, "Adaptive Message Logging for Incremental Program Replay," *IEEE Parallel and Distributed Technology*, Nov. 1993.
- [89] T. Newhall, "Performance Measurement of Interpreted, Just-in-Time compiled, and Dynamically Compiled Executions," Ph.D. Dissertation, University of Wisconsin, Madison, Aug. 1999.
- [90] T. Newhall and B. Miller, "Performance Measurement of Interpreted Programs," Proceedings of Europar'98, 1998.

- [91] Oak Ridge National Laboratory, "Portable Instrumented Communication Library," 1998 URL:<http://www.epm.ornl.gov/picl>.
- [92] D. Ogle, K. Schwan, and R. Snodgras, "Application-Dependent Dynamic Monitoring of Distributed and Parallel Systems," *IEEE Transactions on Parallel and Distributed Systems*, 4(7), pp. 762-778, July 1993.
- [93] D. Padua and M. Wolfe, "Advanced Compiler Optimizations for Supercomputers," *Communications of the ACM*, 29(12) pp. 1184-1201, Dec. 1986.
- [94] Pallas GmbH, "Pallas Products: Vampir," URL:<http://www.pallas.com/pages/vampir.htm> 2001.
- [95] Pallas GmbH, "Pallas - High Performance Computing - Products: VampirTrace," URL:<http://www.pallas.com/pages/vampirt.htm> 2001.
- [96] D. Pase, "MPP Apprentice: A Non-Event Trace Performance Tool for the Cray T3D," Proceedings of Workshop on Debugging and Performance Tuning for Parallel Computing Systems, Oct. 1994.
- [97] Rational Software, "Rational Purify, Rational Quantify, Rational PureCoverage," URL:<http://www.rational.com/products/pqc/index.jsp>.
- [98] D. Reed, "Performance Instrumentation Techniques for Parallel Systems," In L. Donatiello and R. Nelson (editors), *Models and Techniques for Performance Evaluation of Computer and Communications Systems*, Springer-Verlag Lecture Notes in Computer Science, pp. 463-490, 1993.
- [99] D. Reed, R. Ayd, R. Noe, P. Roth, K. Shields, B. Schwarta, and L. Tavera, "An overview of the Pablo performance analysis environment," in Proceedings of the Scalable Parallel Libraries Conference, pp. 104-113, Oct 1994.
- [100] D. Reed, and R. Ribler, "Performance Analysis and Visualization," in I. Foster, and C. Kesselman (editors), *The Grid: Blueprint for a New Computing Infrastructure*, Morgan Kaufmann Publishers Inc, San Francisco, pp. 367-393, 1999.

- [101] J. Reynders, P. Hinker, S. Atlas, S. Banerjee, W. Humphrey, S. Karmesin, K. Keahey, M. Srikant, and M. Tholburn, "Pooma: A Framework for Scientific Simulation on Parallel Architectures," in G.V. Wilson and P. Lu (editors), *Parallel Programming using C++*, pp. 553-594, MIT Press, 1996.
- [102] R. Ribler, J. Vetter, H. Simitci, and D. Reed, "Autopilot: Adaptive Control of Distributed Applications," Proceedings of the 7th IEEE Symposium on High-Performance Distributed Computing, July 1998.
- [103] A. Robison, "C++ gets faster for scientific computing," *Computers in Physics*, 10(5) pp. 458-462, Sept./Oct. 1996.
- [104] S. Sarukkai and A. Malony. "Perturbation Analysis of High Level Instrumentation for SPMD Programs," *SIGPLAN Notices*, 28(7), 1993.
- [105] A. Serra, N. Navarro, "Extending the Execution Environment with DITools," Universitat Politècnica de Catalunya Technical Report UPC-DAC-1999-26, 1999.
- [106] K. Shanmugam, "Performance Extrapolation of Parallel Programs," Master's Thesis. Department of Computer and Information Science, University of Oregon, June 1994.
- [107] T. Sheehan, A. Malony, S. Shende, "A Runtime Monitoring Framework for the TAU Profiling System," in S. Matsuoka, R. Oldehoeft, and M. Tholburn (editors), *Computing in Object-Oriented Parallel Environments*, Third International Symposium ISCOPE'99, Lecture Notes in Computer Science, No. 1732, Springer-Verlag. pp. 170-181, Dec. 1999.
- [108] S. Shende, "Profiling and Tracing in Linux," Proceedings of Extreme Linux Workshop #2, USENIX Annual Technical Conference, June 1999.
- [109] S. Shende, J. Cuny, L. Hansen, J. Kundu, S. McLaughry, O. Wolf, "Event and State Based Debugging in TAU: A Prototype," Proceedings of the 1996 ACM SIGMETRICS Symposium on Parallel and Distributed Tools, May 1996, pp. 21-30.

- [110] S. Shende and A. D. Malony, "Integration and Application of the TAU Performance System in Parallel Java Environments," Proceedings of the Joint ACM Java Grande - ISCOPE 2001 Conference, pp. 87-96, June 2001.
- [111] S. Shende, A. D. Malony, and R. Ansell-Bell, "Instrumentation and Measurement Strategies for Flexible and Portable Empirical Performance Evaluation," Proceedings International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001), CSREA, June 2001.
- [112] S. Shende, A. Malony, and S. Hackstadt, "Dynamic Performance Callstack Sampling: Merging TAU and DAQV," in B. Kågström, J. Dongarra, E. Elmroth and J. Wasniewski (editors) *Applied Parallel Computing. Large Scale Scientific and Industrial Problems*, 4th International Workshop, PARA'98, Lecture Notes in Computer Science, No. 1541, Springer-Verlag, 1998.
- [113] S. Shende, A. D. Malony, J. Cuny, K. Lindlan, P. Beckman, S. Karmesin, "Portable Profiling and Tracing for Parallel Scientific Applications using C++", Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools, pp. 134-145, ACM, Aug 1998.
- [114] Silicon Graphics, Inc., "Speed Shop User's Guide," URL:<http://techpubs.sgi.com>.
- [115] SPARC International Inc., D. Weaver, and T. Germond (Editors), "The SPARC Architecture Manual," Version 9, Prentice Hall, Englewood Cliffs, NJ, 2000.
- [116] A. Srivastava, A. Eustace, "ATOM: A System for Building Customized Program Analysis Tools," Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), pp. 196-205, June 1994.
- [117] A. Srivastava, D. Wall, "A Practical System for Intermodule Code Optimization and Link-time," *Journal of Programming Languages*, 1(1) pp. 1-18, March 1993.
- [118] B. Stroustrup, *The C++ Programming Language*, Third Edition, Addison-Wesley, Reading, Massachusetts, June 1997.

- [119] Sun Microsystems, "Forte Tools," URL:<http://www.sun.com/forte/>.
- [120] Sun Microsystems, "Java Native Interface," URL:<http://java.sun.com/j2se/1.3/docs/guide/jni/>.
- [121] Sun Microsystems, "Java Virtual Machine Profiler Interface (JVMPI)," URL: <http://java.sun.com/products/jdk/1.3/docs/guide/jvmpi/jvmpi.html>.
- [122] Sun Microsystems Inc. "The JAVA HotSpot Performance Engine Architecture," Sun Microsystems White Paper, April 1999. URL:<http://java.sun.com/products/hotspot/whitepaper.html>.
- [123] V. Taylor, M. Huang, T. Canfield, R. Stevens, S. Lamm, and D. Reed, "Performance Monitoring of Interactive, Immersive Virtual Environments for Finite Element Simulations," *Journal of Supercomputing Applications and High-Performance Computing*, 10(2/3) pp. 145-156 (Summer/Fall 1996, special issue I-WAY: Wide Area Supercomputer Applications), 1996.
- [124] University of Oregon, "TAU User's Guide" URL: <http://www.cs.uoregon.edu/research/paracomp/tau/> 2001.
- [125] S. Vajracharya, "Runtime Loop Optimizations for Locality and Parallelism," Ph.D. Thesis, University of Colorado, Boulder, 1997.
- [126] S. Vajracharya, S. Karmesin, P. Beckman, J. Crotinger, A. Malony, S. Shende, R. Oldehoeft, S. Smith, "SMARTS: Exploiting Temporal Locality and Parallelism through Vertical Execution," Los Alamos National Laboratory Technical Report LA-UR-99-16, Los Alamos, NM, 1999 (also appears in Proceedings of 1999 International Conference on Supercomputing, ACM, pp.302-310,1999).
- [127] T. Veldhuizen, "Expression Templates," *C++ Report*, 7(5) pp. 26-31, June 1995.
- [128] T. Veldhuizen, "Using C++ Template Metaprograms," *C++ Report*, 7(4) pp. 36-43, May 1995.

- [129] T. Veldhuizen, and D. Gannon, "Active Libraries: Rethinking the roles of compilers and libraries," Proceedings of OO'98: SIAM Workshop on Object-Oriented Methods and Code Inter-operability in Scientific and Engineering Computing, 1998.
- [130] T. Veldhuizen and M. E. Jernigan, "Will C++ be faster than Fortran," Proceedings ISCOPE 97, LNCS Vol. 1343, Springer, pp. 49-56, December 1997.
- [131] J. Vetter, and D. Reed, "Real-time Performance Monitoring, Adaptive Control, and Interactive Steering of Computational Grids," *The International Journal of High Performance Computing Applications*, 14(4), pp. 357-366, Winter 2000.
- [132] D. Viswanathan and S. Liang, "Java Virtual Machine Profiler Interface," *IBM Systems Journal*, 39(1) pp.82-95, 2000.
- [133] VMGEAR, "VMGEAR - Tools for Java Performance: Optimize It," URL:<http://www.vmgear.com/>.
- [134] D. Wall, "Systems for late code modification," in R. Giegerich and S. Graham (editors), *Code Generation – Concepts, Tools, Techniques*, pp. 275-293, Springer-Verlag, 1992.
- [135] D. Whitfield, and M. Soffa, "An Approach for Exploring Code Improving Transformations," *ACM Transactions on Programming Languages*, 19(6) pp. 1053-1084, Nov. 1997.
- [136] R. Wilson, R. French, C. Wilson. S. Amarasinghe, J. Anderson, S. Tjiang, S. Liao, C. Tseng, M. Hall, M. Lam, and J. Hennessy, "SUIF: An Infrastructure for Research on Parallelising and Optimizing Compilers," *ACM SIGPLAN Notices*, 29(12) pp. 31-37, Dec. 1994.
- [137] F. Wolf, and B. Mohr, "EARL - A Programmable and Extensible Toolkit for Analyzing Event Traces of Message Passing Programs," Technical Report FZJ-ZAM-IB-9803, Forschungszentrum Jülich GmbH, Germany, April 1998.

- [138] M. Wolfe, "High Performance Compilers for Parallel Computing," Addison-Wesley, Redwood City, CA, 1996.
- [139] J. Yan, "Performance Tuning with AIMS—An Automated Instrumentation and Monitoring System for Multicomputers," Proceedings of the 27th Hawaii International Conference on System Sciences, Jan. 1994.
- [140] C. Yan, and S. R. Sarukkai "Analyzing Parallel Program Performance Using Normalized Performance Indices and Trace Transformation Techniques", *Parallel Computing*, 22(9), pp 1215-1237, November 1996.
- [141] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, "Performance Analysis Using the MIPS R10000 Performance Counters," Proceedings Supercomputing '96, IEEE Computer Society, November 1996.
- [142] F. Zambonelli, R. B. Netzer, "An Efficient Logging Algorithm for Incremental Replay of Message-Passing Applications," in Proceedings of the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing (IPPS/SPDP), 1999.
- [143] H. Zima, P. Brezany, B. Chapman, P. Mehrotra, A. Schwald, "Vienna Fortran – A Language Specification, Version 1.1," Technical Report Series ACPC/TR 92-4, Austrian Center for Parallel Computation, University of Vienna, Austria, 1992.