INSTRUMENTATION DATABASE APPROACH TO THE ANALYSIS OF LARGE PARALLEL AND OBJECT-ORIENTED SCIENTIFIC APPLICATIONS

By

Jeffrey Nesheiwat

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

Approved by the
Examining Committee:

Dr. Boleslaw K. Szymanski, Thesis Adviser

Dr. Joseph E. Flaherty, Member

Dr. Franklin T. Luk, Member

Dr. David L. Spooner, Member

Dr. Charles D. Norton, Member

Rensselaer Polytechnic Institute
Troy, New York

October 2000
(For Graduation December 2000)

INSTRUMENTATION DATABASE APPROACH TO THE ANALYSIS OF LARGE PARALLEL AND OBJECT-ORIENTED SCIENTIFIC APPLICATIONS

By

Jeffrey Nesheiwat

An Abstract of a Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: Computer Science

The original of the complete thesis is on file
in the Rensselaer Polytechnic Institute Library

Examining Committee:

Dr. Boleslaw K. Szymanski, Thesis Adviser

Dr. Joseph E. Flaherty, Member

Dr. Franklin T. Luk, Member

Dr. David L. Spooner, Member

Dr. Charles D. Norton, Member

Rensselaer Polytechnic Institute
Troy, New York

October 2000
(For Graduation December 2000)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

The first person I would like to thank is Dr. Boleslaw K. Szymanski, my thesis advisor. I consider myself fortunate for the honor of working with him and I look forward to continued collaboration in the future. His formidable skills as a researcher and as a mentor will serve as an example to me throughout my career. I would also like to thank the members of my thesis committee, Joseph Flaherty, Frank Luk, David Spooner, and Charles Norton. They have given generously of their time in support of this work.

Many people have contributed to this research. I am very grateful to Hui Wang for her Master's project work in support of developing our first automated instrumentation tool. My thanks also go to two very talented students whose independent study projects made this research what it is today: Jonathan Chen's work on automated instrumentation and Simon Karpen's work on front-end visualization tools took IDB to a new level. They both have my respect and gratitude.

There are many people at the Jet Propulsion Laboratory that I would like to thank. First and foremost, I would like to thank Earnest Stone of the RF and Microwave Subsystem Group in the Deep Space Network for bringing me to JPL as a co-op in 1991. I learned a great deal in the four summers I worked on the Pioneer 10 and Block V receivers. That opportunity marked the beginning of a long and productive relationship with the lab that ultimately led me to the High Performance Computing Systems and Applications Group. I would like to thank Linda Rogers and Bill Whitney for their support along the way. Much of this research was guided by feedback from scientists at JPL. I would like to take this opportunity to thank Gerhard Klimeck for his input at various stages of IDB's development and for his help using NEMO to validate our approach. Similarly I would like to thank Dan Katz and Bill Gustafson for making OTIS available to us.

As important as the people who do the research are those that make it possible. My sincerest thanks to all those who have supported this work financially. This includes Carol Hix from the NASA GSRP and Tom Cwik and the High Performance

Computing Systems and Applications Group who sponsored this work and in turn made available the funds, resources and people needed to make IDB a reality. I would also like to extend a heartfelt thanks to Pat Valiquette, the director of Helpdesk Services, my thesis advisor, Ephraim Glinert, and Diane Lipins for their funding over the past six years.

Nothing would be possible without the system administrators who keep everything running smoothly. I would like to thank the members of *labstaff* here in the Computer Science Department at RPI for their help over the years: Nathan Schimke, Amitha Perera, Chris Parker, James Kilbride, and, most of all, David Cross. David has often given generously of his time in support of this work. I would be remiss if I did not thank Nooshin Meshkaty for her diligent work in keeping the multiple Beowulf Clusters at JPL running as well as the many people here at CIS. These people create the virtual environment that is conducive to research and maintain it every day; I am very grateful for their help. I would also like to thank the people that create a real environment conducive to research. I am fortunate to have the opportunity to work with the very dedicated administrative staff in the Computer Science Department. I thank Terry Hayden and Chris Coonrad for their assistance along the way. I would especially like to thank Pam Paslow. To me, Pam is the person to talk to about a wide range of issues from payroll to lunch. I am very grateful for her help and for her friendship.

Analysis of SCOREC's adaptive PDE codes was a crucial element to validating this work. I would like to extend my sincerest thanks and heartfelt gratitude to Jim Teresco, who donated many hours of his time to help me construct meaningful problems on which to test my work. I have a great deal of respect for Jim's technical expertise and patience. I would also like to thank Ray Loy, Wesley Turner, Louis Ziantz, and Toshiro Ohsumi for answering many questions about SCOREC software. I would like to thank my long time friend Danny Daglas for his moral support and technical insight along the way, Varina Hammond and Deb Wentorf for proof-reading help and, Ron Frederick for his input. I would also like to thank Vishal Apte, Patrick Fry, Amir Sehic, Donna Dietz, Michael Balma, Bill Malchisky, Amy Blodgett, Bob Dugan, Dave Mercer, Robyn Hetrick, Brian Kennedy, Sonya Dahnka, Mark Ferman,

Tom Jennings, Tim Klink, Laura Klink, Craig Lampert, Tim Jones and many others for making the past six years more enjoyable. I would like to extend a very special thanks to Judy Fleischner, as she was instrumental in laying the foundations many years ago that would ultimately motivate me to graduate study. For this I am eternally grateful.

Lastly, I owe much appreciation and am forever indebted to my family. I grew up in a very stable family with two brothers and one sister. My brothers Michael and James Nesheiwat have taught me by example. They truly embody all that is dignified, professional, fun-loving, and morally sound. My sister, Renee Youssef, has always been there for me, from 5th grade school projects to the cheering section of graduation. Over the years, I've seen my family grow with the addition of my sister-in-laws Heyam and Elisa Nesheiwat and my brother-in-law Hanni George Youssef; and of course I am reminded of the renewal of life and the innocence of character through my four loving nieces and nephews: Sara, Steven, Michael, and Hannah.

Above all, two people who truly deserve mention and thanks are my parents, Jamal and Najah Nesheiwat, to whom this work is dedicated. Baba and Yama, you filled my life with quality, morality, faith, and every kind of support imaginable. The intangible you have given me can never be explained to anyone; for it is only in my heart that you made the greatest impact ... a quality that I shall take with me wherever my life path may lead. "Thank you" is a mere thought to express my love to you both ... forever. All that I am, I owe to you. I will be forever grateful, and proud to be your son.

The final dedication is expressed to God, who has Blessed me with a family who has supported me, friends who were always there for me, mentors from whom I learned, my precious life, and a future with Him respectfully by my side.

ABSTRACT

The complexity and computational intensity of scientific computing has fueled research on parallel computing and performance analysis. The purpose of this thesis is to critically investigate the state of the art in performance analysis for scientific computing and then propose and demonstrate through implementation, the feasibility of a novel approach to performance analysis and experiment design. The core of this approach is an Instrumentation Database (IDB) that enables comparative analysis of parallel code performance across architectures and algorithms.

The basis of the IDB approach is scalable collection of performance data so that problem size and run-time environments do not affect the amount of information collected. This is achieved by uncoupling performance data from the underlying architecture and associating it with the control flow graph of the program. We consider the subset of nodes that are critical to performance: procedures, loops, calls, and communications/synchronization events. The resulting structure, a control flow hierarchy, is comprised of these nodes. Each node has statistical data collected for it; namely, a minimum, maximum, average, and standard deviation of its execution time, along with the number of times it was executed. Since these values are of fixed size, and the number of nodes is not a function of either problem size, system architecture or execution environments, data scalability is ensured. Another important contribution of the IDB approach is the use of database technology to map program structure onto relational schema that represent the control flow hierarchy, its corresponding statistical data, and static information that describes the execution environment.

To demonstrate the benefits of the proposed approach, we have implemented a POSIX compliant probe library, automated instrumentation tool, front-end visualization programs, database schema using an object-relational DBMS (PostgreSQL), and SQL queries. We also developed a methodology, based on these tools, for interactive performance analysis which we demonstrated by analyzing several different parallel scientific applications.

CHAPTER 1
Introduction and Historical Overview

The complex and computationally demanding nature of scientific applications has fueled research in the area of parallel computing. Moving from conventional uniprocessor systems to multiprocessor systems makes designing, developing, testing, tuning, and maintaining scientific codes much more difficult. These difficulties are counterbalanced by the significant speedup that parallel computing can provide.

Since the primary reason for writing parallel codes is speed [32], it comes as no surprise that performance analysis is a vital part of the development process. Analysis tries to determine if a given algorithm is as fast as it can be, where the program can be further optimized, and how efficiently the underlying system is being used. Raj Jain [35] explains that analysis, for both sequential and parallel systems, can be done in one of three ways:

- **Analytic Modeling** which involves using models of the executing program and its underlying architecture to derive performance information. While this technique can yield data quickly, the accuracy of this data is subject to the number of initial assumptions and complexity of the models used.

- **Simulation** that affords more accuracy than analytic modeling. In this approach, the target system's response to the executing program is simulated. It requires less assumptions to be made about the execution environment. The drawback is that simulation quickly becomes too time consuming, and, in some cases, not feasible as the system complexity or size grows.

- **Measurement** that is the focus of the research described in this thesis. It involves instrumenting an executing application. This is the most time consuming of the analysis techniques, but, since measurements are taken on the target system, it is clearly the most accurate [34].

We begin with a brief overview of the basic structure of performance analysis tools. It is here that we define the criterion with which we evaluate tools that are

representative of the state of the art in performance analysis. Following chapters present a description of the *Instrumentation Database* (IDB) approach, a system which we realized based on our evaluations and proposed criterion; it is the main contribution of this thesis. In addition to describing the approach, we present a detailed description of its primary components:

- **Automated Instrumentation Tool**
- **Probe Library and multi-language API**
- **EDFtool and Vistool graphical front-end**

We demonstrate the features of the framework and resulting methodology by evaluating performance results of several scientific computing applications. We conclude with a discussion of IDB's contributions, and list future directions for further research in this area.

1.1 Motivation of Research

The goal of this research is to explore new techniques in which both sequential and parallel scientific applications can be analyzed for purposes of optimization and performance tuning. This involves exploiting technology from other areas of computer science, specifically databases, to collect, store, and analyze collected performance data.

To be useful, such techniques must afford the programmer flexibility to conduct and manage customized performance experiments, power to answer key performance related questions, support for multiple languages and platforms, and, lastly, extensibility to analyze data and task parallel object-oriented codes.

1.1.1 Analysis Techniques

Analyzing the performance of an application is an iterative process which begins with executing the program to determine if further analysis is necessary. The next step involves profiling the code at a high level to determine which modules spend the most time executing. These performance critical modules are analyzed

further to localize bottlenecks to specific program constructs. It is up to the programmer to determine the cause of these bottlenecks and to remove them. This top-down approach to performance analysis prevents the programmer from spending time optimizing areas that are not on the critical path [7] [32]. Analysis of this nature is carried out mainly for two reasons:

- **Optimization.**
- **Comparative Analysis.**

In addition to ensuring that a code is as fast as possible, it is important to be able to determine how effectively the code is able to make use of the target architecture. In many cases, a code will perform significantly better on one system than another, depending on where the strengths of that system lie and the demands the executing program places on resources. This is of particular importance when evaluating different architectures for specific applications [11] [19] [31].

Adaptive parallel finite element codes represent a class of applications that can benefit significantly from the analysis approach we present. Typically, these codes impose significant computational demand, hence the need for parallelism. The difficulties that typically arise from analyzing these large and computationally demanding applications are exacerbated because these codes have scores of input and tuning parameters that drastically effect performance as a function of the underlying system and the specific problem being solved. The scalable instrumentation and program database approach that we present addresses these issues by defining each program execution as an experiment and providing a formal framework for experiment management.

1.1.2 Structure of Analysis Tools

Performance analysis tools are comprised of three basic components: *instrumentation*, *visualization*, and *support for analysis* [34]. Instrumentation refers to steps taken to collect performance related information from executing programs. Visualization involves processing this data in such a way that program behavior can be viewed graphically. Support for analysis is assigning causes and prescribing

solutions to performance problems exposed in the instrumentation and visualization phases.

Instrumentation includes counters showing how frequently a module is invoked, timers showing how much time is spent executing, or timers measuring communication and synchronization delays. There are four metrics on which the quality of instrumentation should be judged:

- **Probe effect**
- **Granularity of data**
- **Mapping to source code**
- **Cost of invocation**

All instrumentation is subject to the *Probe Effect* [58] to varying degrees. Introducing instrumentation alters the behavior of the program being analyzed. Instrumentation not only changes the execution time of the program, but asynchronous events that were governed by the program are now governed by the program and its accompanying instrumentation [43]. The probe effect goes beyond tainting timing and altering synchronization; it can drastically alter memory access patterns, thus affecting cache utilization. Minimizing probe effect is a significant issue in the area of performance analysis research.

The granularity of instrumentation data range from individual statements to a holistic view of the state of the machine. This leads to the issue of *data scalability*, in which the amount of instrumentation data collected is often proportional to the execution time of the program, the size of the problem, and the number of processors used. In some cases, tools such as *AIMS* [34] collect and write instrumentation data to disk at a rate of megabytes per second. Overwhelming system resources in this way taints measured performance because the instrumented program has to wait for resources, in this case the disk, when it normally would not. The same applies to other resources such as memory, or CPU. Preserving data scalability is a guiding principle in our research.

Instrumentation Unit	Observable Event
FXU	Branch delays Data cache misses TLB reloads Floating-point interactions
ICU	Instruction cache TLB misses Fetched instructions Dispatched instructions Executed instructions Number of interrupts
FPU	Floating-point cycles Floating-point instructions executed IEEE Inf and NaN delays Register renaming / queue stalls
SCU	Number and type of storage operations Storage request latencies DMA activity Bus activity

Table 1.1: Observable events for IBM POWER2 hardware instrumentation units.

Performance data showing the state of the underlying system is not sufficient for localizing bottlenecks and analyzing an executing program. This information needs to be mapped to specific source code constructs. In this way, the program's effect on the system can be understood in terms of specific modules, loops, communication, and synchronization events. We present a novel approach that addresses the issue of mapping instrumentation to source code features.

If the cost of adding instrumentation to an application is high, it is not likely that it will be used. In the ideal case, instrumentation should be controlled by a parameter that is turned on or off; instrumentation itself should not involve any additional effort on the user's part, aside from customization. Performance tools that require instrumentation to be added manually become impractical for large applications.

Instrumentation can be introduced at various levels:

Event	Counter 0	Counter 1
0	Cycles	Cycles
1	Instructions issued	Instructions graduated
2	Loads/syncs issued	Loads/syncs graduated
3	Stores issued	Stores graduated
4	Conditionals issued	Conditionals graduated
5	Failed conditional	Floating-point instr. graduated
6	Branches decoded	Write back from data cache to secondary cache
7	Write back from secondary cache to system interface	TLB refill exceptions
8	ECC cache errors	Branches mispredicted
9	Instruction cache misses	Data cache misses
10	Secondary cache misses (instruction)	Secondary cache misses (data)
11	Secondary cache way (instruction)	Secondary cache way (data)
12	External intervention requests	External intervention bits
13	External invalidation requests	External invalidation
14	Virtual coherency	Upgrade requests on clean secondary cache lines
15	Instructions graduated	Upgrade requests on shared secondary cache lines

Table 1.2: Assignable counter events for SGI/Cray R10000 processor.

- **Hardware:** Hardware can be used to count the number of mathematical operations, monitor network utilization, time critical events, etc. Examples of hardware instrumentation include network load or internal processor state indicators. These impose little or no cost as they are embedded systems designed to probe machine state. Many platforms include interfaces to instrumentation hardware, for example:

  - **IBM POWER2** provides extensive hardware facilities that are accessible by software through registers. Instrumentation is controlled by a multi-chip module that includes five basic units: Fixed-Point Unit (FXU), Floating-Point Unit (FPU), Instruction Cache Unit (ICU), Storage Cache

Unit (SCU), and Data Cache Unit (DCU). Each unit, except for the DCU, has counters that can measure up to five events simultaneously; the DCU has only one. These units are controlled by a Monitor Mode Control Register (MMCR) where observable events are selected for each unit. Table-1.1 shows observable events for the instrumentation units.

- **SGI/Cray MIPS R10000** supplies two hardware counters. Each counter can track one out of sixteen possible events per counter. In conjunction with the counters are two control registers which are used to specify which events are to be instrumented. When the most significant bit of a counter is set, denoting an overflow, the CPU receives an interrupt. All software interfaces to the 32-bit counters are done through a virtual 64-bit counter. Valid events for each counter are shown in Table-1.2.

- **Kernel:** Special facilities can be provided by the kernel and operating system to monitor system call usage and probe the state of the machine using timed interrupts. For example, UNIX provides several interfaces to report performance data collected by the operating system.

  - **vmstat** reports virtual memory statistics that include memory utilization, page faults, cache utilization, etc.
  - **iostat** reports I/O statistics such as percentage of time spent waiting for disk or throughput.
  - **sar** is the *system activity reporter*, that probes system counters at timed intervals; counters include: file accesses, buffer activity, system calls, etc.
  - **top** displays and updates information about the processes consuming the most CPU resources.

  Many of these facilities are interfaces to hardware- and kernel-based instrumentation services.

- **Binary:** Some tools, e.g., *Paradyn* [49], modify the binary image of an executing program to collect timing and frequency information for critical areas of

the program. This technique involves overwriting a portion of the binary with a jump to an instrumentation module that replaces the overwritten instruction and then starts/stops a timer or increments a counter.

- **Compiler:** Instrumentation can be added without modifying the original program text by having the compiler collect information about the program structure and linking in profiling libraries that extend the functionality of standard libraries. An example of this would be the Gnu profiler, *gprof*. The gprof utility produces an execution profile by incorporating call graph information collected at compile time with basic profiling information collected at run-time. Instrumentation data propagates along the call graph with total execution times and call counters for each of the nodes. Invoking the profiler involves minor changes to compile-time options and no modification to the source code. However, the granularity of information returned is limited to that of the call graph.

- **Source:** When other means are not available or appropriate, adding code to the program is the easiest way to collect performance data. This can be as simple as adding timers and counters to measure specific regions of the code where bottlenecks are likely to exist.

There are benefits and trade-offs to instrumenting at each of these levels. Hardware-based instrumentation provides the most accurate and fine grain data with minimal probe effect, but it suffers from several drawbacks. Probing machine state in this way provides little or no mapping of performance data to specific regions of the source code. Although many supercomputers provide specialized instrumentation hardware, it is often specific to each system and does not have a standard interface. The PAPI project at University of Tennessee at Knoxville is addressing this issue [52] and is discussed later. Another drawback is that there is also little or no cost for invoking hardware-based instrumentation. Kernel, or operating system, based instrumentation includes sampling environments where the machine's state is periodically probed by the kernel. Like hardware instrumentation, this provides fine grain data with little or no mapping to the executing program, but with low

invocation cost. Exclusively probing machine state (as Figure-1.1a shows) is not sufficient as the sole means for localizing bottlenecks and analyzing performance because data cannot be correlated with specific features of the executing program's source code.

The other instrumentation methods provide varying degrees of granularity and invasiveness. Instrumenting at the binary level may be dynamic provided there is external system support for it [50] [33]. This type of instrumentation is mainly applicable to long-running programs where there is time to dynamically add and remove probes. It is more costly to invoke because the run-time environment must provide support for dynamic instrumentation. Compiler and source code level instrumentations offer variable levels of granularity and traditionally are most invasive, however the benefit is that performance data is more easily mapped to the executing program's source code. The cost of invoking compiler-level instrumentation is minimal in that it involves linking in instrumentation modules but does not provide the flexibility that directly instrumenting source code does.

In recent years, object-oriented languages used for parallel programming have nearly become standard, thus exacerbating the need for new instrumentation techniques [54]. Traditional instrumentations are based on the program's control flow graph. Object-oriented programs require a higher level of abstraction based on coupling control-flows with object connectivity.

Visualization. A great deal of effort has gone into performance visualization. There are a large number of tools on many platforms that provide visual displays of performance data [29] [30] [28] [55]. Similar to data scalability, there is the notion of *visual scalability*. Most visualizations work well for programs that run for a short time on a small number of processors. As these quantities increase, the visual display becomes less useful and more overwhelming to the user.

Support for Analysis involves more than drawing conclusions. It involves breaking down, filtering, and organizing available information, thus enabling users to draw multiple conclusions [64]. It is here that our instrumentation database approach is most useful. A framework for understanding a program's performance can be formed based on instrumentation data collected in a scalable manner. This in-

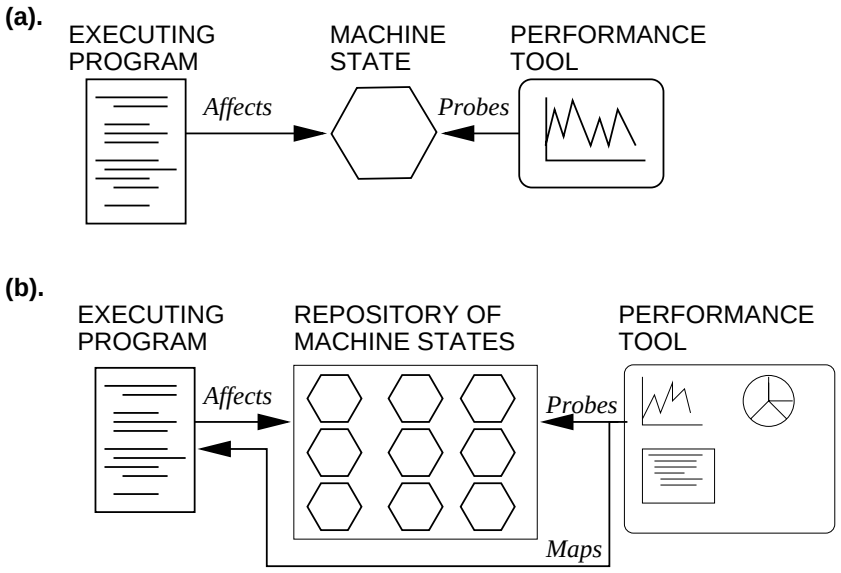


Figure 1.1: (a). Illustrates traditional performance tools that probe the state of the machine while executing applications affect the machine. (b). Illustrates an environment where the program's affect on machine state is archived and mapped back to specific program constructs. This information is visualized in different ways.

formation, in conjunction with static information about the architecture, program, and inputs, can be used to derive an integrated view of the program's performance across multiple runs, input vectors, and architectures.

1.2 IDB Overview and Objectives

The focus of this work is to explore issues and research problems central to performance analysis of parallel and object-oriented scientific applications. The key issues we address include:

- Support for comparative analysis / experiment management.
- Data scalability.
- Mapping of performance data to source code.
- Support for object oriented parallel codes.

- Ease of use and accuracy.
- Standardizing interfaces to performance data.

After considering these issues, it follows that performance data should be uncoupled from the underlying architecture and associated with the control flow graph of the executing program. The resulting data structures are too complex to be captured using event trace files. Instead, we exploit existing database technology by mapping program structure and fixed-size statistical data onto formal database schema. This novel use of an instrumentation database provides a framework for experiment management, and enables source code mapping since performance data is cast in terms of program structure and not underlying architecture. Scalability is maintained by aggregating statistical data during data collection. Database queries provide a powerful interface for front-end visualization and analysis tools.

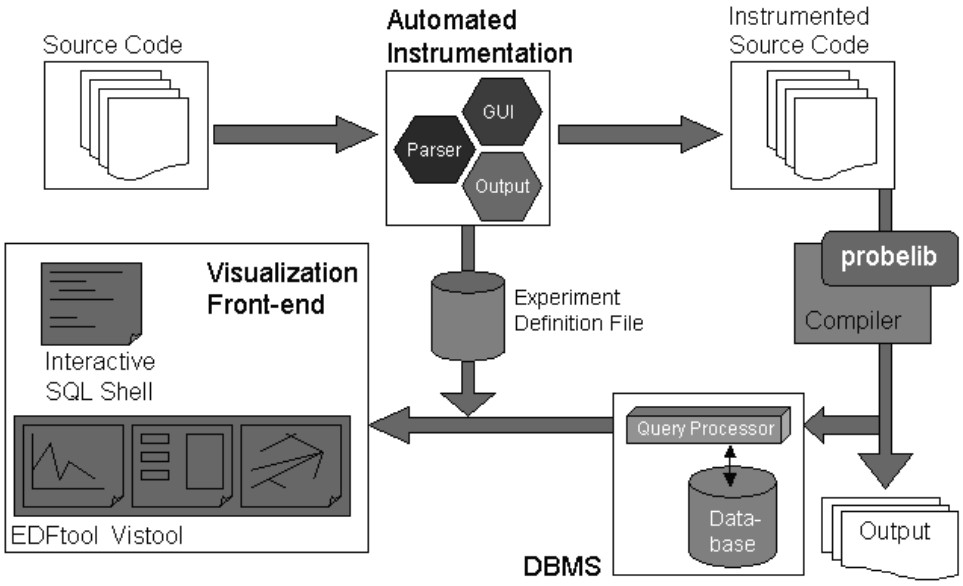


Figure 1.2: Instrumentation database architecture.

This instrumentation database framework (see Figure-1.2) archives collected performance data for a given program. As the program is run repeatedly with different parameters, the database can be used to derive conclusions about overall performance (see Figure-1.1b). [36] [53]

CHAPTER 2
Survey of Previous Work

Analyzing program performance on a specific platform is possible due to the large number of performance visualization and analysis tools provided by computer manufacturers and vendors [34] [9]. Problems arise when there is a need to compare program performance across multiple architectures. In addition to these vendor provided tools, there are systems being developed by various research teams that are portable from one architecture to another [72], [52], [49].

We critically evaluate some of these tools, summarized in Table-2.2, that are representative of the state of the art in performance analysis with respect to the criteria we outlined in the preceding chapter. The approach that we present combines the innovation from some of these research tools with the sophistication and power of the platform specific vendor tools while realizing issues that are not fully addressed by any of the tools that we considered.

2.1 Upshot

Message Passing Interface (MPI) [58] [65] has been implemented on many systems. The MPIch implementation comes bundled with Upshot, a TCL/Tk program used to visualize the performance of MPI programs. Upshot works as a visual front-end for MPE, a profiling interface for MPI. Instrumentation is added when the user links in MPI profiling libraries by specifying command-line options at compile-time. Upshot provides a graphical front-end to the resulting profiler data for MPI specific functions. Instrumentation is transparent to the user because MPI functions are overloaded with calls to instrumented functions in the profiling library. The user can further extend instrumentation by bracketing code segments with explicit instrumentation calls.

Gantt charts, which show processor state and arrows between each processor represent message passing, are a powerful visualization tool, but suffer from poor scalability. Figure-2.1 shows an Upshot visualization of an adaptive finite element

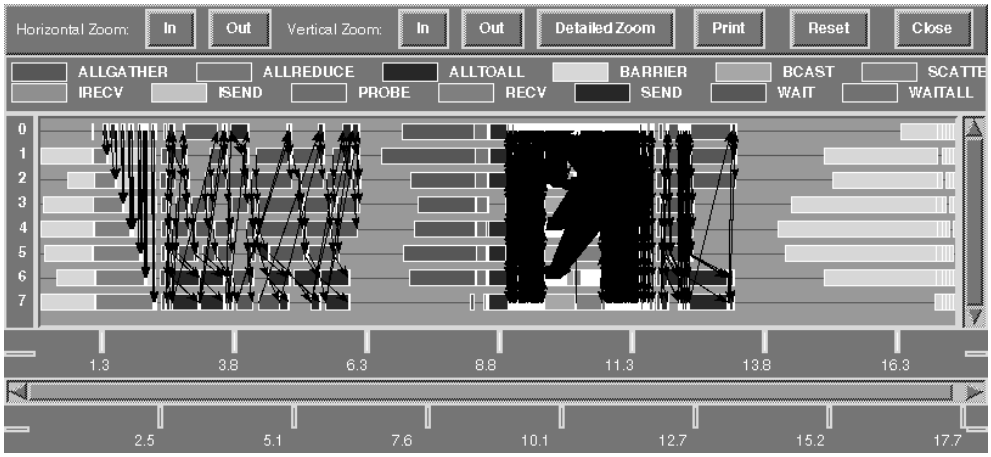


Figure 2.1: Upshot visualization of mesh partitioning algorithm using 8 processors. High degree of interprocessor communication render this an ineffective way to visualize performance.

application [44] with heavy interprocessor communication. Heavy communication, many processing elements, or long running programs quickly make this type of visualization unwieldy for analysis. Upshot's usefulness is bound by the size of the application being analyzed.

In addition to visual scalability, it is desirable to be able to answer different performance questions from the same instrumentation data set. Upshot does not provide the flexibility of multiple visualization modes to meet this need. This differs from the approach we propose; IDB instrumentation data is uncoupled from the visualization front-end thus allowing independent development of visualization tools.

2.2 MPP Apprentice

MPP Apprentice instruments and visualizes performance of C and FORTRAN parallel programs on SGI/CrayT3D and T3E systems [9] [34]. Instrumentation is done at the compiler level with the creation of a compiler information file (CIF file) that contains static information about program structure. When the program executes, a run-time information file (RIF file) is created automatically. These two files are used by the visualization front-end to generate scalable performance displays. Instrumentation is introduced automatically when the user links MPP

Apprentice libraries by specifying appropriate command-line options at compile-time.

Apprentice provides multiple program visualizations, as illustrated in Figures-2.2, 2.3, 2.4, and 2.5, which show performance results of an MPI application that enumerates twin primes [24]. Visual scalability is achieved because module performance is displayed as a histogram (see Figure-2.2) and is coupled with program structure. Timing information derived from the RIF file is used in conjunction with structural data contained in the CIF file (see Figure-2.3).

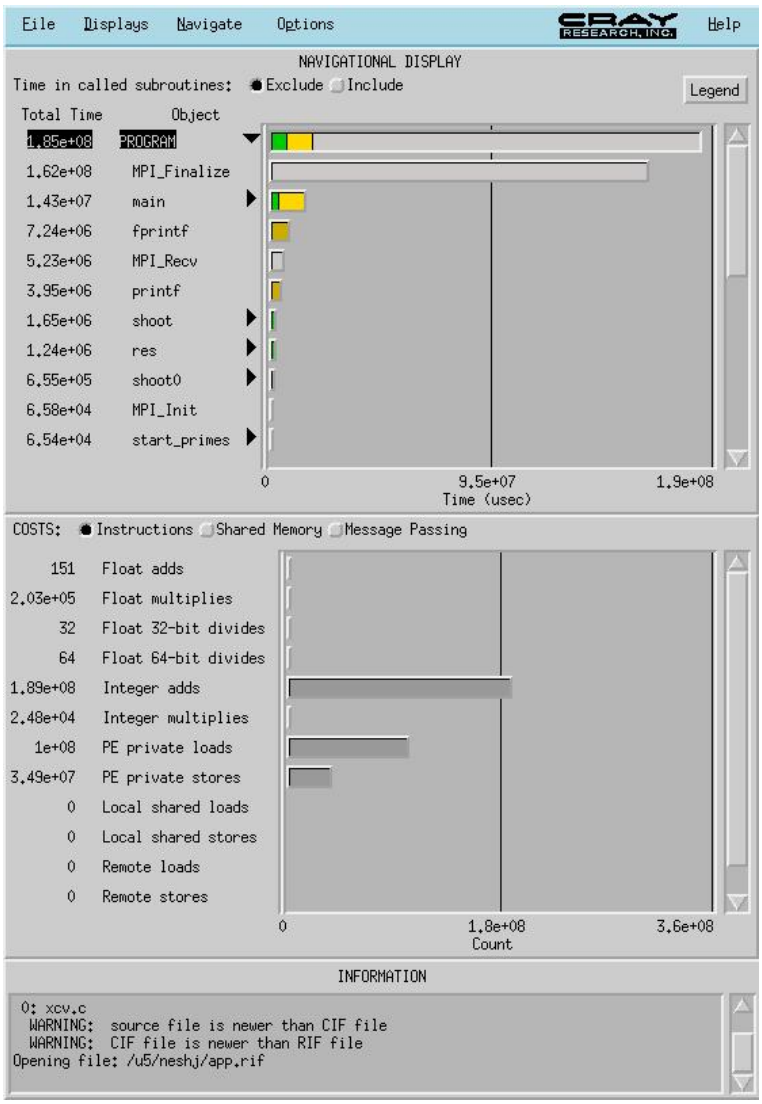


Figure 2.2: Main screen of Apprentice running on a CrayT3D.

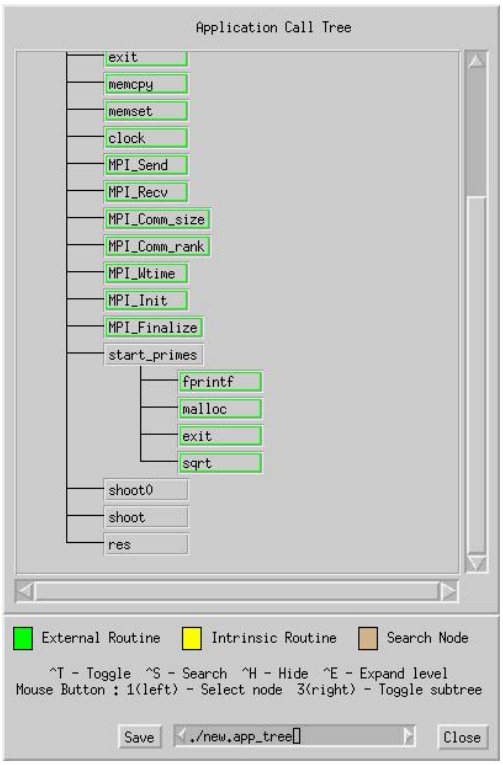


Figure 2.3: Call graph view of Apprentice running on a CrayT3D.

Apprentice makes use of hardware instrumentation facilities provided by the Cray T3D/T3E, that include information about cache utilization, floating point operations, I/O performance, etc. An extensive knowledge base uses this data to simplify analysis. Specifically, it correlates machine state information collected in hardware with structural information about the program and generates a report of where bottlenecks may exist in specific regions of the program based on accurate statistics showing floating point operations and cache utilization. Figure-2.5 shows *suggestions* for where in the code bottlenecks may exist. The sophistication of these suggestions are limited to observations such as *poor cache utilization* or *time spent waiting for I/O* in a given subroutine.

MPP Apprentice excels in a number of areas. It provides clear and scalable visualizations of program activity and effectively maps performance data to the source code, as Figure-2.4 shows, due to compile-time instrumentation used in conjunction with traditional run-time instrumentation. Conversely, some drawbacks are that Apprentice only runs on Cray computers and that instrumentation adds noticeable

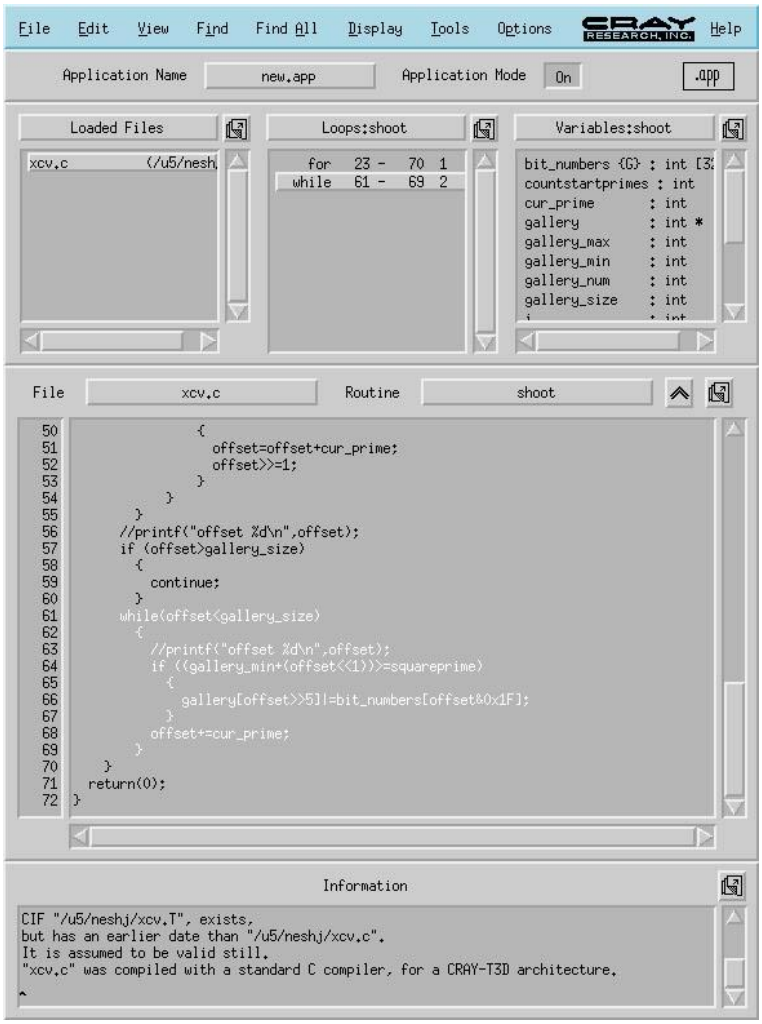


Figure 2.4: Source view of Apprentice running on a CrayT3D.

overhead to program execution. In some cases the overhead is so great that the problem size must be reduced in order for the tool to run.

Portability and scalability are not issues in the mechanism that we propose as instrumentation data is uncoupled from the underlying architecture. Furthermore, structural information is computed at run-time. This reduces overhead because, unlike the CIF files used by MPP Apprentice, IDB only collects structural information for code that is executed at run-time.

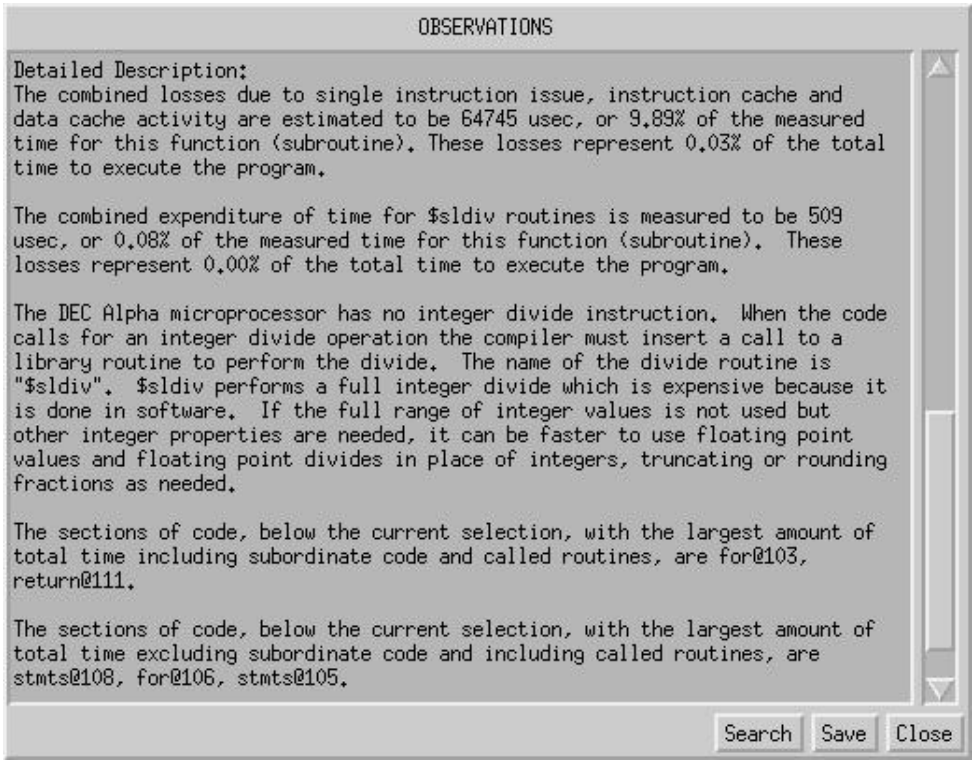


Figure 2.5: Observations view of Apprentice running on CrayT3D.

2.3 PAT

Performance Analysis Tool [1], or PAT, provides a low-overhead method for viewing execution time of functions, processor load, event traces, run-time stack, and hardware instrumentation output. PAT can be used to analyze programs written in C, C++, and FORTRAN 90 by linking in PAT libraries.

Instrumentation is done by using the real time clock and the program counter to time-stamp events. This information is stored in *pdf.nnn* files, where *nnn* is the process ID of the executing program. PAT provides accurate measurements of program performance while introducing very little overhead. This comes at the expense of being able to map this data to the source code of the executing program. Module level data is the finest granularity that can be achieved. When using libraries not linked with PAT, it is not possible to get module level information. Data scalability is also a significant issue, as the instrumentation files can become very large for non-trivial applications. In terms of accuracy, PAT is the clear leader

among the tools that we considered.

PAT is lacking in its ability to reconcile instrumentation with the executing program, because it is tightly coupled with the underlying hardware. This differs from the approach that we propose in that performance data is coupled tightly with the source code and completely separate from the underlying system. Moreover, maintaining time-stamps of critical performance events is not scalable. IDB differs in that it aggregates statistical data at run time.

2.4 AIMS

AIMS, or Automated Instrumentation and Monitoring System [72], was developed at NASA Ames Research Center, and includes the following tools for analyzing performance of parallel programs:

- **Source instrumentors** that are capable of instrumenting multiple languages and parallel programming libraries.
- **Monitoring libraries** that provide architecture dependent support for run-time instrumentation.
- **Intrusion compensator** which factors out instrumentation overhead and preserves partial ordering of events.
- **View kernel** that generates performance visualizations.
- **Statistics kernel** for profiling.
- **Index kernel** to facilitate performance tuning.
- **Modeling kernel** for performance modeling and prediction.
- **Trace converters** to convert AIMS trace files to formats used by other performance analysis tools.

AIMS is more than a performance analysis tool; it is a framework containing many tools which are available on several platforms and can be extended easily. All

the components of AIMS fall into one of four categories: source code instrumentation, run-time performance monitoring, noise reduction, and trace post-processing. Inserting source-level instrumentation in large codes can be a tedious process; AIMS automates this process by automatically adding instrumentation at the module level and at communication points. That is, data is gathered at each subroutine and message passing call. It also allows the user to manually define additional instrumentation points. Source level instrumentation is stored in a flat file that resides at the beginning of run-time instrumentation trace files. This allows collected run-time data to be mapped to source code constructs.

Run-time instrumentation enables collecting information when specific events are encountered. These events include:

- Subroutines, loops, and user specified regions.
- Communication events.
- File I/O.
- Global reduction operations, barrier synchronizations, and blocking events.

Information collected for these events include time-stamps, processor ID, event types, and additional data specific to each event. As the program runs, a trace file is generated containing results of static and run-time instrumentation. This trace file is then post-processed by an intrusion compensator which factors out instrumentation overhead, specifically calls made to run-time instrumentation functions. Trace files are then used by various visualization, modeling, and tuning kernels for analysis. The VK, or visualization kernel, provides a powerful interface for generating performance displays of event trace files, and uses static instrumentation to map run-time data to specific source code constructs. In addition to providing basic analysis tools, AIMS provides a set of converters for using trace files with other performance analysis tools.

Overall, AIMS is a very powerful tool which has made many innovative breakthroughs in this area. It is lacking in that it does not enable re-use of trace files across multiple platforms for the purposes of comparative analysis. Moreover, the

run-time instrumentation is not data scalable. For long running programs, trace files can become very large. Yan [72] describes this by saying:

> Although we have shown that AIMS can be a powerful tool for the development of parallel applications, there is much room for improvements ... Scalability: The 2-dimensional display formats and "event trace" approaches are not scalable. Enormous amount of performance data can be generated rather easily. The run-time overhead (e.g. flushing) and analysis overhead (e.g. the need to sort the records before feeding them to the visualization/analysis toolkit) could render the performance tuning methodology described here impractical. [72]

There are three scalability issues raised here: data scalability, where trace files get very large; visual scalability, where the performance displays do not convey large amounts of performance data well; and the notion of *analysis scalability*, where post processing is time-consuming because of large amounts of data produced by instrumentation. These limit the usefulness of AIMS to applications of manageable size. Also, while AIMS is portable across different architectures, it does not formally provide the support for comparative analysis or experiment management that IDB provides.

2.5 Godiva

Godiva, or Goddard Instrumentation Visualizer and Analyzer was developed at NASA's Goddard Space Flight Center to analyze FORTRAN90 and C codes running on SGI/CRAY T3E and Beowulf class supercomputers [59]. Godiva works in stages. First, the source files are annotated with instrumentation directives by the user. Next, these annotated source files are run through a preprocessor, which expands annotations to executable instrumentation code. This new source program is linked with the Godiva run-time library. When the program executes, trace files are generated for each processor based on this instrumentation. These trace files are then used to generate histograms, tables, and graphs showing various performance metrics.

There are a number of aspects that make Godiva unique with respect to the other tools. Foremost is the manually introduced source level instrumentation; it can be easily mapped to specific program constructs. The disadvantage of this approach is that manually introducing instrumentation is both tedious and prone to error. A preprocessor expands this instrumentation to actual code making use of the Godiva run-time components. This code is, by default, introduced outside of deeply nested iterative constructs to reduce intrusion. This differs from the approach we present in that all IDB instrumentation consists of calls to a standard application programmer's interface (API) that can be automatically introduced. Both tools are similar in that performance data is aggregated to preserve data scalability. Earlier versions of Godiva kept event trace files which quickly became too large to handle. Godiva, like AIMS, is one of a few tools designed to enable limited comparative analysis between different architectures, in this case SGI/Cray and Beowulf systems [40].

2.6 Paradyn

Among the most unique analysis tools that we considered is Paradyn [49], which uses *dynamic instrumentation* to reduce instrumentation overhead. Paradyn is ideal for analysis of long running programs. In addition to instrumenting and visualizing performance, it attempts to draw analytic conclusions. Initially, when a program runs, the user selects metrics to be considered. Some of these include:

- Execution time.
- Message bytes sent.
- Message bytes received.
- I/O wait.
- I/O bytes.
- Synchronization wait.

The user also has the option to restrict data collection to individual modules or *execution phases*. These phases are classified as intervals where overall performance is uniform with respect to time; for example, input, decomposition, output, fan-in, or computation. Paradyn searches for high-level problems relating to synchronization, blocking, I/O, and memory. Once a general bottleneck is found, fine-grained instrumentation to find more specific causes is added dynamically while the program is running. There are two abstractions for collecting, presenting, and analyzing performance data:

- Multi-focus grid.
- Time-histogram.

The multi-focus grid is comprised of two lists. The first is a list of performance metrics: CPU time, blocking time, message rates, I/O rates, etc. The second corresponds to program components, such as procedures, processors, shared resources, etc. These lists are used to define a matrix where metric data is recorded for program components. The metrics contained in this matrix can be single values or time-histograms which describe how the metric changes with time.

The overriding design goal is to automate the search for performance limiting regions of a given program. Figure-2.6 shows a schematic of Paradyn's run-time environment. There is a single multi-threaded process that contains the Performance Consultant, Metric Manager, Visualization manager, and User Interface Manager. Outside of this process, there are multiple daemon processes that act as Instrumentation and Metric Managers that connect to executing applications. There are also multiple visualization processes. The Performance Consultant searches for bottlenecks by using a W^3 search model [49]. It refines its search by dynamically introducing and removing instrumentation from the program as it executes.

2.6.1 W^3 Search Model

Performance tuning requires answering three basic questions:

- **Why is the program performing poorly?**

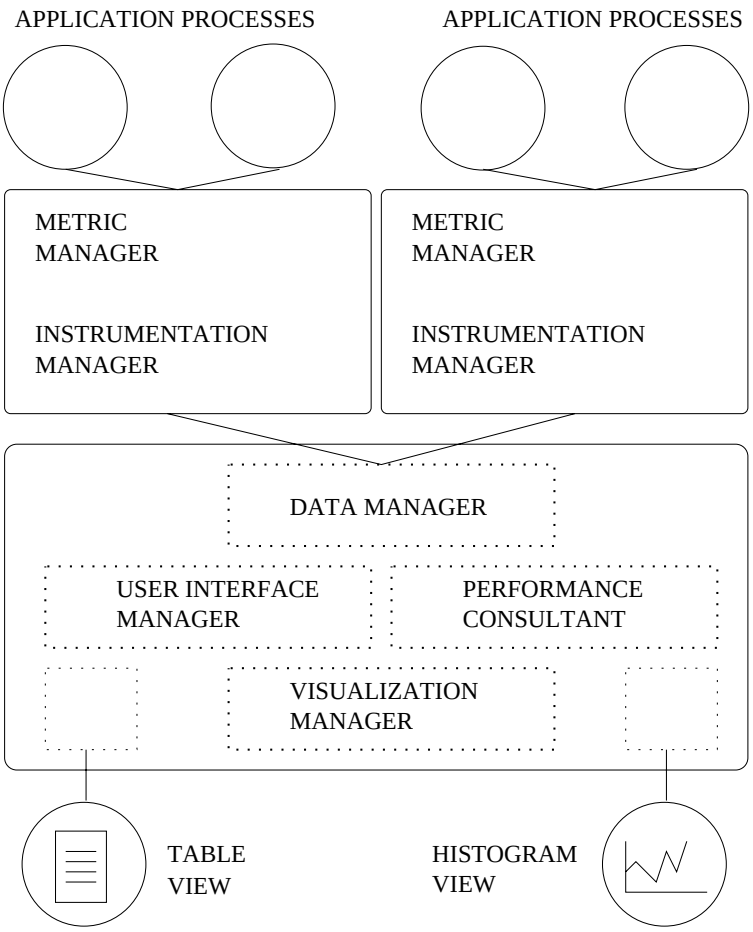


Figure 2.6: Schematic of Paradyn software. Solid boxes represent processors and dotted boxes represent threads.

- **When do the problems occur?**
- **Where are the bottlenecks?**

To determine *why* the application is performing the way it is, Paradyn poses a number of hypotheses as to which shared system resources are causing the problem. Each of these high-level hypotheses is validated or dismissed based on collected performance data. If validated, the hypothesis is further refined with respect to *where* and *when* it occurs. Determining *where* involves mapping one or more of these hypotheses to a specific region of the executing program. *When* involves localizing bottlenecks to a specific "execution phase" of the program. Figure-2.7 shows the Performance Consultant traversing a tree of hypotheses, with each level representing

the refinement of a hypothesis based on conclusions reached from the introduction of dynamic instrumentation in the previous level.

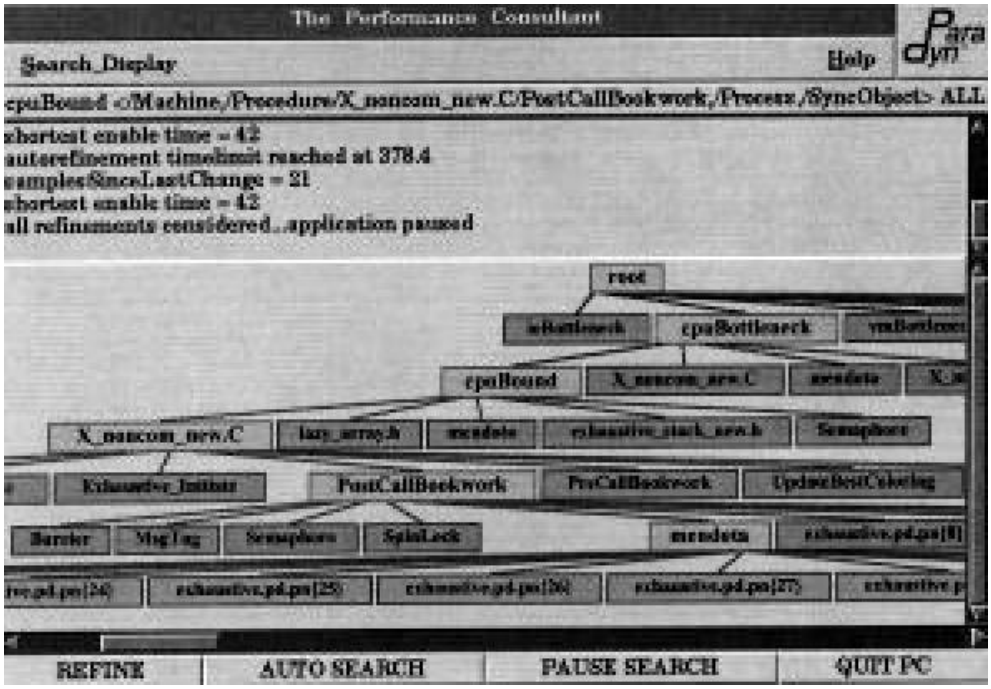


Figure 2.7: Performance Consultant search through w^3 tree for Graph Coloring program.

The method of forming hypotheses (*why*), and iteratively refining them with respect to *where* and *when* can be done interactively with the approach that we present. Instead of dynamically instrumenting the application as it executes, IDB draws on data from previous executions that reside in a database. It currently relies on the user to manually introduce or remove probes.

2.6.2 Dynamic Instrumentation

High-level requests for dynamic instrumentation are converted to architecture specific instructions by the Instrumentation Manager. Once Paradyn connects to the executing application, parts of the binary image are then overwritten with branch instructions called "trampolines". These trampolines perform a jump, introduce the instruction that was overwritten, and execute instrumentation operations. Execution is then diverted back to the point after the trampoline was introduced.

These trampolines are added and removed during execution. Although IDB does not provide support for dynamic instrumentation, it does provide an interface for automatically introducing instrumentation between subsequent runs.

2.7 Performance Application Programmer Interface (PAPI)

Modern microprocessors provide facilities in hardware for measuring system performance. These facilities include instrumentation registers, counters, and signals built directly into the hardware that are designed to collect information about the state of the machine. Some examples include registers that are updated with internal state information for such systems as: instruction/data cache, translation look-aside buffer (TLB), floating point unit, I/O buffers, etc. These facilities are potentially very useful, but there is no easy way to reconcile this data across different architectures.

The PAPI project [52] addresses this problem; it provides a platform independent interface to hardware instrumentation facilities across multiple platforms.

> PAPI provides portability across platforms. It uses the same routines with similar argument lists to control and access the counters for every [supported] architecture. PAPI includes a predefined set of events that we feel represents the lowest common denominator of every good [hardware] counter implementation ... These features provide the necessary basis for any source level performance analysis software. ... for any architecture with even the most rudimentary access to hardware performance counters, PAPI provides the foundation for truly portable, source level, performance analysis tools based on real processor statistics. [52]

PAPI is comprised of low-level and high-level interfaces that are both platform independent. These layers are built on top of a platform dependent *substrate* that, in conjunction with a kernel patch, connects the portable interface to the hardware instrumentation facilities provided by a specific platform (see Figure-2.8)

Substrates and kernel patches enable PAPI support for numerous operating systems on multiple processors, ranging from Intel Pentium-II, Pentium-III, MIPS

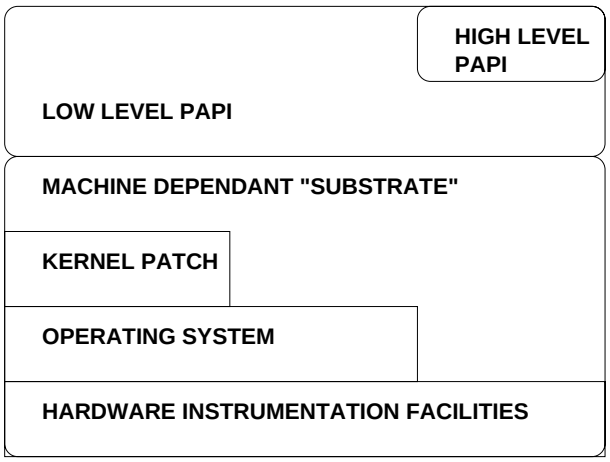


Figure 2.8: PAPI Architecture

R10K, R12K, and IBM POWER series processors. Tool developers and performance analysts can probe numerous events, some of which include:

- L1 Data/Instruction cache misses
- L2 Data/Instruction cache misses
- TLB Data/Instruction misses
- Total instructions executed
- Integer/floating point instructions executed
- Store/load instructions executed
- Total cycles
- MIPS/FLOPS

In addition to probing hardware counters, the PAPI provides sophisticated tools for setting overflow thresholds and multiplexing hardware counters. PAPI represents a significant contribution to performance analysis by making hardware dependent instrumentation accessible through a consistent and robust interface that

Call	Description
PAPI_library_init	Initialize PAPI
PAPI_get_opt	Return system specific attributes. Including: PAPI_GET_DEFGRN, PAPI_GET_HWCTRS,
PAPI_add_event	Add event to event list
PAPI_start	Begin collecting data for an event list
PAPI_stop	Stop collecting data for an event list
PAPI_shutdown	Shut down PAPI

Table 2.1: Commonly used high level PAPI calls.

can be used by performance analysts and tool developers alike. The cost of invocation is high because PAPI calls must be manually introduced by the user. Table-2.1 shows some sample PAPI calls that would be added to a target application.

IDB goes to great lengths to ensure that there are no underlying hardware dependencies. As a result, the type of performance information that can be collected is limited to wall clock and CPU timing. Integration with PAPI will enable IDB to exploit hardware instrumentation facilities that are available while preserving platform independence. As a result, the high cost of invocation associated with PAPI would be greatly reduced with the use of IDB's automated instrumentation tool.

2.8 Discussion of Previous Work

The tools we evaluated in the previous sections are representative of the state of the art in performance analysis for parallel applications. They address the criteria we defined in different ways:

- **Source code mapping.** Mapping performance data to specific program constructs is vital for optimization. Some performance tools simply probe the state of the machine, while the executing program affects the machine's state. In this way, there is no easy way to reconcile performance bottlenecks with specific code constructs. (see Figure-1.1a). The degree to which data can be mapped to code varies greatly; from a holistic view of machine state, to mapping performance data to call graph or run-time stack, to an instruction level

view of system performance.

- **Customizable visualization front-end.** Demand on system resources varies greatly from program to program. Moreover, resource availability is tightly coupled with the underlying system architecture. Program demand on system specific resources mandates that visualization tools support multiple views that emphasize both program structure and machine state. Front-end visualization tools should access performance data through a standard interface thus removing dependencies on underlying system and maximizing re-usability.

- **Comparative Analysis.** When determining which architecture is most appropriate for a given program, it is important to collect and analyze visualization data in a platform independent and re-usable way. Analysts can then gauge how well an application will migrate to a new system. Most tools do not support this type of analysis, and the few that do only provide support for selected architectures. The notion of comparative analysis can be extended to the more general notion of experiment management. In comparative analysis, system architecture is simply a parameter that we permute whereas experiment management provides a framework for managing data where many such parameters are permuted. These parameters include input vectors, number of processors, and even program modules. Of the tools we considered, AIMS and Godiva come closest to providing rudimentary support for comparative analysis; neither provide support for experiment management.

- **Probe Effect and Dilation.** Introducing instrumentation involves some degree of intrusion on the performance being measured. Instrumentation and accompanying intrusion compensation take time to compute and contribute significant overhead to executing programs. Ideally, instrumentation should provide a minimum of invasiveness such that dilation in "real" running time is minimal, as with PAT and PAPI. This way, programs can be evaluated with large, realistic, input vectors.

- **Easy to use and extensible.** In order for a tool to be useful, it must be used. It is vital that the cost of invoking the tool and adding instrumentation

is minimal. It should also be possible to extend functionality to enable analysis of diverse programs and architectures. AIMS and Paradyn provide this extensibility, but at the expense of being easy to use.

These items represent research problems that need to be addressed. The system that we realized to address these issues, IDB, involved a multidisciplinary approach drawing upon areas such as user interfaces, data visualization, compilers, automated testing, experiment management, and databases [34] [61].

2.9 Summary of Evaluation

Table-2.2 shows a summary of the tools we evaluated with respect to data and visual scalability, support of multiple views, overhead, cost, and source code mapping.

Tool	Data Scalable	Visually Scalable	Multiple Views	Portable	Overhead and Cost	Source code Mapping
Upshot	Moderate	No	No	Yes	Low	Moderate
Apprentice	Poor	Yes	Yes	No	High	High
PAT	Poor	Yes	No	No	Low	Moderate
AIMS	Poor	Yes	Yes	Yes	High	High
Godiva	Good	Yes	Yes	Limited	High	Moderate
Paradyn	Good	Yes	Yes	Yes	High	Moderate
PAPI	N/A	N/A	N/A	Yes	Low	Low
IDB	Good	Yes	Yes	Yes	Low	High

Table 2.2: Survey of evaluated tools. Note that PAPI is not a stand alone tool; rather it is an application programmer interface used by tool developers.

We see that Apprentice, PAT, and AIMS suffer from poor data scalability whereas Upshot suffers from poor visual scalability. This leaves Godiva and Paradyn. While both are scalable, the cost of invocation is high because of manual instrumentation and dynamic instrumentation support, respectively. PAPI provides a portable interface to hardware counters for increased accuracy. It is up to tool developers to use PAPI to build data scalable tools with support for multiple visually scalable views that can be easily reconciled with the program source code.

CHAPTER 3
Scalable Instrumentation

Scalable instrumentation and the use of a program database are two novel characteristics of our approach. Program structure is mapped onto relational database schema using a minimal amount of performance data that is collected in a scalable way at run-time.

3.1 Scalable Instrumentation

Traditionally, the amount of performance data collected is a function of the number of processors used; and trace file size is a function of the execution time of the application. This limitation restricts programs that can be analyzed to those with small input vectors, or which runn for short durations of time. There is a need for scalable data collection where size is a function of program structure. There is also a need to map this data back to the source code of the executing program. These issues are addressed by tightly coupling instrumentation with the program's control flow graph (CFG). A CFG based view of the program ensures that data scalability is achieved, provided that the data collected for each node in the CFG is of fixed size. Moreover, CFG nodes are readily mapped to specific source code constructs. This mapping is one of our main contributions.

3.1.1 Overview of Sample Codes

The use of scalable instrumentation that we illustrate in this chapter is motivated using two applications, *Spark98 Sparse Matrix Vector Product Kernels* and *Pyramid Adaptive Mesh Refinement Library.*

- **Spark98 Sparse Matrix Vector Product Kernels** The examples in this chapter are derived from Spark98, a set of sparse matrix vector product (SMVP) kernels that include shared memory and message passing parallel codes. Spark98, extracted from Carnegie Mellon's Quake Project which models ground movement during earthquakes (see Figure-3.1), is designed to pro-

vide system designers and analysts with a small set of kernels that represent realistic SMVP applications [56]. The control flow hierarchies shown in this chapter are derived from this code.

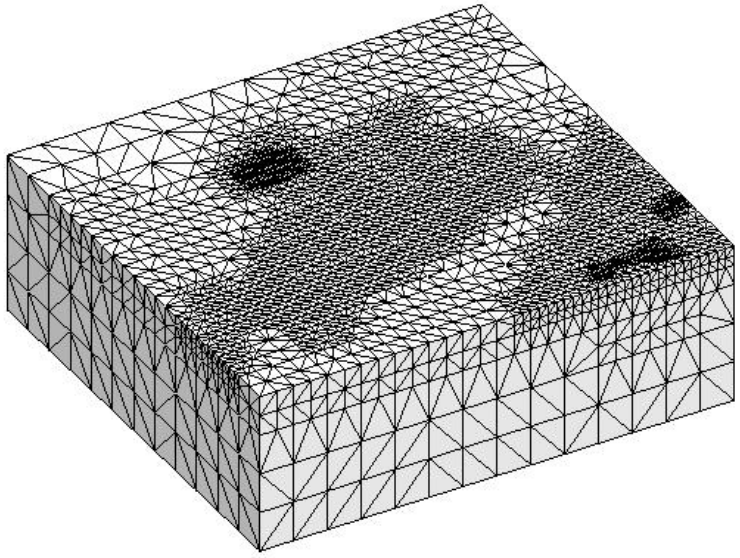


Figure 3.1: Spark98 input mesh modeling ground movement during an earthquake.

The fine-grained instrumentation that was applied to this code demonstrated the need for noise reduction algorithms. Analysis of standard POSIX compliant timing routines showed that overhead was highly platform and operating system dependent. For example, a call to `gettimeofday()` took four times longer under IRIX than on Solaris based computers. Introduction of noise reduction, which we describe later, compensated for this overhead. Actual program execution, however, took 20% longer. This dilation was attributed to system call overhead and affected sequential codes that do not use MPI based timing routines.

- **Pyramid Adaptive Mesh Refinement Library** *Pyramid* is a software library for performing parallel adaptive mesh refinement on unstructured meshes [44]. The library is designed to work on triangular and tetrahedral meshes and supports development of unstructured parallel applications such as finite

element, finite volume, and visualization. The library is implemented in FORTRAN 90 and has an interface to MPI.

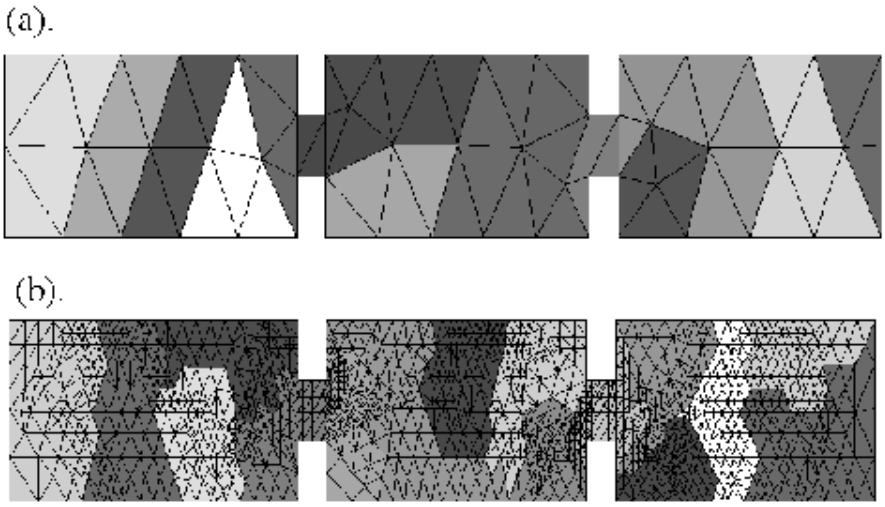


Figure 3.2: (a). Initial beam-waveguide mesh (b). The same mesh after three refinement phases. Shading indicates on which processors mesh elements reside.

The program used to test the Pyramid library performed three refinements of an input beam-waveguide mesh. Figure-3.3 shows the structure of the test application. All tests were run on NASA Goddard's SGI/CrayT3E. Figure-3.2 shows the input test mesh and resulting mesh after three refinements,

3.1.2 Control Flow Hierarchies

There are four events that impact performance significantly: Procedures (PROC), Loops (LOOP), Procedure Calls (CALL), and Communications / Synchronization (COMM) [43]. Instead of considering the entire control flow graph (CFG), we look at the subset of nodes consisting of only these performance critical events. The resulting subgraph is the *Control Flow Hierarchy*, CFH for short. Each node in the CFH maps these performance critical events to statistical data collected at run-time. A *probe* is introduced in the source code corresponding to each of these critical event nodes to collect aggregate timing and statistical data.

To ensure data scalability, individual data points are not collected. Instead,

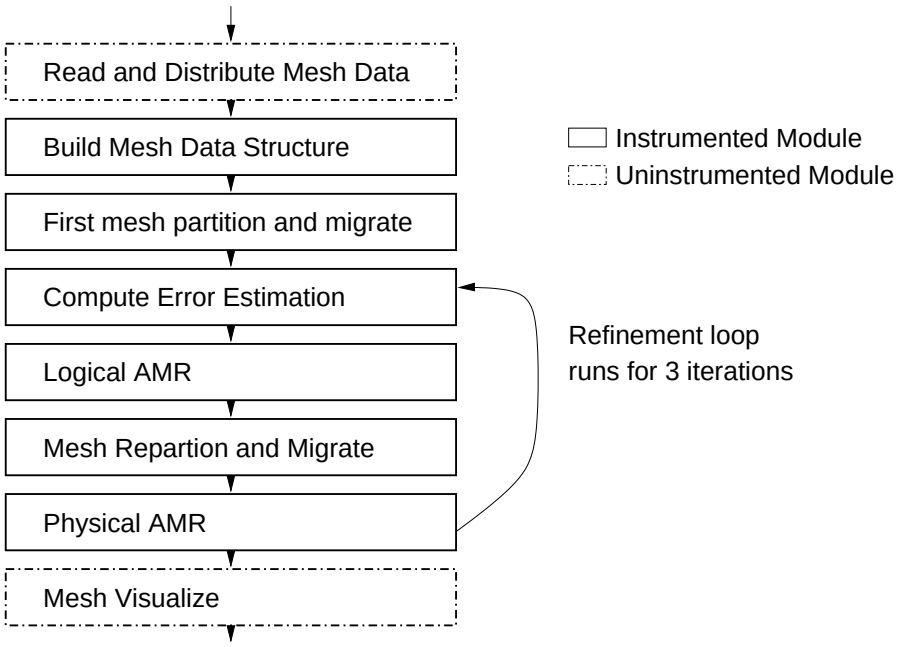


Figure 3.3: Flow of Pyramid test program for beam-waveguide mesh.

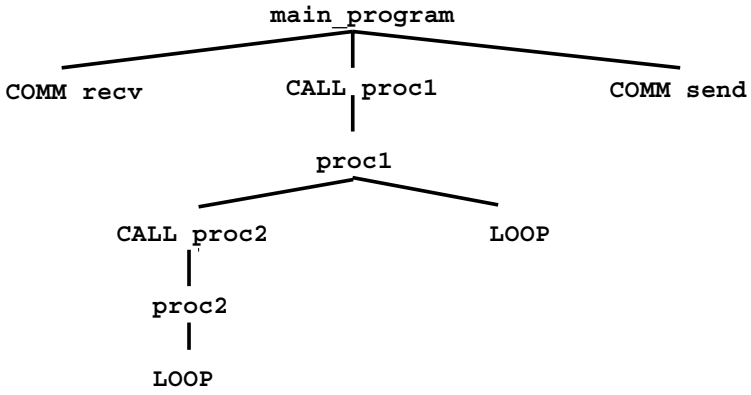


Figure 3.4: Sample control flow hierarchy.

run-time statistics are continually refined when a probe is encountered. When the statistics are updated, the data point is discarded, thus ensuring that each probe's information is of fixed size. Moreover, the size of the CFH is strictly bound by the program structure; hence, data scalability is ensured. Instrumentation data is collected with calls to an instrumentation application programmer's interface (API), as shown in Table-3.1. All probes are assigned a unique ID and TYPE corresponding to one of the critical performance events. This information is used by a database,

OPERATION	PARAMETERS	DESCRIPTION
INIT	IDB-FILE, CFG-FILE	Initializes data structures, reads configuration file (CFG-FILE). Prepares database (IDB-FILE) for writing.
CLOSE	NONE	Flushes data structures to database and deallocates storage
START	PROBE-ID, PROBE-TYPE	Allocates data structures for probe if needed, based on ID. Records event TYPE, Starts probe timers, increments counter and begins collecting noise reduction data.
STOP	PROBE-ID	Stops timer, updates noise reduction data

Table 3.1: Instrumentation API

DATA	DESCRIPTION
AVG	Average time spent executing event
MIN	Shortest time spent executing event
MAX	Longest time spent executing event
TIME	Time spent on current execution of event
SDEV	Standard deviation of measured times
COUNT/ITER	Number of times event was executed

Table 3.2: Statistics collected by each probe for a given PROC, CALL, LOOP, or COMM event.

which stores probe data, to map CFH connectivity information to statistical data for each probe. Specialized performance data can be derived, or inferred, from a minimal set of statistical data collected at run time. Table-3.2 shows what is collected by the probes introduced at each node of the CFH.

A database is used to capture these control flow hierarchies and their accompanying statistical data for each run of the program. Figures- 3.4, 3.5, 3.6, and 3.7 show sample control flow hierarchies from the Spark98 SMVP Kernels.

3.1.3 Noise Reduction Techniques

Performance tools strive to collect accurate data with as little intrusion to the executing program as possible. Various noise reduction techniques are used to ensure that the collected data is accurate. Instrumented programs suffer from a *dilation*

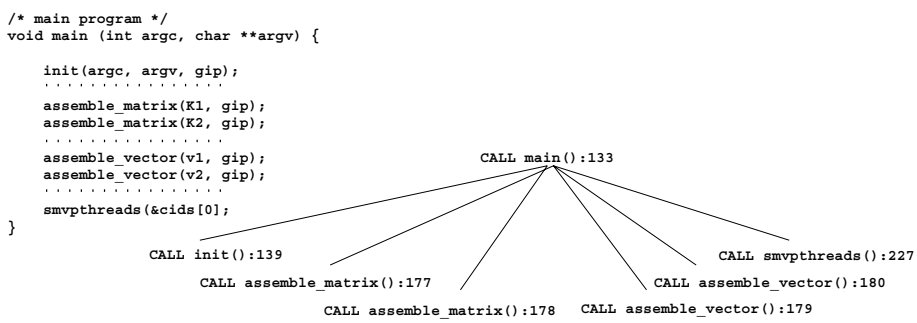


Figure 3.5: A fragment of the Spark98 `main` program and its accompanying control flow hierarchy.

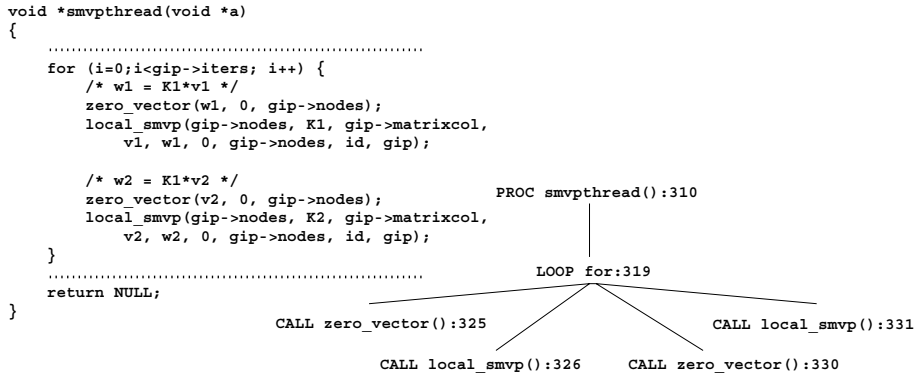


Figure 3.6: A fragment of the `smvpthread` module and its accompanying control flow hierarchy.

effect, which causes the program to take longer to execute because of instrumentation overhead. Dilation and noise are reduced by factoring out instrumentation overhead, efficiently implementing the instrumentation API, and selectively instrumenting critical portions of the executing program [47].

Each probe in the CFH has noise associated with it. This noise is a function of two factors: the nesting level of the probe from the root node in the CFH and the number of times it is activated. Each time a probe is encountered, data is collected and stored in the CFH. Thus, overhead is incurred on every PROC, CALL, LOOP, and COMM event. We employ two techniques to minimize and factor out this noise:

- **Factorization**. Each probe measures how long it takes to complete its own instrumentation activities and stores this information locally in the CFH as its cumulative overhead contribution. Noise is factored out by summing these

```
void assemble_matrix(double (*K)[DOF][DOF], struct gi *gip)
{
    ....................................................................
    for (i=0;i<gip->matrixlen; i++) {
        for (j=0;j<DOF;j++) {
            for (k=0;k<DOF;k++) {
                K[i][j][k] = 0.0;
            }
        }
    }
    for (elem = 0;elem<gip->elems;elem++) {
        for (j=0;j<gip->corners;j++) {
            for (k=0;i<gip->corners;k++) {
               ..................................................
                while (gip->matrixcol[temp1] != gip->vertex[elem][k]) {
                   ..........................................
                }
                for (l=0;l<3;l++) {
                    for (m=0;m<3;m++) {
                        K[temp1][l][m++];
                    }
                }
            }
        }
    }
}
```

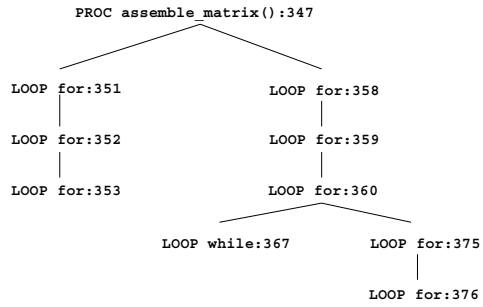


Figure 3.7: A fragment of the `assemble_matrix` module and its accompanying control flow hierarchy.

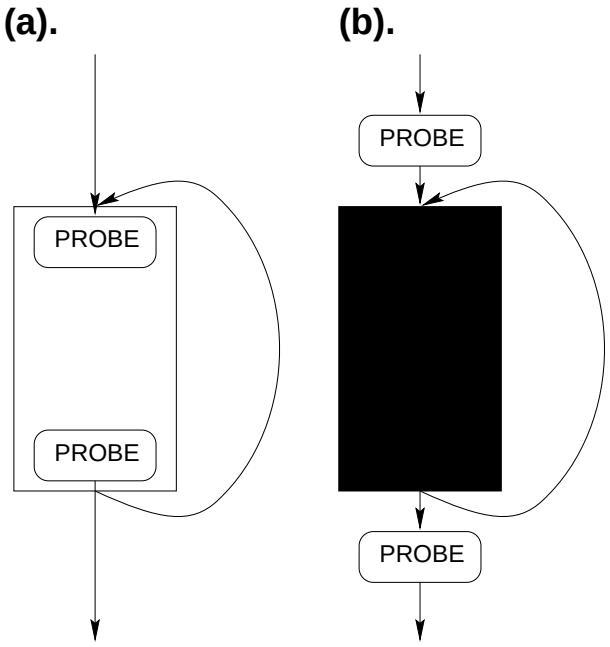


Figure 3.8: (a). Whitebox loop instrumentation collects instrumentation data for each iteration of the loop. (b). Blackbox loop instrumentation collects instrumentation data once, treating the loop as a single event.

values for all nested probes. For example, to factor out total noise for the program, we sum these values for all nodes in the CFH, and then subtract the resulting value from the total time stored at the root node. Figure-3.9 shows accumulated noise for the *Pyramid* adaptive mesh refinement library. This noise becomes insignificant when compared with measured data as Figure-3.10 shows.

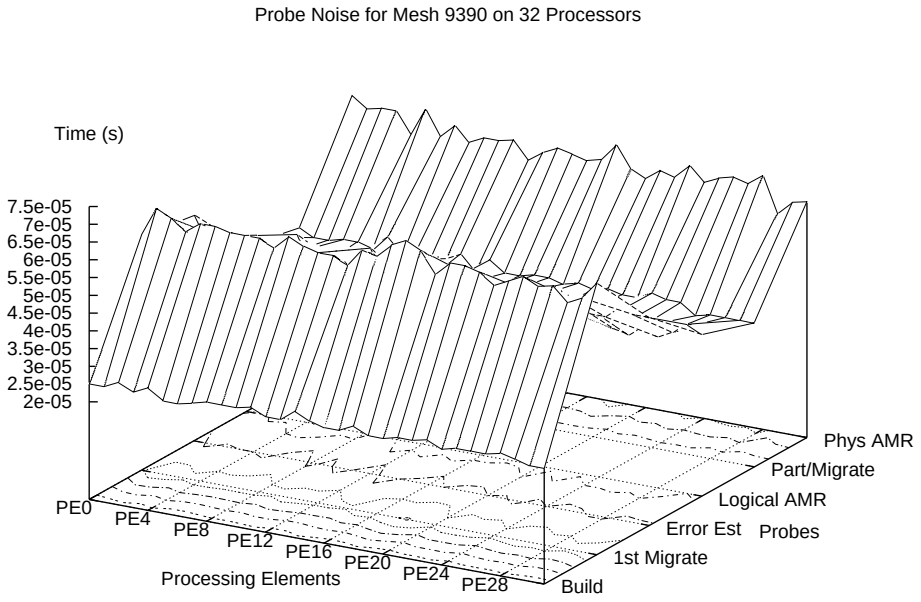


Figure 3.9: Measured overhead incurred by instrumenting the *Pyramid* library. x-axis is processing element, y-axis is probe, and the z-axis is the probe contribution to noise in seconds.

- **Selective instrumentation**. The best way to eliminate instrumentation overhead is to avoid instrumenting at all. There are many regions of a program that do not need to be instrumented because they are either provably optimal or not on the critical execution path. In other words, they do not significantly impact performance. Instrumentation can be selectively inserted in areas of interest. Figure-3.8 illustrates "blackening" out loops. Instrumenting trivial loops, like initialization loops, can contribute significantly to overhead. Blackening them out avoids unnecessary probing, thus reducing noise and di-

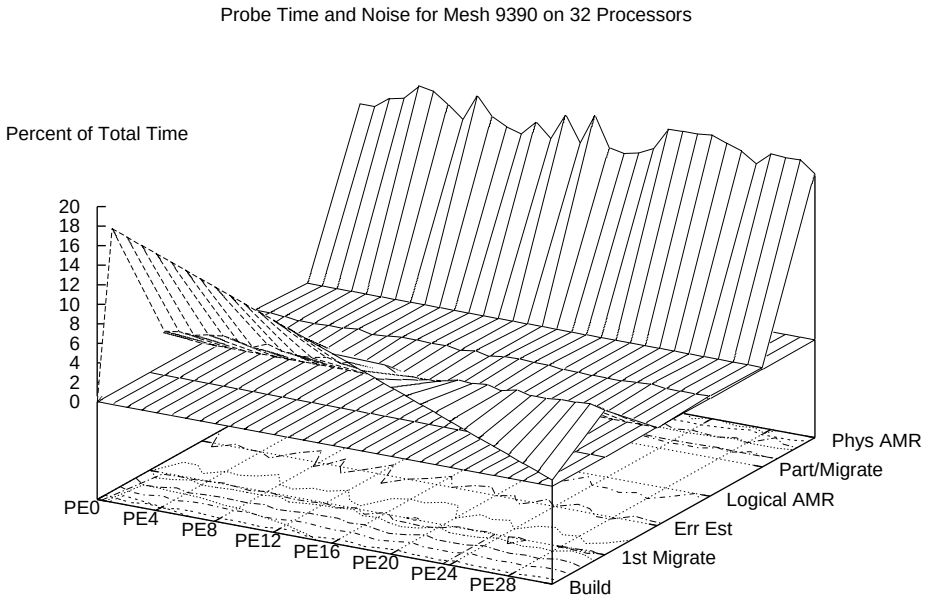


Figure 3.10: Measured overhead and execution time measured for *Pyramid* application. x-axis is processing element, y-axis is probe, and the z-axis is time measured in seconds.

lation.

Factoring out probe contribution to noise significantly improves accuracy. Dilation effects are minimized by introducing less instrumentation.

3.1.3.1 Cache Pollution

Introduction of instrumentation can alter the program in many ways. In addition to noise resulting from execution of additional instructions, the data that is acted on by these instructions has a significant effect on memory and cache utilization. Instrumentation data resides in cache at the expense of application data; the time spent retrieving application data from memory that would have otherwise been in cache is considered noise. This *pollution* of cache memory occurs at the start and completion of performance critical events that are being monitored.

Some architectures provide mechanisms where data can be explicitly excluded from the cache. Such mechanisms are not standard across multiple platforms, and

DATA	TYPE	DESCRIPTION
id	int	User defined ID
xid	int	Logical ID
name	string	Probe name
type	int	Probe type
t1, t2	double	Execution time stamps
w1, w2	double	Noise time stamps
c1, c2	double	CPU time stamps
cpu_acc	double	Accumulated CPU time
count	long	Probe activations
avg	double	Average time
avg^2	double	Square of average time
time	double	Measured time
min, max, avg	double	Min, max, and average time
cpu_min, cpu_max	double	Min and Max CPU time

Table 3.3: Probe Data

DATA	TYPE	DESCRIPTION
id	int	User defined ID
name	string	CFH name
w1, w2	double	Noise time stamps
num_probes	int	Number of probes in CFH

Table 3.4: CFH Data

hence are not POSIX compliant. It is important to minimize cache pollution by accessing instrumentation data in an intelligent manner. Cache pollution is introduced on probe START and STOP operations, roughly 188 and 156 bytes after basic compiler optimizations respectively. Typically, START operations are more costly on a probe's first activation. Tables 3.3 and 3.4 show which data items are frequently accessed. Noise resulting from cache pollution are less significant with scientific applications, as they exhibit a high degree of locality for long durations.

3.1.4 Support for Object-Oriented Codes

Object-oriented technology has significantly changed the way programs are developed. A corresponding change is needed in performance analysis of the resulting codes [54]. Sequential and parallel codes differ in the number of simultaneous

paths being traversed through the control flow graph. Purely CFG oriented views are insufficient for some object-oriented codes.

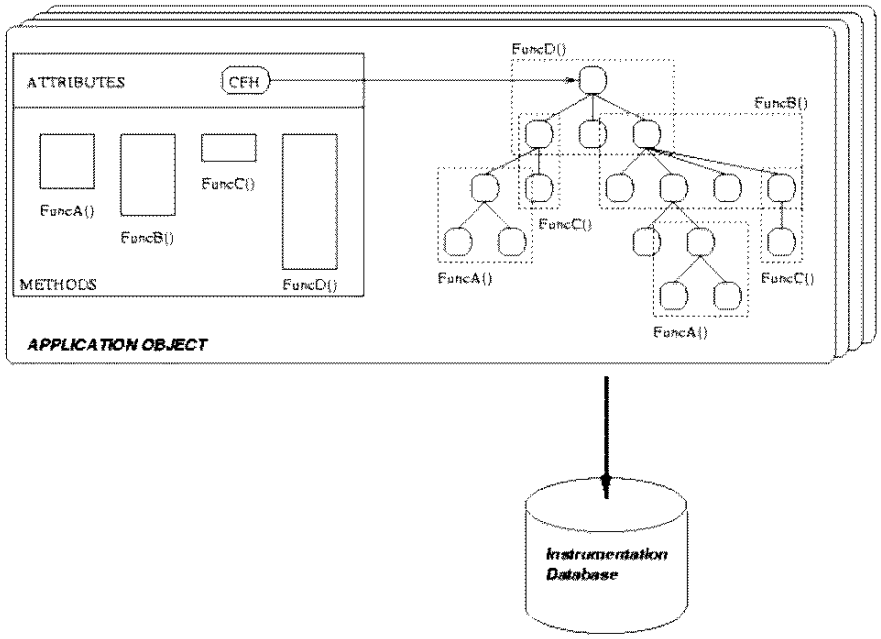


Figure 3.11: Each application object has its own CFH instance. The CFH is populated at run time by member functions.

Object-oriented programs can explore inter-object or intra-object parallelism. The former is based on task parallelism in which multiple objects are executing concurrently; the latter explores data parallelism by processing a single instance of an object running on multiple processors. Traditional control flow based techniques can be used to analyze performance of sequential and parallel member functions but do not extend to inter-object parallelism. To make such extension, we introduce the *object space* that includes all instantiated objects at a given time. Inter-object parallelism yields many control flow graphs representing simultaneous execution of member functions for objects in the object space. Some of these member functions may be running on multiple processors (data parallelism).

Extensions for support of task parallelism go beyond traversing multiple paths through one monolithic control flow graph; instead, we are confronted with multiple graphs executing in heterogeneous environments. To adequately capture task parallel applications, each object in the object space maintains its own CFH instance.

OPERATION	SYNTAX	DESCRIPTION
INIT	CFH local_cfh local_cfh.init(descr)	Instantiate a local CFH. Initialize CFH and description
CLOSE	local_cfh.close() local_cfh.dump(id) local_cfh.dumpall()	close active probes generate SQL /populate database with specific probes or dump all probes.
START	local_cfh.start(id, descr, me, nprocs)	Activate unique probe. If it exists in CFH start new counters else create new probe and start new counters.
STOP	local_cfh.stop(id, descr, me, nprocs)	Stop all probe counters and update statistics

Table 3.5: Object-Oriented API Syntax

Member functions populate the local CFH at run time, as shown in Figure-3.11.

3.1.4.1 Object-Oriented API

Instrumentation of object-oriented C++ codes involves introducing calls to the instrumentation database application programmer interface (IDB API). Performance critical objects have a local instance of a control flow hierarchy (CFH); calls to the CFH object either activate existing probes or create new probes. As the program executes, the control flow hierarchy is populated with probes and their respective statistics.

The API makes use of parameters such as processor ID and number of processors returned during MPI initialization so probes can be uniquely identified. CFH and probes contain description fields that map probes to specific source code constructs.

3.2 Language Interoperability

Improvements in compiler technology and programming languages give developers of parallel scientific applications greater flexibility in choosing programming languages and environments. The Message Passing Interface (MPI) provides support for C, C++, and FORTRAN codes. Initiatives are underway to make Java a viable

alternative for high performance computing. Java is widely used for distributed Internet applications outside of the scientific computing community.

Version 1.0 of the probe library (probelib), implemented in C, was designed to be compact and require minimal memory resources. In version 2.0, the library was re-architected to exploit many object-oriented features provided by C++. Moreover, the ability to support analysis of object-oriented codes did not require the introduction of elaborate data structures to capture object hierarchies. Instead, each CFH maintains a local *class variable* that maintains a unique identifier that is initialized on object instantiation. Thus leverages the existing object structure to store multiple instances of control flow hierarchies. Exploiting features of C++ greatly simplified the implementation of the 2.0 version of the probe API.[1].

3.2.1 C / C++ Interface

An interface is required to access the object-oriented API from C programs. The interface consists of a C++ function that creates a static instance of a CFH. The state of this CFH is preserved through multiple activations of the interface routine. In this way, an instance of the control flow hierarchy is global across all functions of target C application.

Figure 3.12 shows a C++ function that is callable from C programs. In this function, a CFH is statically instantiated. The interface executes appropriate 2.0 API calls using parameters and syntax conforming to the 1.0 API.

3.2.1.1 Support of Static Objects by *++-izing*

The 2.0 version of the probe API utilizes class variables for assigning unique identifiers to multiply instantiated control flow hierarchies. The interface between C functions and C++ methods, on some compilers, does not adequately support static objects. Each language has subtle differences in the way static data is stored by the compiler with respect to structures and objects. Since the interface method must maintain a statically defined instance of a control flow hierarchy to preserve state information, this limitation poses significant problems.

[1]The IDB probe library which implements the IDB API can be downloaded from `http://www.cs.rpi.edu/research/IDB`

```
extern "C" void IDB_probe(int op, int type, int id,
                          int myid, int p, char* s)
{
    static CFH local_cfh;
    static int mode = SQL;

    switch (op) {
        case INIT:
                local_cfh.init(s,myid,p);
                break;
        case START: local_cfh.start(type,id);
                local_cfh.name(id, s);
                break;
        case STOP: local_cfh.stop(id);
                break;
        case DUMP: local_cfh.dump(-1);
                break;
        case DUMPALL: local_cfh.dumpall();
                break;
    }
}
```

Figure 3.12: interface.cc: C interface function to IDB

We resolve this issue by *++-izing* the calling C program. This involves compiling all C code with a C compiler, but introducing a new main program that is compiled and linked with a C++ compiler. This ensures that static objects are handled correctly.

Figure-3.13 (a). shows the new main program that is run. The file contains a prototype for the main program to be analyzed that belongs to the target C application and defines a new main routine, which is linked with the code shown in Figure-3.13 (b), that calls and returns the main routine of the target C program. The file wrappedmain.c redefines the C program's main routine, which is called by the new C++ main program.

The advantage of *++-izing* an application in this way is that it requires no modification to the instrumented C application. The only modifications required of the user to *++-ize* an application are changes to the Makefile. The user is shielded from the complexity of the files in Figure-3.13. All that is required is to add *main.cc*

```
#include <iostream.h>

extern "C" int oldmain (int argc, char **argv);

int main (int argc, char **argv)
{
    return oldmain(argc, argv);
}
```

(a).**main.cc**

```
#define main oldmain
#include "application.c"
```

wrappedmain.c

Figure 3.13: *++-izing*

and wrappedmain.c to the proper Makefile.

3.2.2 FORTRAN 90 Support

IDB support, using the 1.0 version of the API, for FORTRAN 90 codes was developed to enable analysis of the *Pyramid* adaptive mesh refinement library being developed by the High Performance Computing Applications and Systems Group at the California Institute of Technology's Jet Propulsion Laboratory. Although version 2.0 of the API did not exist at the time of this analysis, the instrumented Pyramid library can use the newer API with little modification.

3.2.2.1 Pyramid Instrumentation and Analysis

The instrumentation API was extended such that probes can be introduced and the database can be initialized and closed from FORTRAN, as well as from C. In the 1.0 API, this was done by developing a FORTRAN 90 function whose interface is identical to the C function in Figure-3.12. This FORTRAN interface simply executes the main probe function (IDB_probe) which, in the 1.0 interface, is implemented in C. The 2.0 interface can be readily extended to support FORTRAN 90 in the same way.

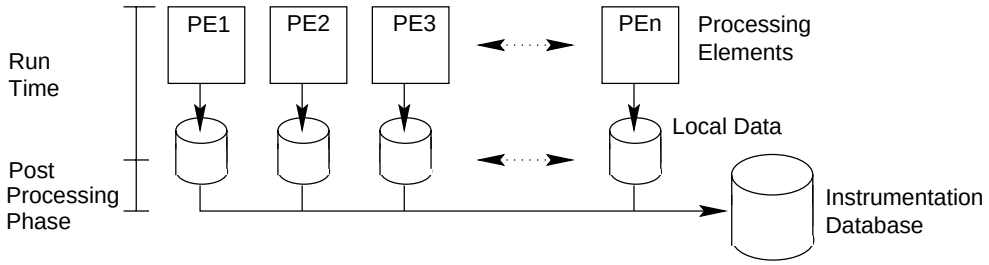


Figure 3.14: At run-time, data is collected on each processing element and integrated during a post-processing phase.

Probes include timing information specific to each processing element. Information collected on each node is stored separately and needs to be integrated into one database. Alternate implementations that we considered involved having each node communicate probe data to the first node for output, or each node write data to a single database in a round-robin scheme. Both these alternatives involved introducing synchronization delays to the executing program. Figure-3.14 shows the scheme that was implemented. Each node collects performance data locally at run-time. During the post-processing phase, these files are coalesced into one database. This is done implicitly in the 2.0 API, along with automatically generating SQL code that directly populates the database.

Prior to integration with the PostgreSQL DBMS, post-processing was done using PERL scripts that simulate database query operations and generate visualizations of collected data.

Figures- 3.15 and 3.16 show execution time for a 1978 element mesh running with 16 and 32 processors respectively. When the mesh data is read in, it is distributed to all processors. These graphs illustrate how moving from 16 to 32 nodes appears to induce a significant load imbalance in the `PhysicalAMR()` module. This imbalance is a result of the application's irregularity. Random distribution of the initial mesh was such that little or no refinement was required for mesh elements residing on processor 21.

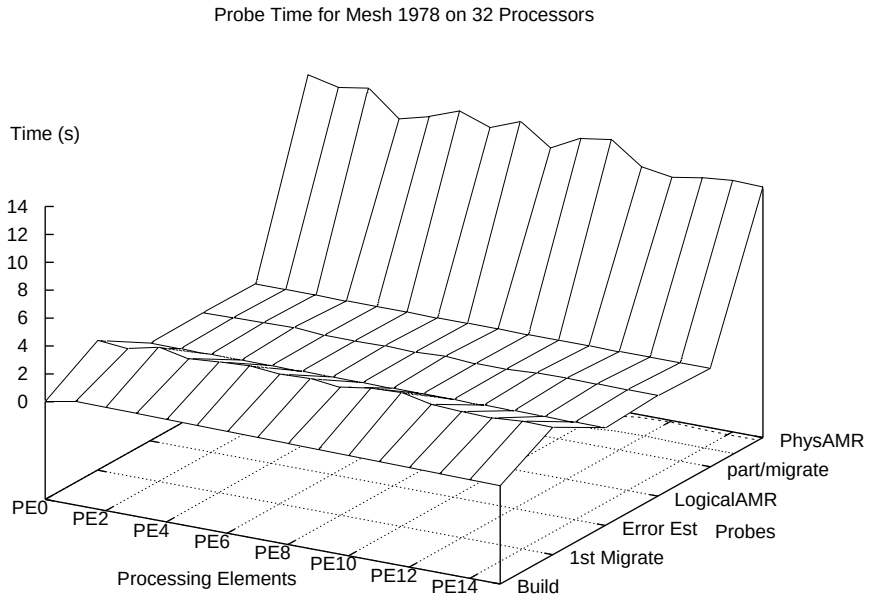


Figure 3.15: Probe times for 1978 element mesh on 16 processors.

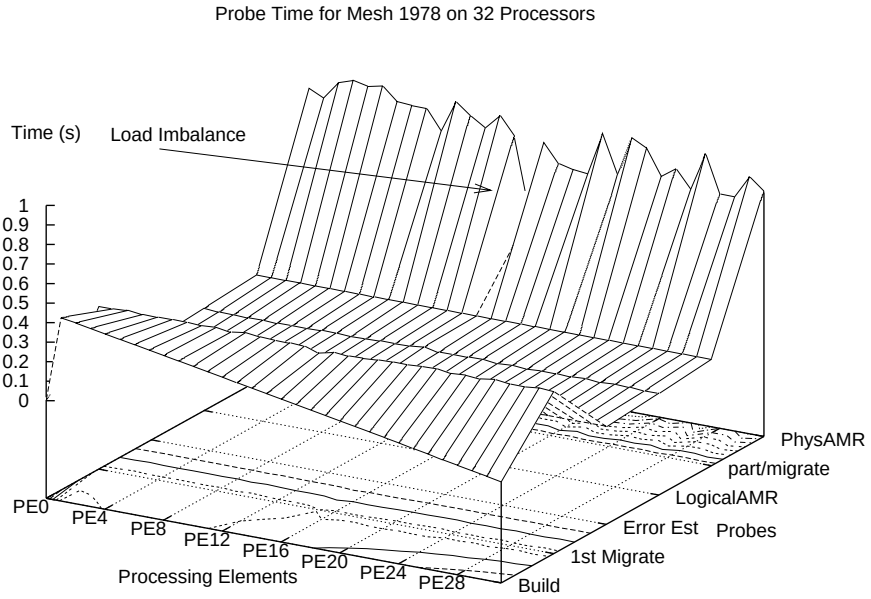


Figure 3.16: Probe times for 1978 element mesh on 32 processors.

PROCESSORS	IDB	PAT
8	80.0 s	80.3 s
16	20.8 s	20.6 s
32	8.4 s	8.4 s

Table 3.6: IDB versus PAT for 2430 mesh on SGI/CrayT3E.

PROCESSORS	PRISTINE	INSTRUMENTED	IDB
8	80.2 s	80.4 s	80.0 s
16	21.8 s	21.4 s	20.8 s
32	9.1 s	9.1 s	8.4 s

Table 3.7: Pristine and instrumented execution times with probe times on SGI/CrayT3E for mesh 2430. Run times measured using the `timex` command.

3.2.2.2 Verification and Dilation

To ensure accuracy of IDB probe data, instrumented and pristine versions of the application were analyzed using PAT. Table 3.6 shows the measured execution time of the instrumented application, along with the corresponding PAT instrumented version. In all cases, PAT and IDB differ by less than one percent. The decision to use PAT was based on the low overhead and accurate timing data relative to other tools on the SGI/CrayT3E.

To measure how probes dilate run time of the application, pristine and instrumented versions of the code were timed. Table 3.7 shows the run time for pristine and instrumented versions of the code. The last column shows the run time measured using IDB instrumentation. In most cases, instrumented code appeared to run faster than the pristine version. This is caused by the cache and small problem size: probe data remains in cache until it is accessed again; for larger problems this data may be displaced from the cache prior to its next access. In all cases, the measured time is less than the wall clock time. This is because the noise reduction factors out instrumentation and system overhead, whereas the `timex` command [1] does not.

3.2.3 Java Support

Performance analysis is vital in other domains outside of scientific and numerical computing. Danny Daglas has developed an interface using the *Java Native Interface* (JNI) [25] to instrument large distributed applications and database query tools used by financial markets and investment banks. Figure-3.17 shows the JNI wrappers of the version 2.0 IDB API calls.

3.3 Automated Instrumentation

Source code instrumentation typically has a high cost of invocation. Manually introducing instrumentation in the form of API calls that start and stop probes at performance critical locations is a tedious and time consuming process that is also prone to errors. To keep invocation cost low, an automated instrumentation tool is used to selectively instrument large codes and manage performance experiments. [8] [70]

The automated instrumentation tool (as shown in Figures- 3.18, 3.19, 3.20, and 3.21) is the product of a Master's project by Hui Wang [70] and an undergraduate independent study project by Jonathan Chen [8].

The user begins by selecting C or C++ files from the source tree using the file selection window shown in Figure-3.18. Selected files are then parsed using the tool's internal C/C++ parser. The instrumentation window shown in Figure-3.19 presents the user with several views:

- **Function Selection View.** Lists all functions and methods defined in the selected files. The first function in the list is the *main()* program. The user checks off the functions to be instrumented. When a function is selected, its source code is viewable and the other views are updated.

- **Loop Selection View.** Lists all loop structures within the currently selected function. The user checks off the loops to be instrumented. When a loop is selected, its contents can be viewed. In addition to turning instrumentation on and off, a probe can be designated as *internal* or *external*. Internal instrumentation of a loop involves placing the API probe calls within the body of

```
JNIEXPORT void JNICALL Java_idbbridge_IdbInit(JNIEnv *env, jclass obj,
                jint cfh_ptr, jstring s, jint myid, jint p) {
    CFH *x = (CFH *)cfh_array[cfh_ptr];
    const char *ptr = (*env).GetStringUTFChars(s, 0);
    x->init((char *)ptr, (int) myid, (int) p);
    (*env).ReleaseStringUTFChars(s, ptr);
}
```

(a) INIT implementation

```
JNIEXPORT void JNICALL Java_idbbridge_IdbStart(JNIEnv *env, jclass obj,
                jint cfh_ptr, jint type, jint id) {
    CFH *x = (CFH *)cfh_array[cfh_ptr]; x->start((int) type, (int) id);
}
```

(b) START implementation

```
JNIEXPORT void JNICALL Java_idbbridge_IdbName(JNIEnv *env, jclass obj,
                jint cfh_ptr, jint id, jstring s) {
    CFH *x = (CFH *)cfh_array[cfh_ptr];
    const char *ptr = (*env).GetStringUTFChars(s, 0);
    x->name((int) id, (char *) ptr);
    (*env).ReleaseStringUTFChars(s, ptr);
}
```

(c) NAME implementation

```
JNIEXPORT void JNICALL Java_idbbridge_IdbStop(JNIEnv *env, jclass obj,
                jint cfh_ptr, jint id) {
    CFH *x = (CFH *)cfh_array[cfh_ptr];    x->stop((int) id);
}
```

(d) STOP implementation

```
JNIEXPORT void JNICALL Java_idbbridge_IdbDump(JNIEnv *env, jclass obj,
                jint cfh_ptr, jint proc) {
    CFH *x = (CFH *)cfh_array[cfh_ptr];    x->dump((int) proc);
}
JNIEXPORT void JNICALL Java_idbbridge_IdbDumpall(JNIEnv *env, jclass obj,
                jint cfh_ptr) {
    CFH *x = (CFH *)cfh_array[cfh_ptr];    x->dumpall();
}
```

(e) DUMP and DUMPALL implementation

Figure 3.17: JNI Implementation of 2.0 API

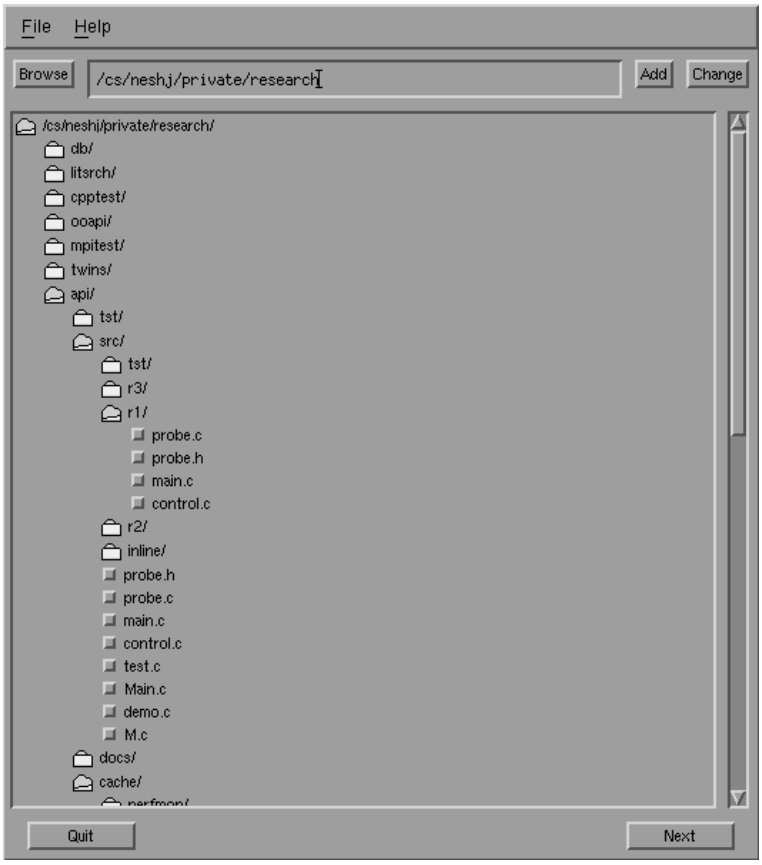


Figure 3.18: Automated instrumentation tool: File selection window. Specific source files can be selected for instrumentation from the source tree.

the loop. In this way, collected statistics account for each iteration. External instrumentation treats a loop as an atomic event. This avoids incurring overhead from probe activations on each iteration.

- **Call Selection View.** Lists all function calls within the currently selected function and loop. Instrumentation is added to the function calls selected by the user.

- **Source Code View.** Presents a context-sensitive view of application source code. This view can be constrained to individual functions or files. The tool provides rudimentary search functionality.

API calls are automatically introduced around selected events; the syntax of

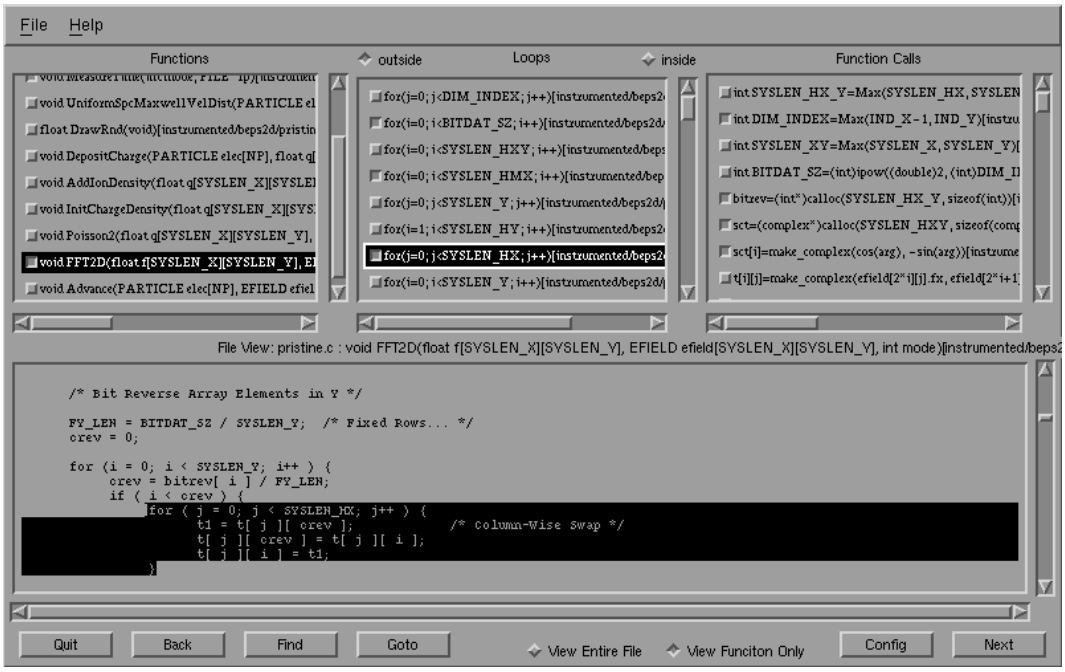


Figure 3.19: Automated instrumentation tool: Method and function instrumentation selection window. The user selects which performance critical event to instrument.

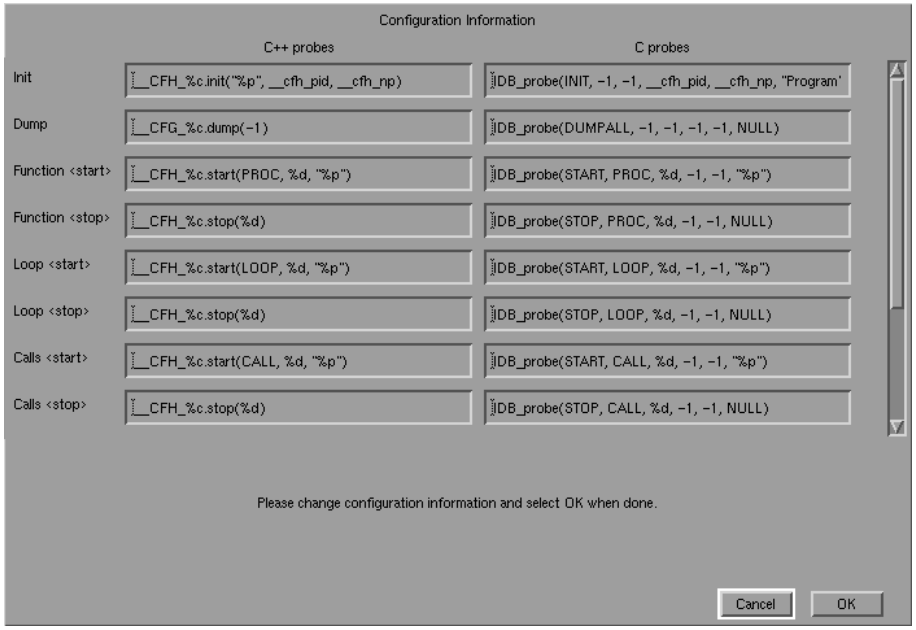


Figure 3.20: Automated instrumentation tool: API configuration window, where calls can be modified to accommodate minor changes in invocation syntax.

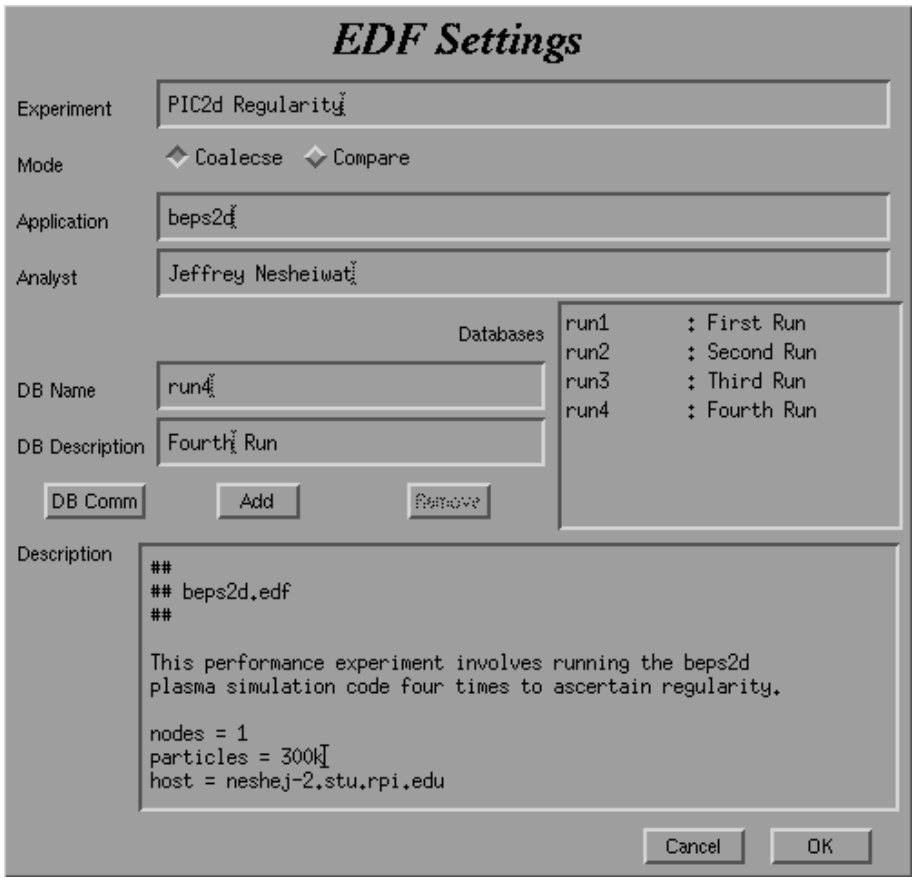


Figure 3.21: Automated instrumentation tool: Experiment definition window, where EDF files and instrumentation state information is saved and retrieved.

these calls can be modified from the configuration dialog, shown in Figure-3.20.

3.3.1 Experiment Definition Files

Conducting performance experiments is an iterative process. An iteration is defined by three steps:

- **Instrumentation.** Adding, deleting, or moving probes within the program such that events on the critical path are instrumented.
- **Systematic execution.** Permuting run-time parameters across multiple executions of the program.

- **Analysis.** Processing instrumentation data to localize bottlenecks, form hypotheses, and initiate optimizations to the code or run-time environment.

Experiment Definition Files (.edf) provide formal structure by centralizing information for conducting performance experiments. The automated instrumentation tool provides a mechanism where EDF files can be loaded and saved (see Figure-3.21). The information contained in an experiment definition file includes:

- **Experiment Name.** A short name describing the current experiment.
- **Analysis Mode.** Currently, two modes are supported: COALESCE, which merges performance data across multiple databases to gauge regularity and COMPARE which enables selective querying of individual probes across multiple databases for comparative analysis.
- **Application.** The name of the executable program.
- **Analyst.** The name of the person conducting the experiment.
- **Database list.** A list of databases and accompanying descriptions.
- **Database communications parameters.** The HOSTNAME, PORT NUMBER, USERNAME, and PASSWORD which are required for connecting to an instrumentation database on a remote server.
- **Description.** A free-form text description of the current experiment.

In addition to the user supplied information above, EDF files contain state information automatically generated by the tool describing where instrumentation was introduced. Specifically, it contains a snapshot of which functions, loops, and calls are selected for instrumentation. This allows the user to resume adding or removing probes between experiments.

3.3.2 Parser Issues

Adding instrumentation to a syntactically correct program is not a straightforward process. One problem is how to ensure that a probe is stopped before the function or program terminates. For example:

```
instrumented_function(int param1, int param2) {
    PROBE_START;
    stmt;
    stmt;
    if (test) {
        return;
    } else if (test2) {
        exit();
    }
    stmt;
    PROBE_STOP;
}
```

In the above code segment, if **test** is true, the function returns control to the caller without closing the current probe. Also, if **test** is false and **test2** is true, the program terminates without closing any open probes. This situation is handled in two ways. First, the function is parsed and any probes opened in the function are closed prior to any **return**. It is possible, however, for a program to terminate before all probes can be closed. This issue is handled from within the version 2.0 API by closing all active probes before termination.

Another minor issue involves adding instrumentation where multiple statements would semantically alter the program. For example:

```
if (test)
    foo(a);
else
    bar(b);
```

In this code segment, adding probes around the calls to **foo** would alter semantics.

```
if (test)
    PROBE_START;
foo(a);
```

```
PROBE_STOP;
else
    PROBE_START;
bar(b);
PROBE_STOP;
```

To avoid this problem, all probes of the type CALL are added such that the start and stop are considered one logical statement:

```
if (test)
    { PROBE_START, foo(a), PROBE_STOP};
else
    { PROBE_START, bar(a), PROBE_STOP};
```

3.3.3 Preprocessor Issues

The internal parser encounters difficulties when preprocessor directives are encountered. Currently, all code contained in `#ifdef` and `#if` blocks are parsed, therefore all segments blocked off by the preprocessor must yield syntactically valid code. The only exception to this is code contained in `#if 0` blocks; which we ignore. Instead of commenting out large segments of code, it is sometimes bracketed with `#if 0` and `#endif`; parsing such regions results in syntax errors for some of the codes that we consider.

Other issues stem from the parser's inability to expand macro definitions. Instrumentable code that is the result of expanding a macro definition is not visible to the internal parser. If a segment of source code depends on macro expansion to be syntactically correct, the parser returns errors. These are all limitations with the parser that can be addressed in future releases.

CHAPTER 4
Program Database

The term "database" in this context refers to a *Database Management System* (DBMS). Elmasri and Navathe define a DBMS as "a collection of programs that enables users to create and maintain a database ... that facilitates the process of defining, constructing, and manipulating databases for various applications." [17].

4.1 Derived Attributes and Comparative Analysis

Given the information in Table 4.1, simple queries can be issued to an instrumentation database to ascertain which function has the longest running time.

```
Q1 =  SELECT  procedure
      FROM    runs(Spark98, Nodes=1,
              Arch='Solaris_25', mesh='sf5.1.pack')
      WHERE   run_time = MAX(procedure.run_time)
```

This query returns the database tuples containing probe data corresponding to the control flow hierarchy (CFH) rooted at the `local_smvp` node, the function with the longest running time. This is the first step in a top-down analysis.

```
Q2 =  SELECT  Event
      FROM    Q1
      WHERE   run_time = MAX(event.run_time)
```

Function	Running Time
local_smvp()	50.37s
assemble_matrix()	23.21s
zero_vector()	00.69s

Table 4.1: Running times of selected Spark98 functions measured using `gprof`.

The next step is to find the bottleneck within this function. Q2 returns the node in `local_smvp`'s CFH with the largest running time.

This trivial example does not show the full analytic capabilities of our approach. However it does show how a relational *view* is formed using the `run()` portion of the FROM part in Q1. It also shows how metrics that we do not explicitly collect can be derived, or inferred, from instrumented data. In this example, the search for the function with the longest run time is restricted to sequential runs of the Spark98 kernel [56] for a specific architecture and mesh. These queries demonstrate how SQL syntax [48] can be extended to provide an easier interface to database contents. Analysis queries are dependent on the schema that defines how data is stored in the database.

4.2 Schema Design

The data stored in the instrumentation database fall into one of three categories:

1. **Static Data** associates program execution with static information such as: architecture, input vector, compiler, etc. This information provides the basis for experiment management and is specified in the *experiment definition files*.

2. **Probe Data** contains identity and statistical information about each probe, such as NAME, PROBE_ID, MIN, MAX, AVG, SDEV, TIME, COUNT, CPU, NOISE, etc. Each probe's data is of fixed size.

3. **CFH Data** defines how probes for a given run are related to each other. This hierarchy resembles the control flow graph of the program, but differs in that the only nodes represented correspond to CALL, PROC, LOOP, and COMM events. The size of this data is bound by the structure of the program.

A relational database is ideal for storing tabular information [17], such as the *Static Data* or the *Probe Table* as shown in Figure-4.1. However, relational databases are not convenient for storing graph based information, such as the CFH, in such a way that it can be efficiently queried. Object-oriented databases are particularly

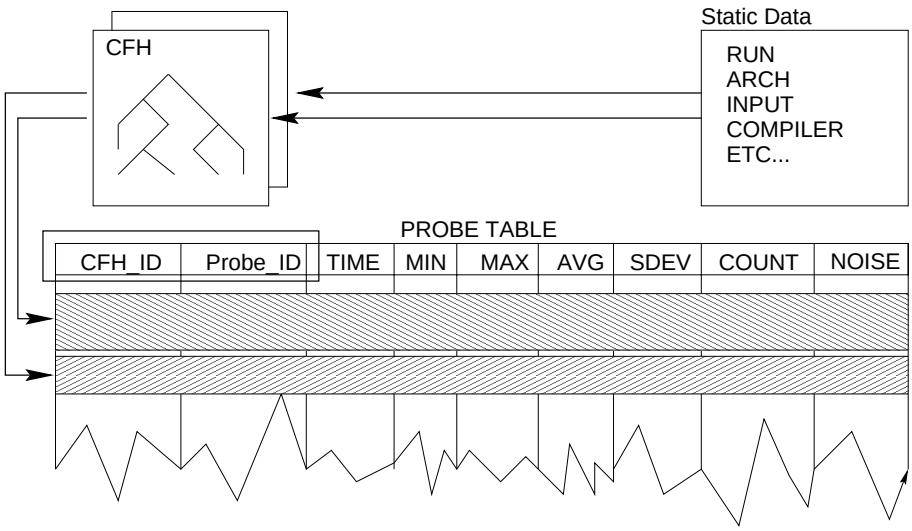


Figure 4.1: Static Data is associated with numerous Control Flow Hierarchies that represent multiple executions of the program. A CFH is associated with the Probe Table. The CFH provides parent and child information for each probe. The box around the first two entries in the Probe Table signify that the CFH_ID and the PROBE_ID together act as the primary key for indexing probes. This means that no entries in the database will have the same values for both attributes.

well suited to storing and querying graph data, but are not as efficient for querying statistical data [66]. PostgreSQL [2], a freely available and stable object-relation database, is used to capture control flow hierarchies, traditional statistical data, and static information. As a result, queries can be cast in terms of how probes are interconnected and across multiple control flow hierarchies. Leveraging database technology in this way provides a powerful framework for managing performance experiments independent of the underlying architecture and run-time environment.

4.3 Schema Extensions for Object-Oriented Support

The formal database schema was modified slightly to support data collection from object-oriented codes. Extensions to the probe API allowed for multiple instances of control flow hierarchies within one program execution. The version 2.0 API assigns a unique identifier to each CFH instance on a given processor. The database uses this identifier in conjunction with the processor ID to create a

composite key that uniquely identifies every CFH instance within the scope of the program. A CFH relation was added to the schema that contains a composite key along with a description attribute. The description attribute for both the CFH and probe tables map performance data to specific instances of an object and to actual program text, respectively.

4.4 Database Interface

PostgreSQL provides multiple interfaces to the instrumentation database; the simplest of these interfaces is the SQL shell (pgsql) which allows the user to submit queries interactively, view tables, and define functions. Appendix A shows some of these queries and functions that enable the user to "converse" with the database to extract obscure performance related information. Interfaces exist that allow queries to be submitted from numerous programming and scripting languages. Queries can be submitted to a local database server or over a network to a remote server.

4.5 Visualization

Once populated, the program database can be used to generate a wide variety of performance visualizations. Languages such as C, C++, Java, PERL, TCL, and others support graphical toolkits that include Motif, Tk, Qt, AWT/SWING, etc. SQL queries can be embedded within the code and executed using language specific database extensions (DBE). All front-end IDB visualization tools are functionally similar. The user graphically constructs a query which is then sent to the database server, where the results are graphically displayed.

The visualization tools presented in this section were the result of an independent study project completed by Simon Karpen [38]. Both tools were implemented using the PERL programming language, the Tk GUI toolkit and the DBE (Database extension) PERL modules with the PostgreSQL database interface (DBI). Languages and toolkits that were considered include:

- **Java with SWING toolkit.** This was considered for a number of reasons, including: (i). Code portability between Windows, UNIX, and Macintosh sys-

tems, (*ii*). Common look and feel across platforms, and (*iii*). Database interface using JDBC. In addition, Java bytecode does not require code to be recompiled when run on different platforms. The disadvantages include compatibility with some platforms and mostly poor performance.

- **C++ with Qt toolkit.** Advantages include good performance and a high quality graphical user interface (GUI). C++ and Qt were not used because the interface to the PostgreSQL database adds unneeded complexity. Moreover, the tool would have to be recompiled on multiple platforms. Another disadvantage is limited portability with the Windows platform due to Qt licensing constraints.

- **PERL with Tk toolkit.** This was chosen for several reasons, prominently because PERL is widely available, is easily installed on many platforms, and that it can be run on multiple platforms without recompiling. The PERL DBI provides a simple abstracted interface to the PostgreSQL database. However, there are some disadvantages to using PERL and Tk in that PERL/Tk is not as fast as C++/Qt, and the look and feel of the Tk interface is not as clean as the Qt interface. Furthermore, PERL code is considerably less readable than Java or C++.

The tools presented here illustrate how powerful visualization tools can be constructed using standard database interfaces.

4.5.1 Orbital Thermal Imaging Spectrometer

The following examples are derived from instrumentation collected from multiple executions of the Orbital Thermal Imaging Spectrometer (OTIS). OTIS is one of several applications used as part of the NASA Remote Exploration and Experimentation project [26]. OTIS is a parallel MPI code that computes atmospheric corrections and radiance calculations for orbital thermal emissivity data. Instrumentation was introduced using the automated instrumentation tool for function calls and loops in the main program.

4.5.2 Vistool

Vistool, short for Visualization Tool, connects to a populated instrumentation database (see Figure-4.2) and allows the user to construct a query by selecting attributes from a series of drop-down menu boxes (see Figure-4.3). These include:

- **Y-axis** that specifies which attribute should be plotted. Vistool currently supports Total_time, CPU_time, Range (max - min), and Standard Deviation.
- **Graph Type** that defines how data points should be represented on the graph. Scatter plots, histograms, and line graphs are currently supported.
- **CPU** that selects probe data for a specific processor.
- **Y Function** that defines if data are graphed on a linear or logarithmic scale.

The user can display multiple probe graphs at once. For example, in Figure-4.3, two graphs are superimposed; the bar graph shows the CPU time for probes executing on processor 4 and the line graph shows the total time for probes executing on the same CPU.

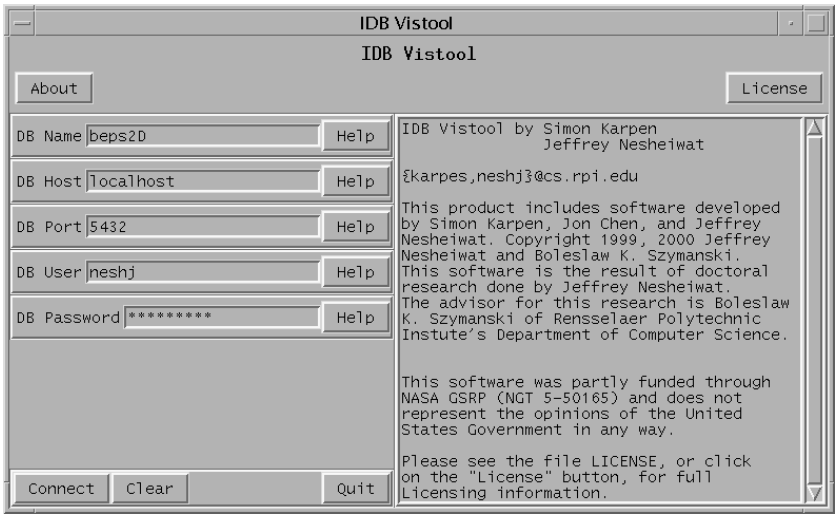


Figure 4.2: Vistool Connection Window. The user specifies the database name along with the host, port, and authentication information for the database server.

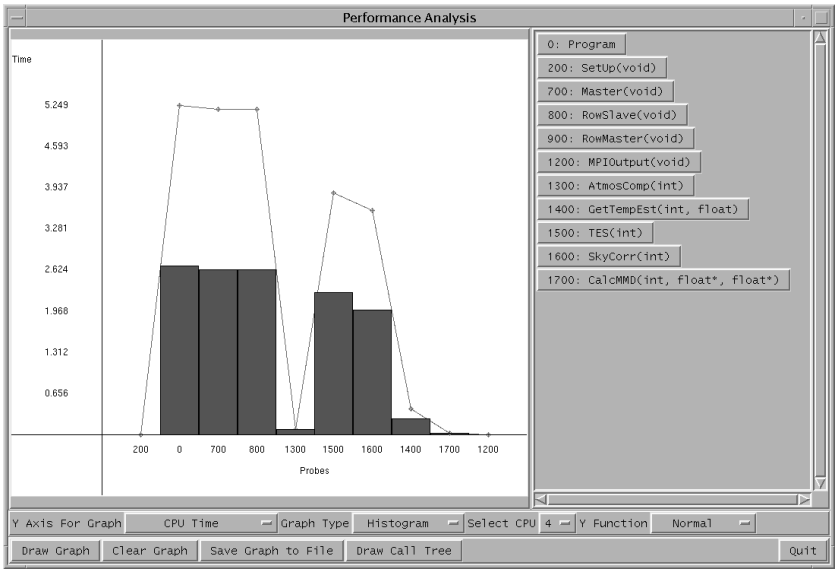


Figure 4.3: Vistool Main Window. The user selects graphing options from the toolbars at the bottom of the window. The legend on the right corresponds to active probes. Graph data is presented on the center canvas area.

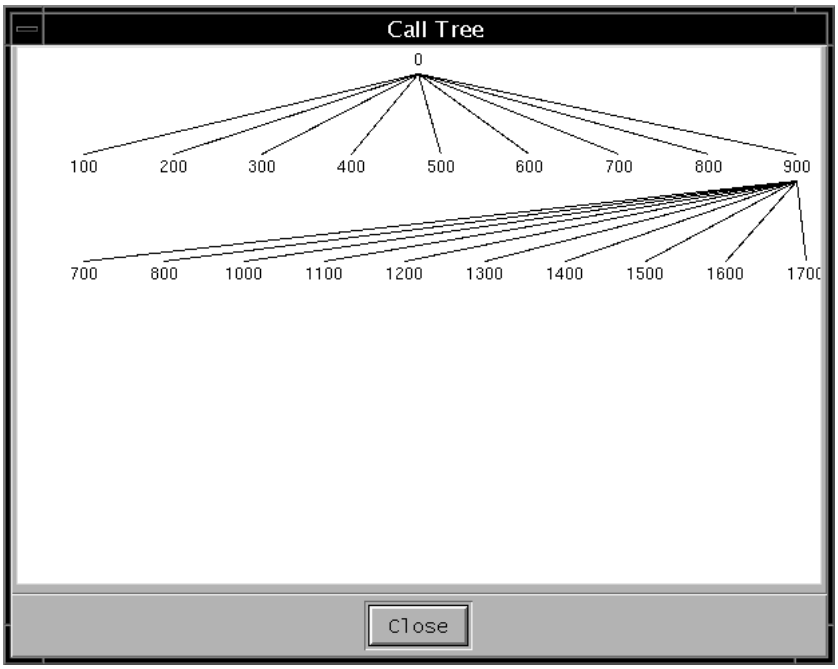


Figure 4.4: Vistool CFH View. Displays probe CFH connectivity.

Vistool supports multiple views. The first is the traditional graphing canvas, where scatter plots, histograms, and line charts can be displayed individually or in combination. The second is a tree view representing the selected probes in the control flow hierarchy, as shown in Figure-4.4. Lastly, Vistool provides a tabular view where the entire contents of a probe can be displayed. When the user clicks on a probe in the legend, the following query is sent to the database:

```
Q =  SELECT  *
     FROM    probes, lookup
     WHERE   probes.lid=lookup.lid AND name='PROBE_NAME'
```

where PROBE_NAME is the name of probe clicked on by the user.

4.5.3 EDFtool

The EDFtool, short for Experiment Definition File tool, parses experiment definition files and graphically displays query results from multiple program executions. The EDFtool supports two different experimental modes: COALESCE and COMPARE.

4.5.3.1 Coalesce Mode

The COALESCE experiment mode merges probe data from multiple executions of the same program. This is useful when trying to gauge regularity for an application in part or whole. Moreover, merging probes across multiple identical control flow hierarchies reduces noise by collecting multiple data points. This is especially useful when running benchmarks on multi-user systems or in *dusty deck* [34] environments.

EDFtool generates a concise visual representation of these datasets using box plots, or quartile charts, as shown in Figure-4.5 [51]. A box plot simultaneously describes multiple facets of a dataset including center, spread, departure from symmetry, and aberrant data points (outliers).

Figure-4.6 shows six executions of OTIS on 4 processors which were run at different times on a multi-user Beowulf [40] cluster. EDFtool read the experiment definition file in Figure-B.1. The user can confine visualization of the dataset to a

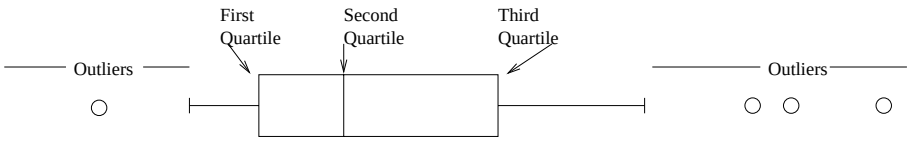


Figure 4.5: A box plot shows three quartiles in a rectangular box. The line inside the rectangle is drawn at the second quartile (50th percentile). The lines extending from the box, whiskers, show the smallest and largest data points within 1.5 quartile ranges. The points beyond the whiskers denote outlier data points.

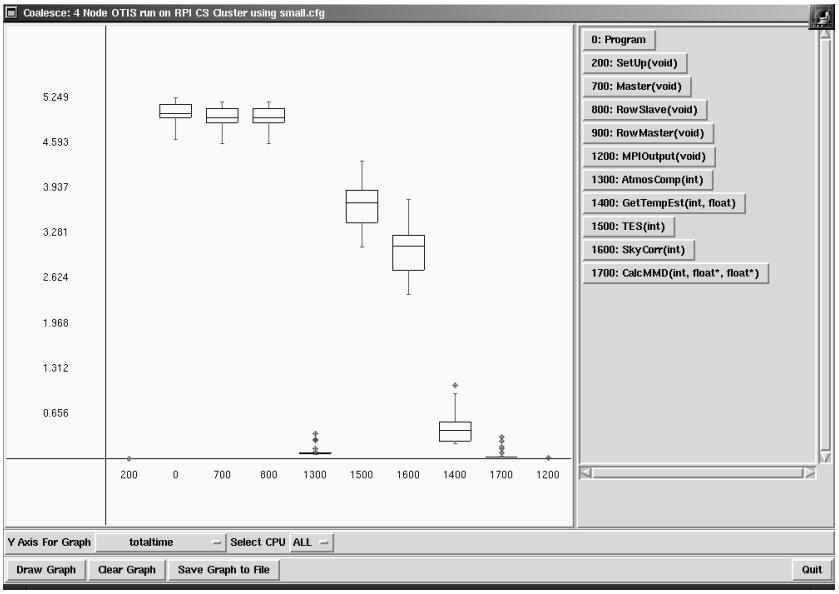


Figure 4.6: EDFtool coalesce mode. Quartile graph showing six 4 processor runs of an orbital thermal emissivity code (OTIS).

specific processor or to all processors. Clicking on a probe in the legend opens a tabular view corresponding to that probe.

4.5.3.2 Compare Mode

The COMPARE experiment mode supports analysis across multiple control flow hierarchies where compile time and run time parameters differ. In this mode, the user selects the probe to be graphed from a drop-down list. The value for the selected probe is plotted for each experiment listed in the EDF.

OTIS can be modified using a configuration file that is read in at run time.

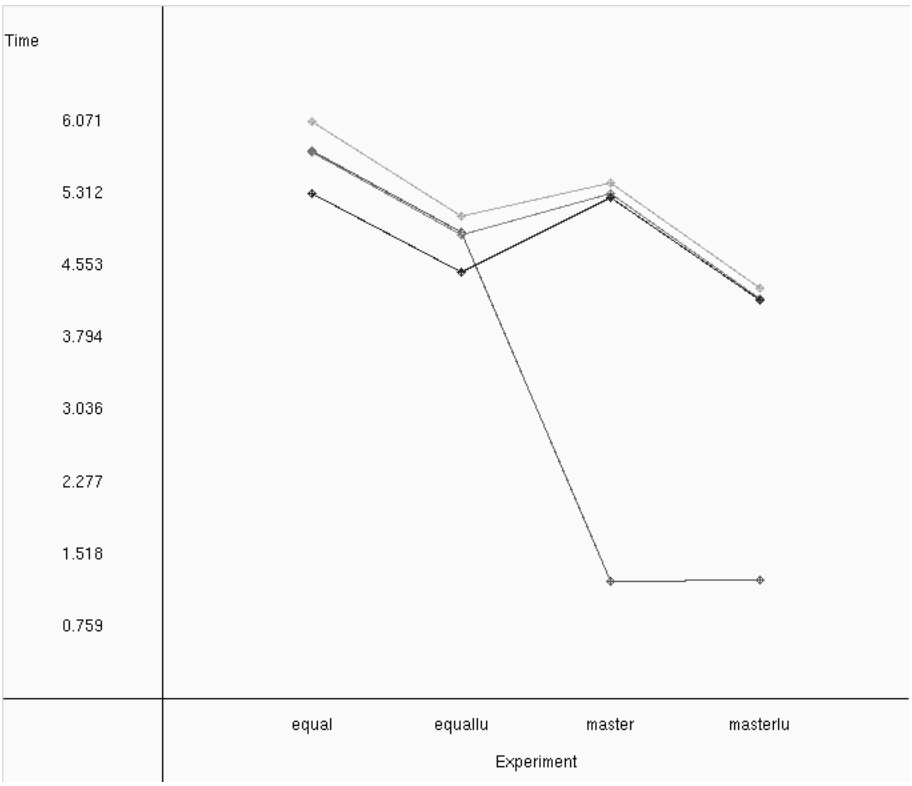


Figure 4.7: EDFtool compare mode. Each line depicts the CPU time for each of the 4 processors.

Figure-4.7 shows the results from the experiment definition file specified in Figure-B.2. Two configuration parameters were permuted:

- Master/Slave versus Equal Distribution
- Lookup tables on and off

The graph shows CPU time for each of the four OTIS runs. The CPU time for processor 0 is significantly lower for both master/slave runs. Processor 0, or the master process, is responsible for collating results, hence does less work. In both cases, the introduction of lookup tables improves performance.

This example illustrates how COMPARE mode can be used to measure how changes effect overall performance. The tool allows the user to select specific probes, processor, or graph type. Graph types include scatter plots, line charts, and histograms. As with Vistool, multiple graphs can be superimposed.

4.6 Emergent Methodology

The introduction of IDB and accompanying tools presents the user with a framework and an evolving methodology for conducting performance experiments. This methodology involves introducing probes at a high level and iteratively adding and removing probes such that performance events along the critical path are instrumented. The resulting code is defined as the *reference instrumentation* which is then used to populate the instrumentation database for the purposes of:

- General assessment.
- Determining quality of load balancing / parallelism.
- Assessing performance effects of migration to other architectures.
- Comparing optimizations applied to multiple versions of the code.
- Gauging performance regularity across multiple runs.

The methodology defines an experimental approach to localizing bottlenecks and introducing optimizations by providing support for the W^3 search model and providing multiple experiment modes, respectively. We demonstrate this in the following chapters where introduction of IDB to a diverse set of parallel scientific codes yields commonalities in how bottlenecks are identified and effectiveness of optimizations are measured.

CHAPTER 5
Particle in Cell Simulation Code

In this example, we demonstrate how IDB instrumentation can be applied to a parallel object oriented application. Probes are introduced using the version 2.0 API; we show how the underlying object structure is affected and how basic performance experimentation is possible using this approach. Our goals are to show that the IDB approach is scalable and to demonstrate how it is used to analyze object oriented codes.

5.1 Background

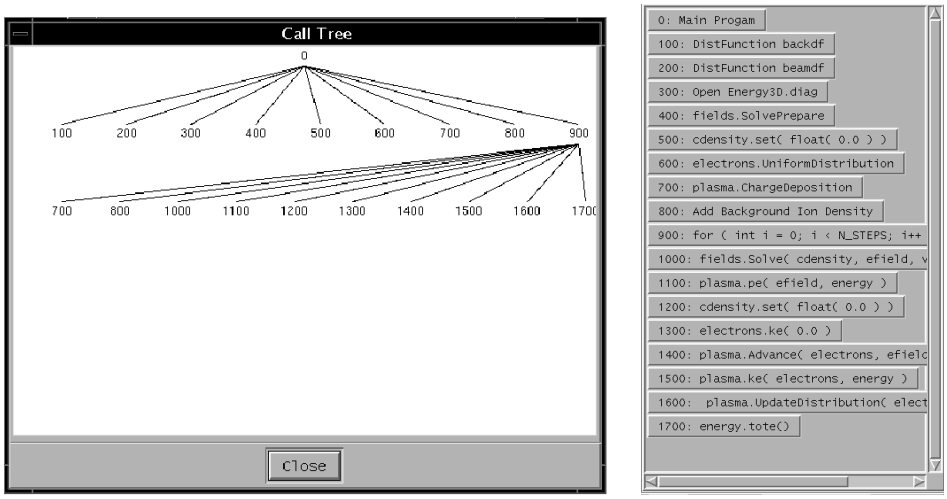
Particle in cell, or PIC, codes [4] [71] are used to simulate spatial non-linear kinetic systems. The PIC code discussed here simulates plasma flow by modeling the plasma as millions of particles in a self-consistent electro-magnetic field. Each time-step of the computation consists of two phases:

- **Particle push.** Particle positions are updated and their charge / current density are computed. This is done with the following steps:
  - *Gather:* Interpolate fields from grid points to particles.
  - *Local Step:* Compute the new position of each particle; this entails local computation involving no interprocessor communications.
  - *Scatter:* Deposit charge/current from particles to grid points.
- **Field solve.** Electromagnetic field is recomputed based on new charge/current in grid points.

The size of the simulations, number of particles, domain size, etc. are constrained by available memory and processors. The domain is decomposed spatially in one, two, or three dimensions.

5.2 Instrumentation

This 3D PIC code [54] is implemented in C++ and runs on 2^k processors on an IBM SP2. Simulations were run using 4K, 32K, and 8M particles on 4, 8, 16, and 32 nodes. Minimal instrumentation was initially added to characterize scalability of the application with respect to problem size and number of processors. Later, probes were added incrementally along the critical path to localize potential bottlenecks. To this end, all performance critical events (CALL, PROC, LOOP, and COMM) were instrumented in the `main()` function and main event loop. Figure-5.1a shows the resulting control flow hierarchy (CFH) for this *reference instrumentation*. The root of the tree, node 0, corresponds to the main program; node 900 represents the main event loop of the program (as shown in Figure-5.1b).



(a). CFH for 3D PIC code. **(b). Probe legend.**

Figure 5.1: Instrumentation applied to main program and main event loop.

Instrumentation was realized using the automated instrumentation tool which instantiates a CFH object globally. All probe START and STOP operations call public methods on the CFH object to add or update statistical data to the CFH. Figure-5.6 shows how the CFH object and associated instrumentation fit into the overall UML diagram for the program. IDB instrumentation introduced the following objects:

- Control Flow Hierarchy (CFH)
- IDB_Probe
- IDB_Stack
- IDB_List

API calls introduced in the code invoke methods on the CFH object. The CFH object in turn instantiates probes as needed and invokes start and stop methods on them. The IDB_Stack is used by the CFH object to maintain active, or open, probes. When a probe operation is started, its ID is pushed onto the stack; on a probe stop, its ID is popped from the stack. The IDB_list is used by the probe objects to store connection information used by the instrumentation database to construct parent/child relationships between probes. The CFH object is not directly connected to any application objects in the UML diagram because it is instantiated and accessed globally. This PIC simulation code exhibits data parallelism thereby requiring only one instance of a CFH.

5.3 Analysis

The program was run repeatedly for different problem sizes and number of processors to show scalability. Table-5.1 shows that the code is scalable with 99% efficiency. In addition to showing scalability of the code, we observed that the size of the instrumentation database was the same (56 Kb) for each run. In addition to showing scalability, IDB probes are used to identify where optimization efforts should be focused. The following SQL query yields the contents of Table-5.2, which is represented graphically by Figure-5.2.

```
Q =  SELECT DISTINCT   probes.lid, name, count, cpu_acc,
                      (avg*count) AS total_time
     FROM             probes, lookup
     WHERE            probes.lid = lookup.lid AND proc = 0;
```

Particles	Processors	Time
4k	4	10.90
4k	8	6.91
4k	16	N/A
32k	4	180.22
32k	8	88.17
32k	16	48.07
8M	8	2902.14
8M	16	1288.23
8M	32	661.65

Table 5.1: Execution times, averaged across three runs, of 3D PIC code on an IBM SP2. 4K elements distributed on 16 processors is not shown because the problem size is too small to run on more than 8 nodes.

ID	Name	Count	CPU Time	Total Time
0	MAIN PROGRAM	1	694.13	694.233024
100	DistFunction backdf	1	0	2.5e-05
200	DistFunction beamdf	1	0	2.1e-05
300	Open Energy3D.diag	1	0	0.030342
400	fields.SolvePrepare	1	0	0.000647
500	cdensity.set(float(0.0))	1	0.01	0.000324
600	electrons.UniformDistribution	2	0.92	0.921092
700	plasma.ChargeDeposition	426	30.13	30.093492
800	Add Background Ion Density	426	0.06	0.048138
900	for (int i = 0; i < N_STEPS; i++)	425	693.07	693.167775
1000	fields.Solve(cdensity, efield, vpm)	425	15.79	15.82785
1100	plasma.pe(efield, energy)	425	0	0.011475
1200	cdensity.set(float(0.0))	425	0.1	0.07395
1300	electrons.ke(0.0)	425	0	0.008925
1400	plasma.Advance(electrons,efield,DT)	425	642.17	642.06025
1500	plasma.ke(electrons, energy)	425	0.01	0.0102
1600	plasma.UpdateDistrib(electrons,vpm)	425	4.59	4.73535
1700	energy.tote()	425	0	0.00935
2000	for (register int i=0;i<spec.npp;i++)	15664998	244.8	360.294954

Table 5.2: Probe table for CPU 0 of 3D PIC code.

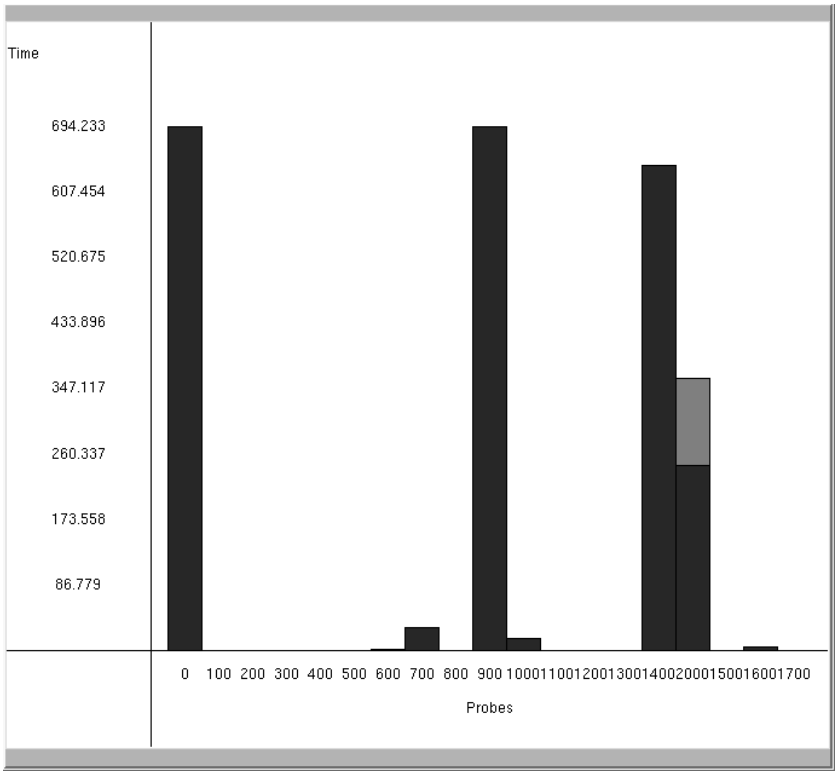


Figure 5.2: Total time and CPU time for 3D PIC code.

They show total time and CPU time measured on an 8 processor run. Probe 900, the main loop, consumes 99.4% of the program's execution time. Further, the `plasma_advance()` function consumes 92.6% of the loop's execution time.

Within `plasma_advance()`, probe 2000 is introduced to instrument its main loop, shown in Figure-5.3; we see that it consumes over one third of the program's total execution time. Closer inspection of the data in Table-5.2 shows that the loop iterates approximately 10^7 times, with a significant difference between the total time and the CPU time. This is where optimization efforts are to be focused.

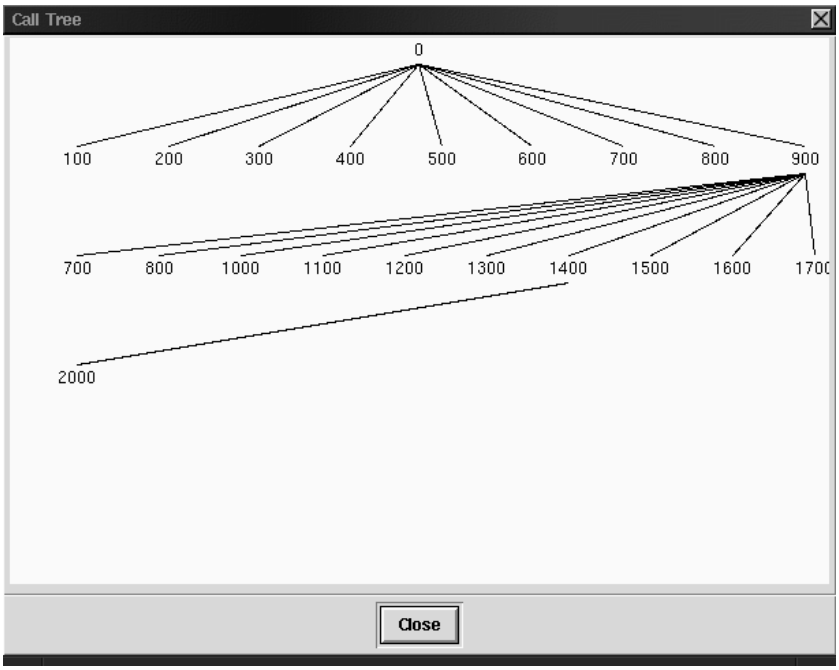


Figure 5.3: Control Flow Hierarchy for 3D PIC code with additional probe added.

```
dxp = 0.5 * (pow( (0.5+dx), 2));
```

(a). Original code.

```
dxp = 0.5 * (0.5 + dx) * (0.5 + dx);
```

(b). Modified code.

Figure 5.4: Optimization introduced to `plasma_advance()` function.

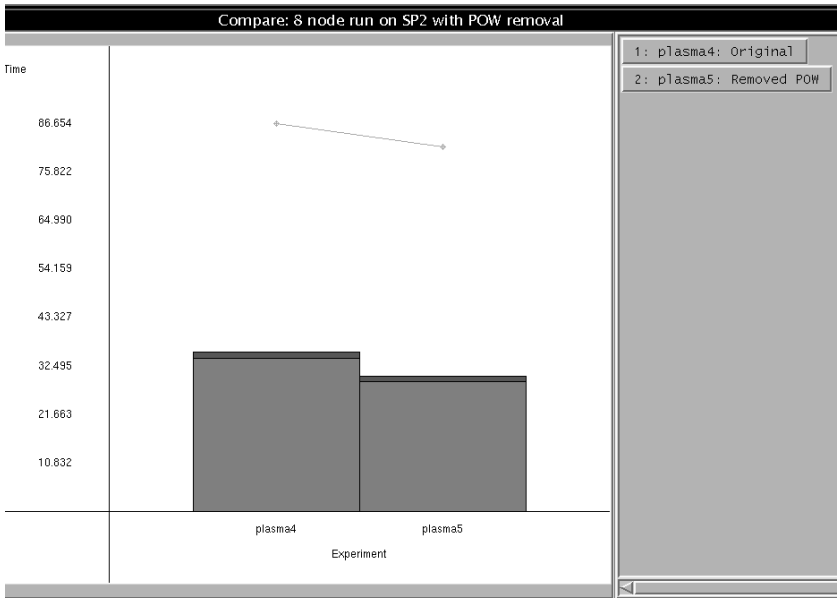


Figure 5.5: Comparison of 3D PIC execution with `pow()` function calls removed. The line shows total execution time of the program, the dark band shows total execution time of `plasma_advance()` function, and the lighter bar shows the execution time of its inner loop.

Examination of the source code shows that within the body of this loop, the `pow()` function is used to square simple mathematical expressions, as shown in Figure-5.4a. This operation was originally introduced because it simplifies using more advanced interpolation schemes and was significantly less costly in the earlier FORTRAN implementation of this code. This occurs 10 times within the loop. The small modification in Figure-5.4b eliminates approximately 10^8 calls to `pow()`, squaring the expression in-line.

This modification resulted in an 18.1% speedup in the `plasma_advance()` function, or a 6.1% speedup in total execution time. Figure-5.5 shows this graphically, using the experiment definition file in Figure-B.3. Amdahl's law states that the fastest speedup that can be obtained by optimizing `plasma_advance()` is 33%. It follows that overall speedup is $\frac{1}{3}$ of the function speedup.

We have demonstrated that the introduction of a globally accessible CFH object in the object model of this PIC simulation is an effective means to collect scalable instrumentation from multiple objects and routines. We have also shown IDB to be scalable with respect to the size of the problem.

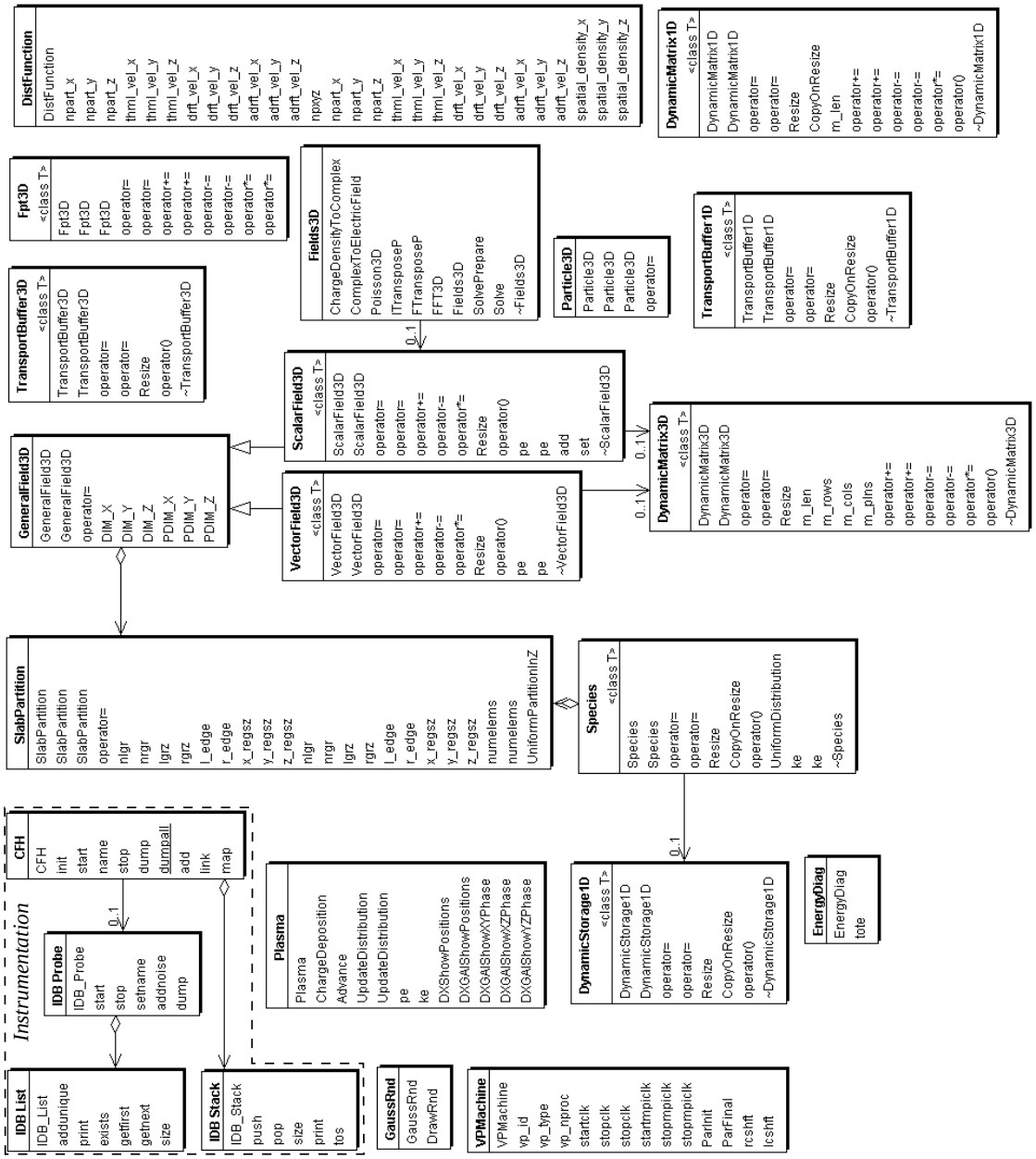


Figure 5.6: UML Diagram of the instrumented PIC code. The IDB objects are contained in the box in the upper left corner.

CHAPTER 6
Quantum Device Simulation Tool

This example further demonstrates the use of the instrumentation database approach. Specifically, we explore aspects of the emergent methodology associated with the IDB instrumentation and visualization tools. We also demonstrate how the framework for experiment management, provided by IDB, is used to improve code performance. Our goal is to apply this methodology to compare multiple optimized versions of the same application.

6.1 Background

Nanoelectronic Modeler, or NEMO, is a comprehensive quantum device simulator being developed at the High Performance Computing Systems and Applications group at NASA's Jet Propulsion Laboratory under the direction of Dr. Gerhard Klimmeck [6] [41]. NEMO simulates numerous quantum devices including: RTD, HBT, HEMT, MOS, Esaki diodes, and Super-lattices. The simulator provides an extensive graphical front-end through which the user controls many aspects of the simulation, and can view results graphically. Some features include:

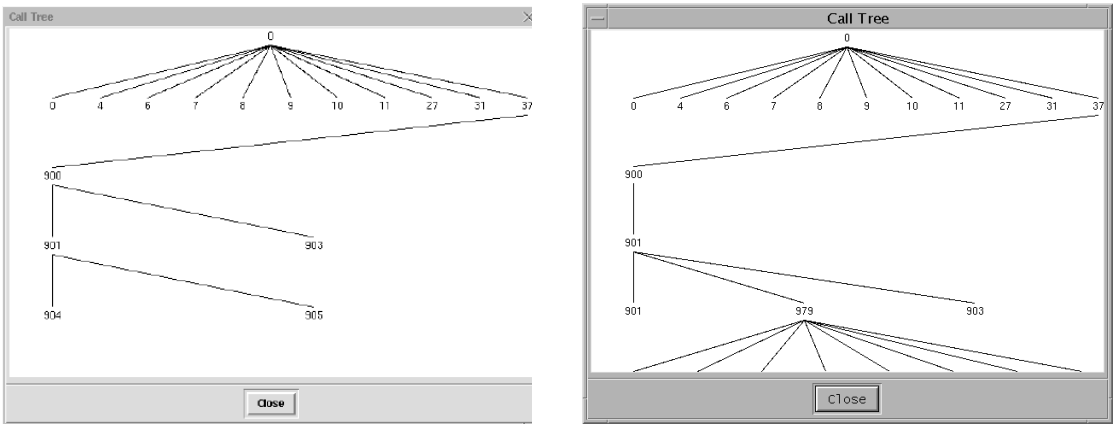
- Graphical control of all device, material, and simulation parameters.
- Default physical values for materials provided.
- 2D, 3D, and contour plots of calculation results.
- "On the fly" band profile calculations.
- *Plot Slicer* to display slices of 3D data sets.
- Library of sample device simulations.
- Graphical band structure tool to visualize energy band and electron density vs. Fermi energy.

- Graphical material properties tool.

NEMO's extensive graphical interface provides the modeler with visual input and presents the user with multiple views of simulation results.

6.2 Instrumentation

IDB probes were introduced to NEMO's computationally intensive modeler and run repeatedly on a 12-node Beowulf cluster [40].



(a). Intermediate Control Flow Hierarchy (b). Reference Control Flow Hierarchy

Figure 6.1: CFH showing instrumentation added to NEMO in an iterative fashion. Probes are added such that the CFH is grown until likely bottlenecks can be exposed.

Instrumentation was introduced iteratively; Figures- 6.1a and 6.1b show two iterations of probe addition. The following query yields the data in Table-6.1 which maps these probes to specific regions of the source code.

```
Q =  SELECT DISTINCT   probes.lid, name
     FROM             probes, lookup;
```

The need to instrument further becomes obvious when we consider the graph in Figure-6.2, which shows that the probes along the critical path consume the majority of execution time. In fact, the function lanc_driver_c_par() consumes 90% of the total run time for the modeler. Additional instrumentation is introduced in this function, represented by the CFH in Figure-6.1b, and yields the graph in

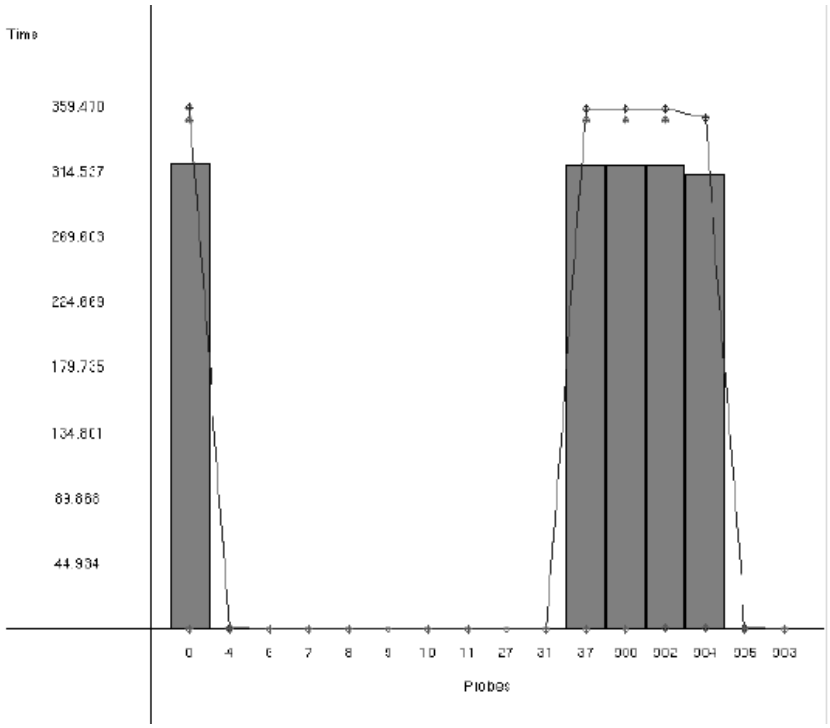


Figure 6.2: Intermediate profile of execution times.

Figure-6.3. This new instrumented version of NEMO is the *reference version* used during optimization.

6.3 Analysis

To demonstrate the experiment management capabilities of IDB, the function `h3d_offdiag()` is the focus of optimization efforts. Initially, we observe that it consumes approximately $\frac{1}{3}$ of the main loop time.

Closer inspection of the `h3d_offdiag()` function shows that it is invoked 8,325,611 times. Moreover, a 20% difference between CPU and total execution time for this function make it a likely candidate for further optimization. Three modifications were made to the instrumented function:

- **Static version.** All local data was re-declared as `static`.
- **Enumerated type macro version.** All enumerated types were re-defined as macros.

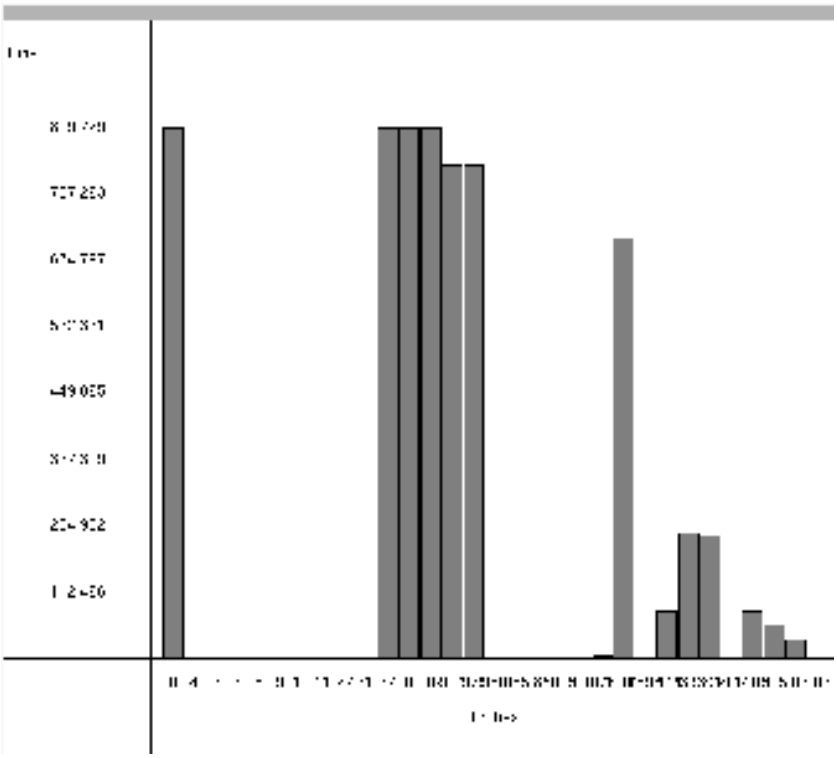


Figure 6.3: Execution times of reference version.

- **In-line version.** The function `h3d_offdiag()` was re-defined as an in-line function.

Figure-6.4 shows the visualization specified in the experiment definition file (EDF) in Figure-B.5. The optimizations resulted in a 25% speedup in `h3d_offdiag()` and an 8.1% total speedup.

This example illustrates how IDB can be used iteratively to (*i*). converge on a reference version of instrumented code, and (*ii*). design and compare performance experiments and results derived from multiple versions of the code. Figures- 6.5 and 6.6 show examples of typical IDB sessions.

Probe Description	ID
hline Program	0
argv_process(args, argc, argv)[run3d.c:38]	4
d=make_qd_struct()[run3d.c:45]	6
Set_QD_Global(d)[run3d.c:47]	7
inputfile=copy_str(CmdOptions.input)[run3d.c:50]	8
i_read(inputfile, d, &Abort_result)[run3d.c:55]	9
i_save_shell(d, d-inputfile, 1, &Abort_result)[run3d.c:57]	10
mat_param_init(d)[run3d.c:62]	11
ham_init_spds(Shape, d)[run3d.c:155]	27
ham_init_final_spds(d)[run3d.c:176]	31
lanc_driver_c_par(d)[run3d.c:213]	37
lanc_driver_c_par(qd_struct d) [eig3d_par.c:730)	900
do-while loop in non master ID waiting for no convergence	902
Sort converged eigenvals and dump to file	903
sym_lanc_it_c_par()	904
for (i=0;i<it;i++)	905
for (j=0;i<n_old;j++)	906
Beta	975
int sym_lanc_it_c_par()	979
for (i=0; i < dnproc; i++)	980
for (target_i=0; target_i<d-nproc; target_i++)	985
for (target_i=0; target_i<d-nproc; target_i++)	988
PI_Allreduce(&sum_local,&sum, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);	990
colpar	1001
send-rcv	1003
local mem	1005
local mem2	1007
h3d_diag(d-h, d-basisa, d-spin, d-param, d-nb)[eigsys3d.c:959]	1111
int h_mult_spds_col_par(cvectr y, cvectr yc, ivectr ycmap, qd_struct d, real s,	1898
h3d_offdiag(d-h, nnv, d-basisc, d-basisa, d-spin, d-parmat[d-atom[l][m]][d	2222
cmatmul_spds()	3333

Table 6.1: Probes that comprise the reference instrumentation of NEMO.

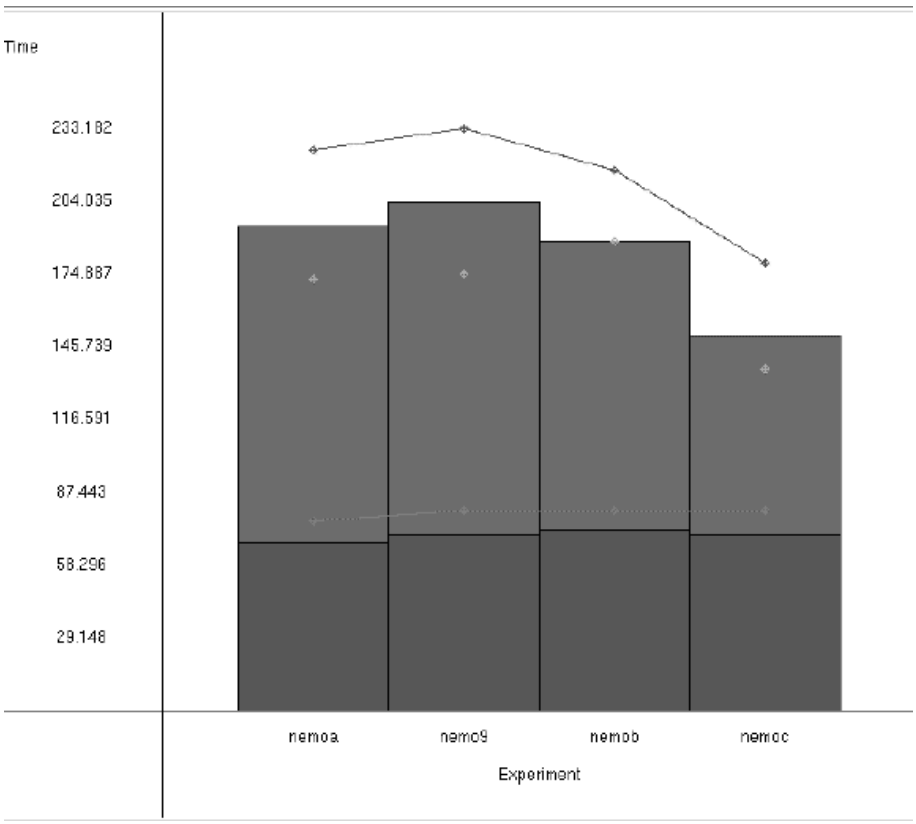


Figure 6.4: The lighter bar indicates the CPU time of the `h3d_offdiag()` function. The line above represents Total time. The darker bar and its associated line represent the `h3d_diag()` function. This is to determine if modifications to `h3d_offdiag()` change execution time of `h3d_diag()`.

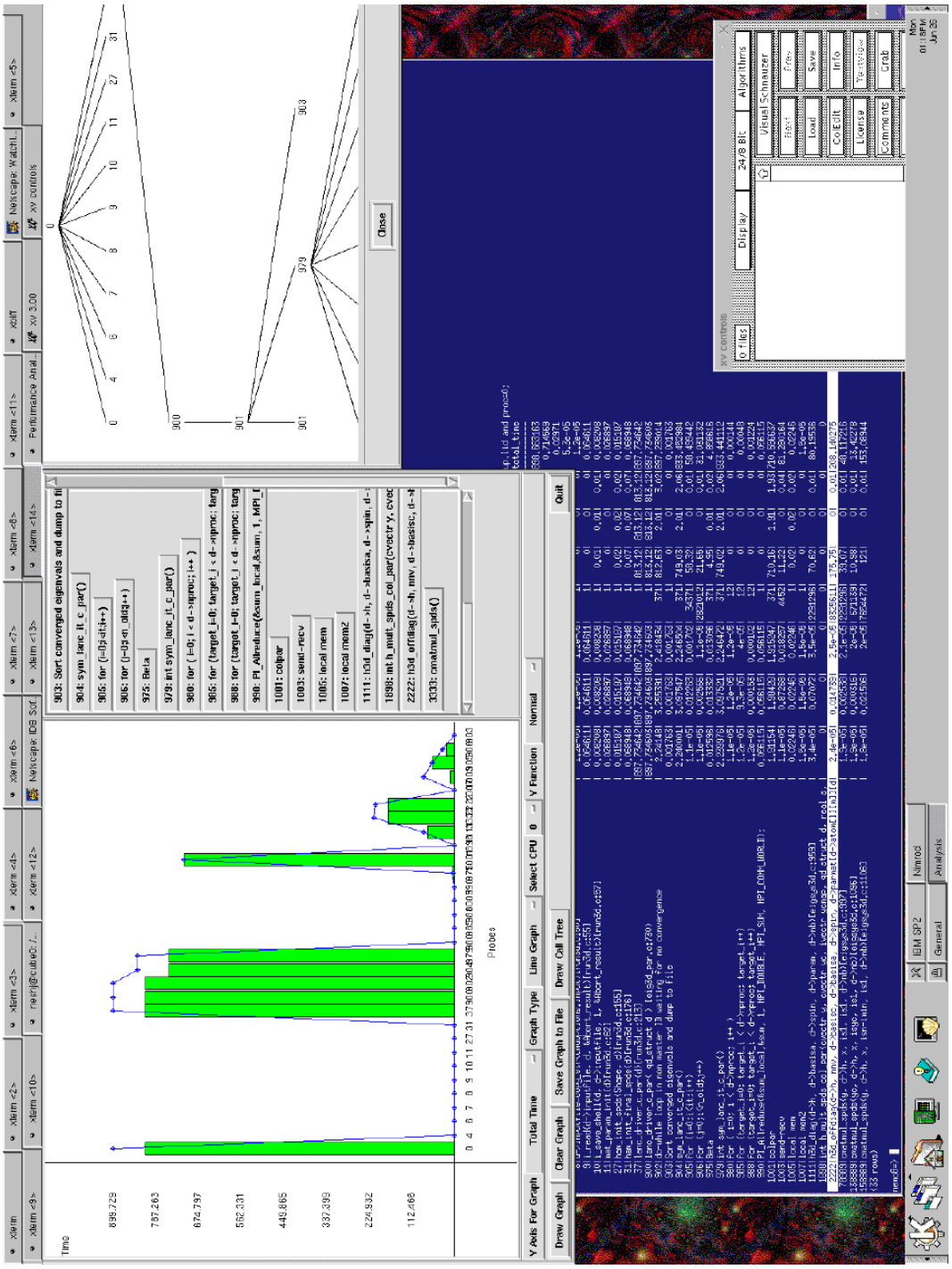


Figure 6.5: Typical user session. Vistool displays performance data for a single NEMO run on 12 processors. Interactive query results are displayed in background window. The highlighted row in the relation indicates the probe where optimzation efforts are focused.

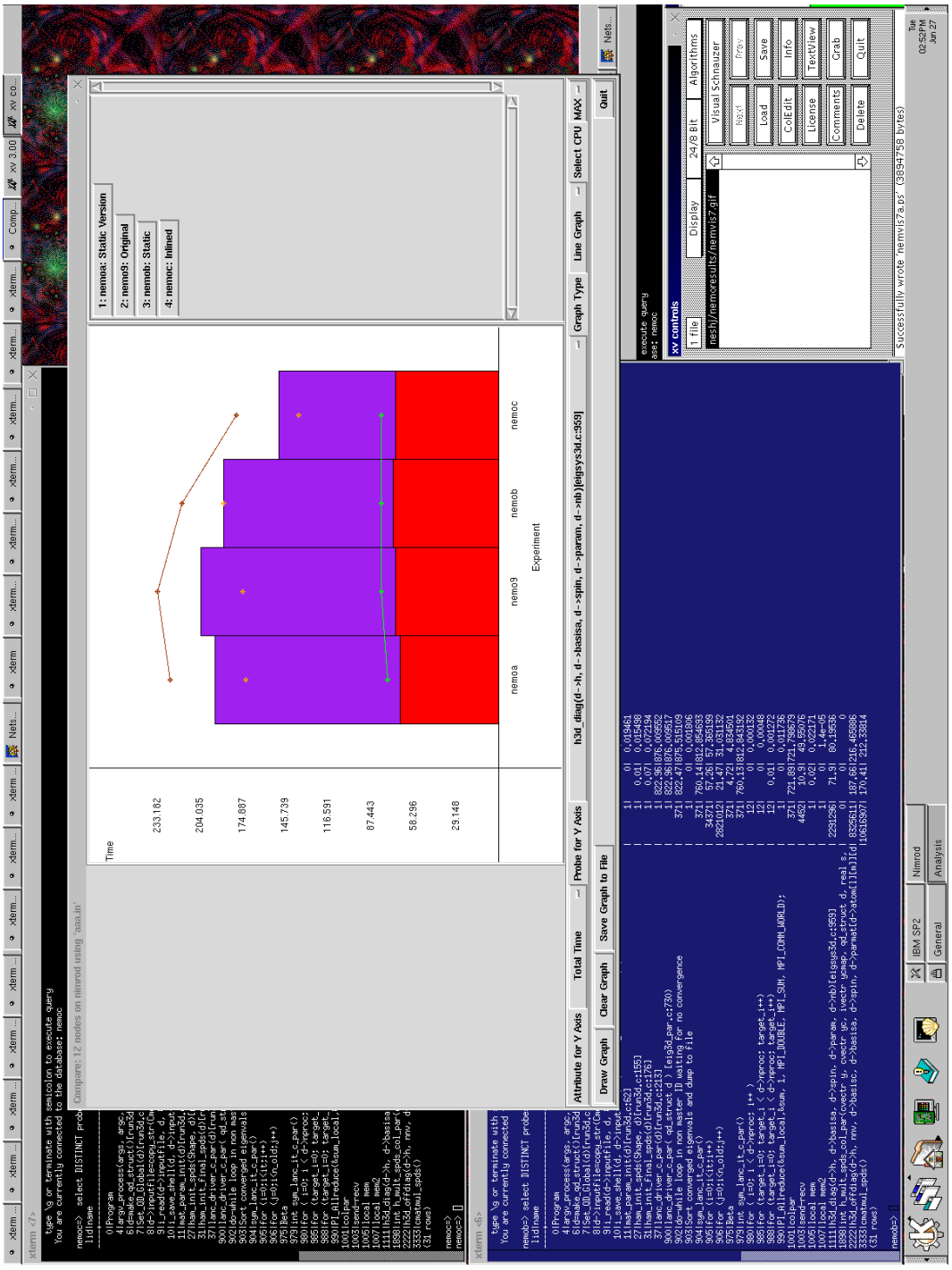


Figure 6.6: Typical user session. EDFtool displays performance data for 4 experiments involving NEMO on 12 processors. Interactive query results are displayed in background windows.

CHAPTER 7
Parallel Adaptive Finite Element Analysis Codes

In this example we further demonstrate the evolving methodology associated with use of the instrumentation database. The codes we consider in this chapter are the largest and most complex applications we have analyzed. The instrumentation introduced and visualizations generated are not used for optimization, instead we examine facets of comparative analysis by permuting run-time and compile-time parameters and observing their effect on performance. We show how IDB is used to identify unexpected performance characteristics.

7.1 Problem Description

The finite element method (FEM) [37] is used for solving partial differential equations (PDEs). For numerical reasons, adaptivity is introduced which focuses computation to "interesting" regions of the problem domain. This results in improved time and spatial efficiency; it entails refining portions of the discretized domain, or mesh, during computation [10]. The types of refinement include space-time (h-refinement), method order (p-refinement), and mesh movement to follow evolving phenomena (r-refinement) for time dependent, or transient problems. All of these serve to concentrate or dilute computation within the discretized domain of transient and steady-state solutions.

Parallelism must be introduced to handle computational and memory requirements of large three-dimensional problems. This adds significant complexity in that the user must maintain load balance between processors, handle interprocessor communications, and maintain large data structures across multiple nodes. These issues are exacerbated because distribution, or partitioning, of the discretized domain must be updated as adaptive refinements are applied.

7.1.1 SCOREC Tools

The Scientific Computation Research Center (SCOREC) at Rensselaer Polytechnic Institute has developed tools in support of building parallel adaptive finite element codes to solve elliptic, parabolic, and hyperbolic PDE problems [27]:

- **Mesh Database**

  The SCOREC Mesh Database (MDB) [57] [22] stores and manipulates object-oriented and hierarchical finite element mesh entity data. A mesh entity consists of three-dimensional regions, faces, edges, and vertices. The MDB provides support for creating, modifying, removing, and querying these mesh entities. The Parallel Mesh Database (PMDB) is built on top of the MDB and provides support for distributed mesh data.

- **Mesh Migration**

  Minimizing interprocessor communication and distributing work equally requires a mechanism for migrating mesh elements to other processors. Mesh regions are assigned to a unique processor while interprocessor boundary elements are duplicated on nodes that contain shared regions. Figure-7.1 shows an example of two-dimensional mesh element migration.

- **Mesh Enrichment**

  The mesh enrichment library [62] performs refinement or coarsening operations based on error indicator information and set threshold values. If error falls below a minimum threshold, coarsening (which involves coalescing neighboring mesh entities to form a larger entity) is applied. If error estimations are too high, refinement (which involves subdividing an entity to create multiple new entities) is applied. This enrichment process ensures a consistent level of solution quality throughout the discretized domain; it occurs at the cost of introducing load imbalance among processors.

- **Load Balance**

  The purpose of load balancing, or partitioning, is to distribute work equally among processors. While it ensures that no processors are idle, it does not

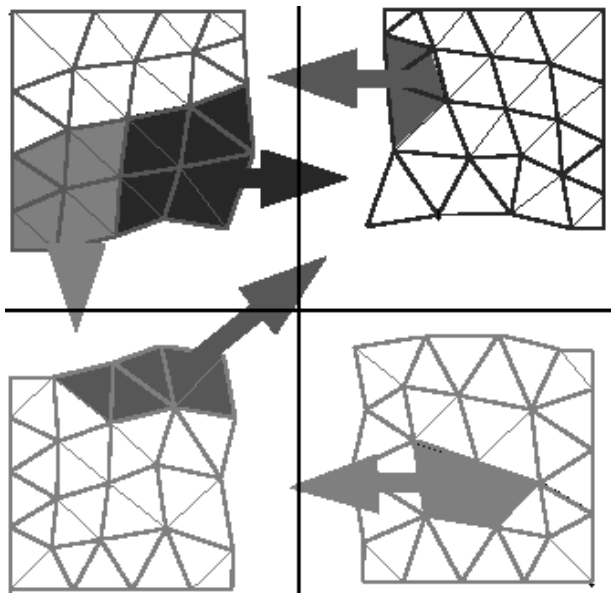


Figure 7.1: Illustration of migration process. This example shows two-dimensional mesh elements being migrated between four processors (adapted from [68] and [22]).

ensure that the overall computation is done efficiently. Partition quality is also a function of the number of element faces on interprocessor boundaries; which influences interprocess communication. Two metrics measure partition quality:

- **Maximum Local Surface Index (MLSI)**. Measure of the maximum (among processors) percentage of element faces on the boundary of a processor.
- **Global Surface Index (GSI)**. Measure of the percentage of all faces on processor boundaries.

In addition to communication load, partition quality is measured in terms of compactness which significantly effects efficiency of the solver. We consider three dynamic load balancing algorithms: *Iterative Tree Balancing* (ITB), *Parallel Sort Inertial Recursive Bisection* (PSIRB), and *Octree Partitioning* (OCTPART) [22], [21].

Iterative Tree Balancing relies on lightly loaded processors requesting load from their most heavily loaded neighbors. These requests result in a forest of trees, as Figure-7.2 shows. Load is balanced by iteratively migrating layers of boundary elements. Since ITB is "diffusive", it works well for small imbalances but suffers when mesh elements are poorly distributed [57], [63]. ITB can run for multiple iterations to achieve global load distribution within a specified tolerance or fixed number of iterations. Since ITB migrates boundary elements, it does not significantly affect partition compactness.

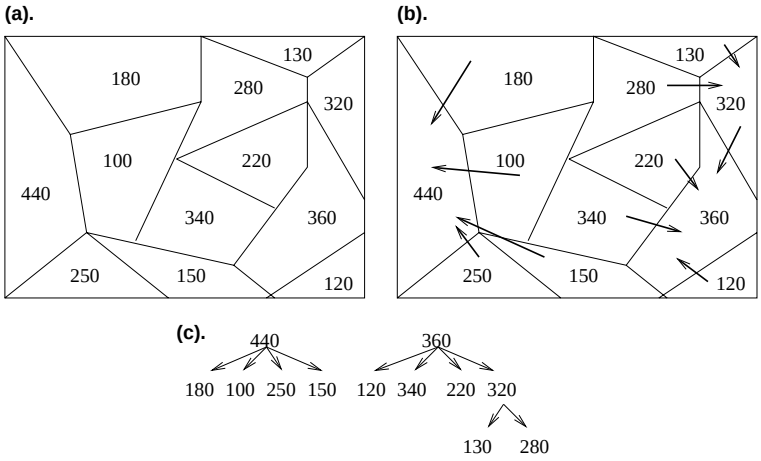


Figure 7.2: (a). Original unbalanced load. (b). Load requests. (c). Forest of trees generated by load requests.

Parallel Sort Inertial Recursive Bisection is a parallel implementation of Inertial Recursive Bisection (IRB) [45], a static partitioning procedure used for initial distribution of the mesh. Elements are sorted based on their location within the domain, which is bisected in a direction orthogonal to the principal axis of inertia. Parallel sorting of the coordinates within the inertial frame enables IRB to be parallelized for dynamic repartitioning [21]. Figure-7.3 shows an example of PSIRB partitioning. This algorithm quickly produces a strict and compact balance of elements but does not do as well as other schemes with respect to the MLSI and GSI metrics.

Octree Partitioning. OCTPART uses the traversal of an Octree structure underlying the mesh to achieve a partitioning. Traversals vary across different

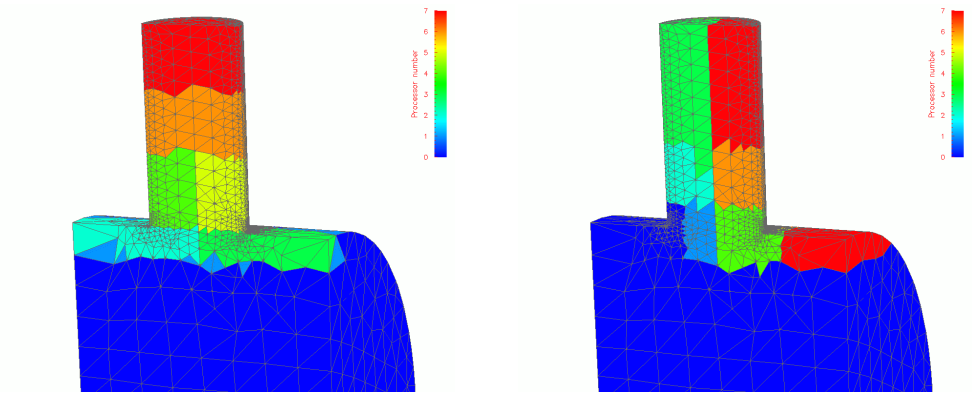


Figure 7.3: Initial mesh distributed to 8 processors using ITB (left) and OCTPART (right) for perforated muzzle brake solution.

OCTPART implementations. In the implementation we consider, a depth first traversal of underlying octrees is used. After Octree generation, cost metrics are computed for all subtrees. Cost metrics are often the number of elements in each octant, but other weighting schemes based on the polynomial degree of elements can be used. The optimal load for each processor is computed based on total number of processors and the total cost of all subtrees. Subtrees are placed on a partition if the cost of that subtree plus the load on the current partition is less than the optimal load. If it is not, one of two things occurs: [21]

1. Depth first traversal decomposes the subtree further until a smaller subtree is added that fits in the current partition.
2. The current partition is closed and the subtree is inserted into the next partition.

The generated tree structure is stored on multiple processors. In the implementation we considered for our analysis, these subtrees are updated between iterations. Figure-7.3 shows an example of Octree partitioning.

- **Predictive Load Balancing**

  Error indicators and refinement thresholds can be used to assign weights to specific mesh elements that are likely to be migrated after the enrichment

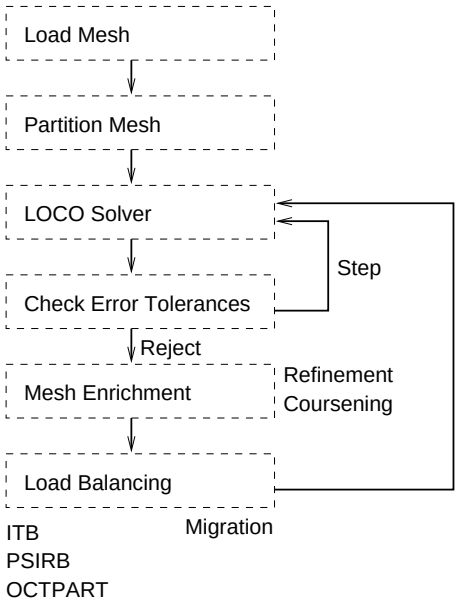


Figure 7.4: Schematic of Loco solver and SCOREC tools.

stage. Instead of introducing imbalance during enrichment, a weighted load balancing phase is introduced prior to mesh refinement. This can significantly improve overall performance [22] [21]. IDB analysis of predictive load balancing is beyond the scope of what we consider in this chapter.

Most of these components are used together with *Loco* [23], a parallel adaptive solver which implements a discontinuous Galerkin solution method [5] [12] [13] of the compressible Euler equations.

Figure-7.4 shows the structure of the solver and SCOREC tools that we used. Initially, the mesh is loaded into memory and partitioned across all processors. An iteration begins with the solver acting upon the mesh. Once a solution for the current time step is computed, error estimations are evaluated; if they fall within a specified tolerance, the solver continues on to the next step, else the mesh is refined or coarsened and load balancing migrates elements before the solver resumes.

7.2 Instrumentation

We consider two hyperbolic transient problems which can be solved with *Loco*: (i) a Rayleigh-Taylor flow instability, and (ii) the flow in a perforated shock tube.

The same solver, mesh enrichment, migration, and load balancing libraries are used. Instrumentation is automatically introduced at specific points in the respective control flow graphs that are common to both problems. In this way, the code is instrumented only once.

Description	ID
while (meshnum<MESHITER && (STEPTYPE==0 || time<=TSTOP))	1300
MD_init(), load_model(), pmdb_ge_tabl_new(), MM_new()	100
Program	0
balance_rebal()	1700
balance_rebal()	2000
balance_setup_init_params()	200
file_load_initial()	300
if (!solver_step_or_reject())	1350
solver_err()	1100
solver_init_boundary()	1800
solver_refine()	1500

Table 7.1: Probe Table. This table is the result of executing the following query "SELECT DISTINCT * FROM lookup;

Probes were introduced incrementally. Figure-7.5 shows how the CFH grows as instrumentation is added. Probes not on the critical path, and those associated with file check-pointing, were removed. Table-7.1 shows probes that make up the reference instrumentation of the *Loco* solver. Probes 1700 and 2000 instrument the same function: 1700 instruments the call to balance_rebal() while 2000 instruments the function internally; this routine initiates one of the three load balancing schemes we considered. Probe 1350 instruments the call to the solver and 1500 instruments mesh refinement.

7.3 Rayleigh-Taylor Flow

Simulation of astrophysical phenomena, specifically nuclear flashes in bodies such as neutron stars and white dwarfs, is a computationally and mathematically complex problem. A crucial element to these simulations is correctly modeling the flame front as it travels through the body of the star. As the front travels from less dense to more dense regions, it is subject to Rayleigh-Taylor instabilities, which

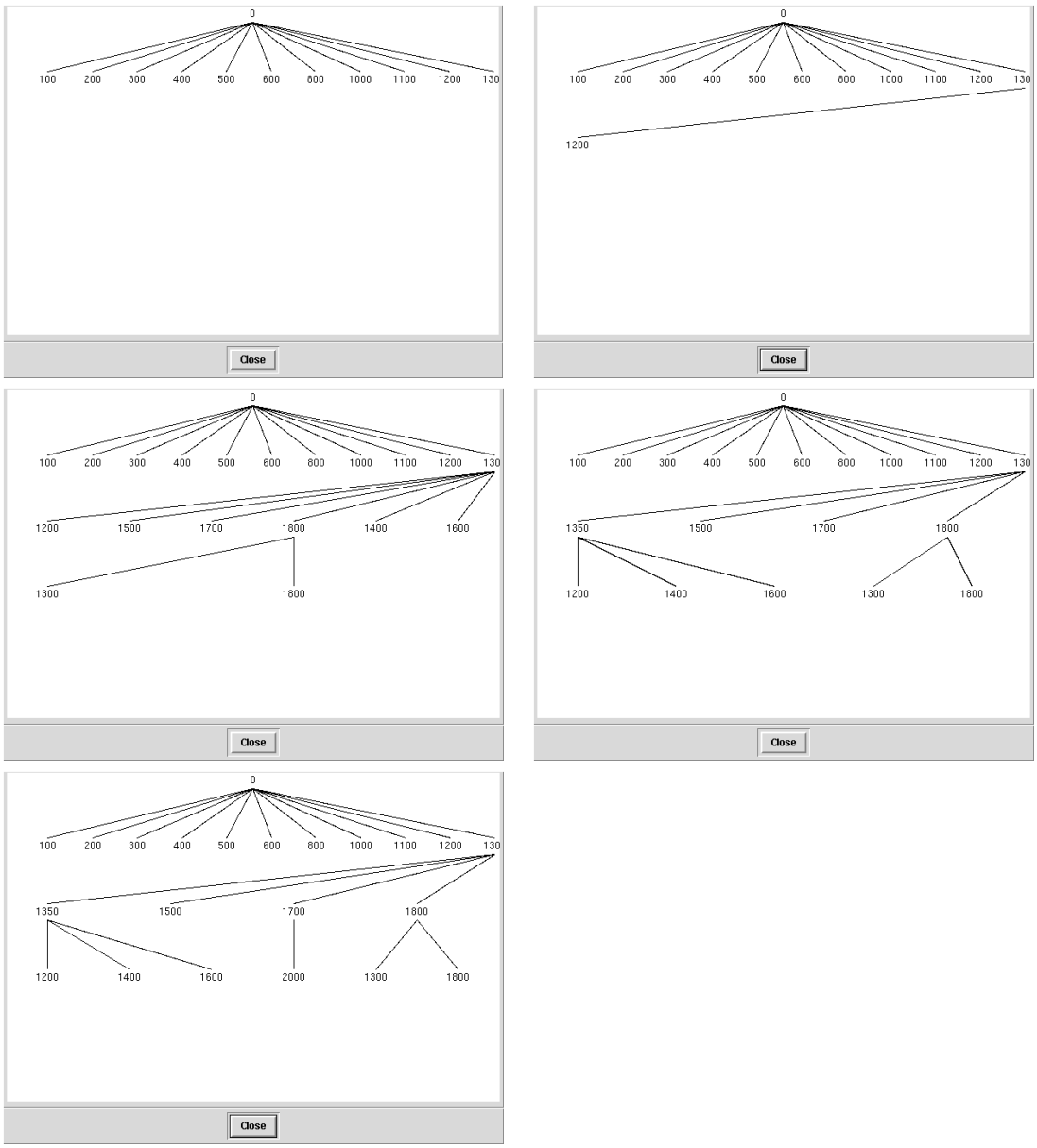


Figure 7.5: Instrumentation progression. These control flow hierarchies indicate the progression in which probes were introduced to the solver. Probes were initially introduced to all events in the `main()` program. In subsequent runs, probes not on the critical path were removed and new probes were added.

drastically effect the size and duration of the resulting nuclear flash. The quality of such simulations depends on accurate modeling of these instabilities [68].

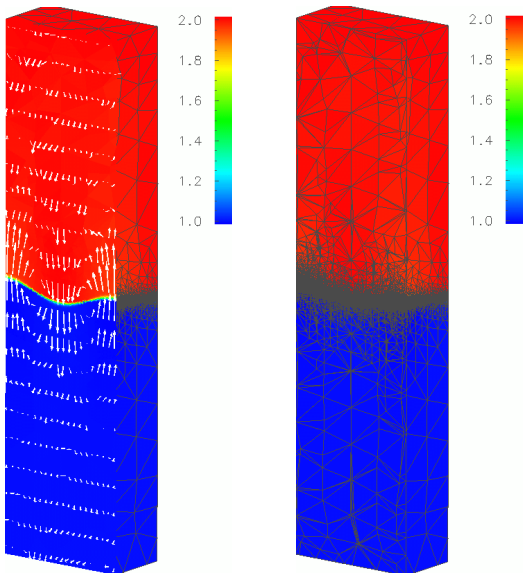


Figure 7.6: Rayleigh- Taylor interactions and mesh visualizations.

The Rayleigh-Taylor flow we considered is bound by a box (RTBOX), with a height 8 times larger than the length and width. (see Figure-7.6). The simulation models interactions of a more dense fluid on top of a less dense fluid. IDB is used to measure how the different load balancing procedures affect solver, enrichment, and overall performance. We also use IDB to show that the the physics of the underlying system being modeled is significant. A mathematical description of the Rayleigh-Taylor flow problem can be found in [68].

7.3.1 Performance Experiments

There are scores of run-time parameters that affect RTBOX performance. These include load balancer selection, simulation time, number of processors, optimization, etc. Table-7.2 lists the performance experiments that were run.

The RTBOX was run on 8 processors using ITB, PSIRB, and OCTPART. Each run partitions the mesh and simulates 0.0001 time units of the solution. The same experiments were repeated on 16 processors which simulate 0.200 simulation time units. We also demonstrate that IDB can be used to measure the significance

Experiment Name	Nodes	Load Balancer	Simulation Time (TSTOP)	Optimized YES
rt_itb_o_8	8	ITB	0.0001	YES
rt_psirb_o_8	8	PSIRB	0.0001	YES
rt_oct_o_8	8	OCTPART	0.0001	YES
rt_itb_o_16	16	ITB	0.2000	YES
rt_psirb_o_16	16	PSIRB	0.2000	YES
rt_oct_o_16	16	OCTPART	0.2000	YES
rtbox2 (no opt)	8	OCTPART	0.0001	NO
rtbox1 (opt)	8	OCTPART	0.0001	YES
rt_56-1	56	OCTPART	0.1000	YES
rt_56-2	56	OCTPART	0.1000	YES
rt_56-3	56	OCTPART	0.1000	YES
rt_56-4	56	OCTPART	0.1000	YES
rt_56-5	56	OCTPART	0.1000	YES

Table 7.2: Rayleigh-Taylor performance experiments.

of permuting compiler parameters such as optimization.

Figure-7.7 shows that on 8 processors the PSIRB version outperforms both the ITB and OCTPART versions overall. Although less time is spent in the solver using OCTPART than PSIRB or ITB (as Figure-7.8a shows), Octree partitioning is more costly than the others (see Figure-7.8b). The time lost by OCTPART in load balancing would be recovered in the solver for larger values of TSTOP. When the same experiments are conducted using 16 processors for larger simulation time, we observe that the total execution time of the OCTPART version is surpassed by PSIRB, as Figure-7.9 illustrates. This result appears to be counter-intuitive based on previous analysis of Octree partitioning versus other load balancing schemes [45], [21].

This behavior is partly the result of OCTPART's spatial decomposition of the problem domain; it defines the *Octree Universe*, a cube that is inscribed by the problem domain. In this example, the problem domain occupies a very small part of this *Universe*, as Figure-7.10 illustrates. Octree partitioning typically outperforms other balancing schemes when the problem domain occupies a large area of the total Octree Universe. The IDB approach was useful for collecting and presenting performance data used to identify this unexpected performance result.

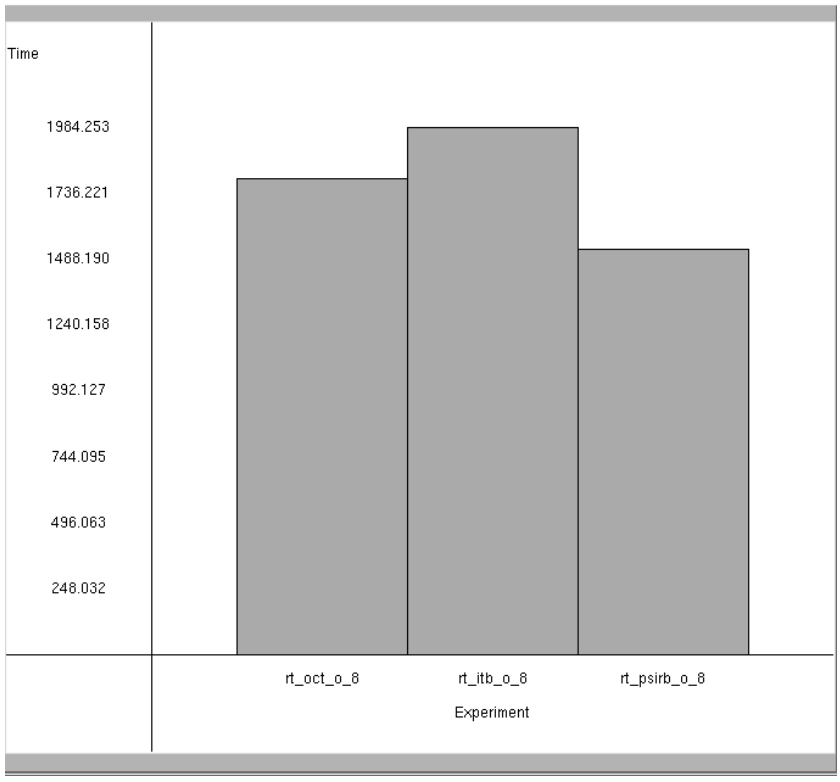
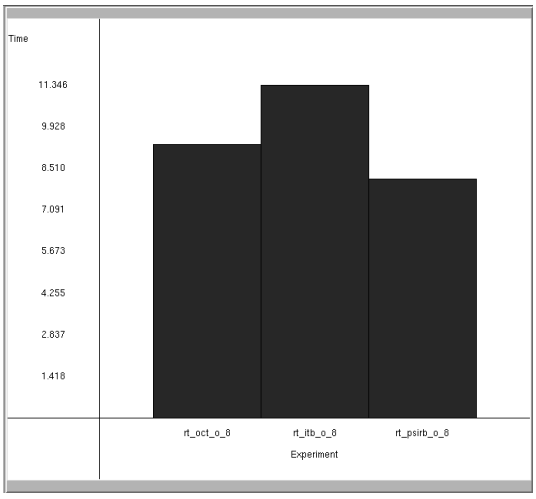
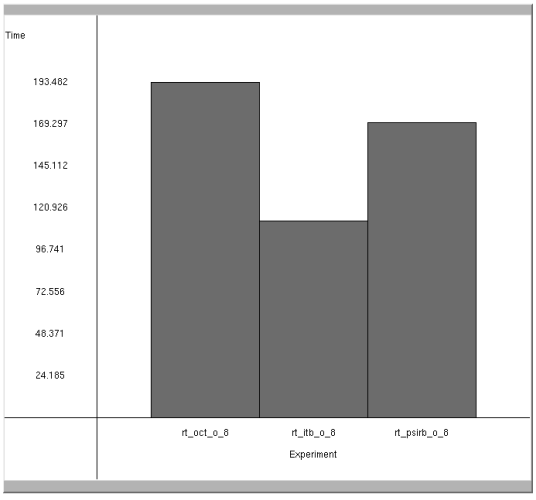


Figure 7.7: Run time of RTBOX with ITB, PSIRB and OCTPART on 8 nodes.



(a). Solver time on 8 nodes



(b). Load balance time on 8 nodes

Figure 7.8: Rayleigh- Taylor interactions and mesh visualizations.

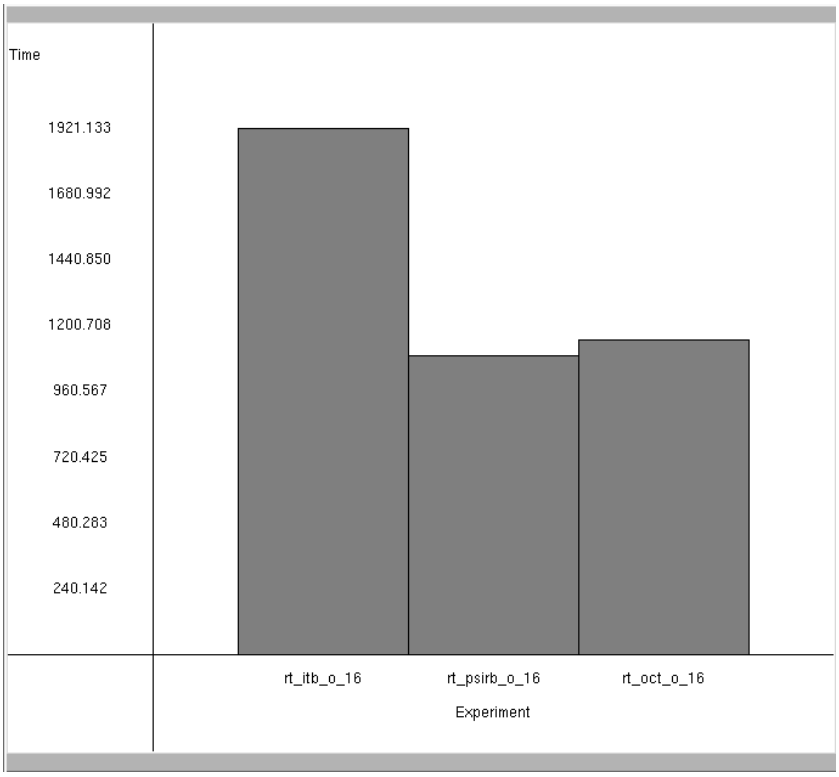


Figure 7.9: Run time of RTBOX with ITB, PSIRB, and OCTPART on 16 nodes.

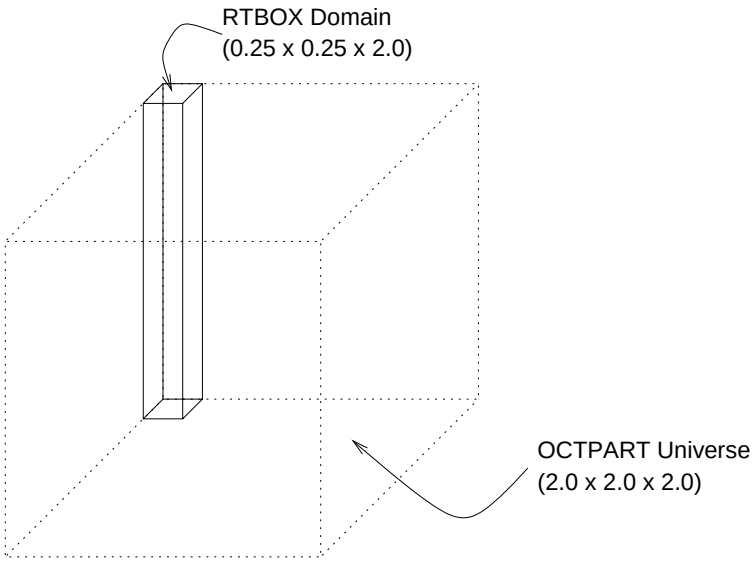


Figure 7.10: RTBOX problem domain superimposed with OCTPART universe.

The 8 processor RTBOX runs used a very short simulation time (where TSTOP = 0.0001) in the interest of keeping execution time low, approximately one hour. The 16 processor examples were run with a significantly longer solution time (where TSTOP = 0.200), roughly 30 hours of execution. The resulting instrumentation data showed that as simulation time passes, more time is spent refining and coarsening the mesh as opposed to executing solver computations as shown in Table-7.3. Again, this result appears to be contrary to what is expected. However, closer inspection of *Loco*'s log files show that mesh refinement becomes more frequent as simulation time progresses; the mesh is refined as the front propagation through the domain is modeled. This shows that the physics of the Rayleigh-Taylor problem does impact performance.

Description	Count	CPU Time	Total Time
Main Loop	39	14229.64	15680.109432
Solver	39	2386.83	2386.808268
Refinement	39	9638.49	10617.09714
Load Balancing	39	2548.73	2549.316627
Initialization	1	0.05	2.944329

Table 7.3: Probe Data for 16 node RTBOX run with TSTOP = 0.200

IDB was also used to compare optimized and unoptimized versions of RTBOX using OCTPART for TSTOP = 0.0001. Figure-7.11 shows that the optimized code is significantly faster. Closer inspection of the code shows that the optimized version is not laden with as much error checking code as the unoptimized version.

Lastly, we ran RTBOX using Octree partitioning with TSTOP = 0.100 on 56 processors. This performance experiment was repeated five times for the purpose of demonstrating the scalability of IDB and gauging the regularity of RTBOX. Figure-7.12 shows the quartile graph of the coalesced data. We can conclude that performance is consistent across multiple executions.

7.4 Perforated Muzzle Brake

The next problem we consider is a three-dimensional unsteady compressible flow in a cylinder with a cylindrical vent [45], [68], [21]. The motivation for this

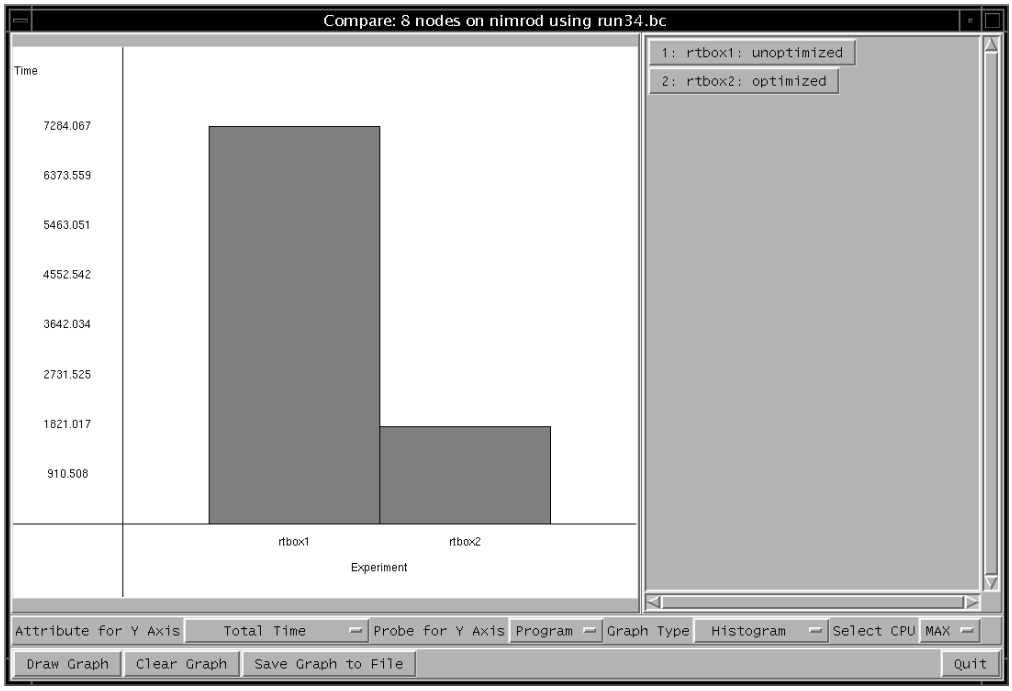


Figure 7.11: Total execution time of optimized and unoptimized OCT-PART version of RTBOX.

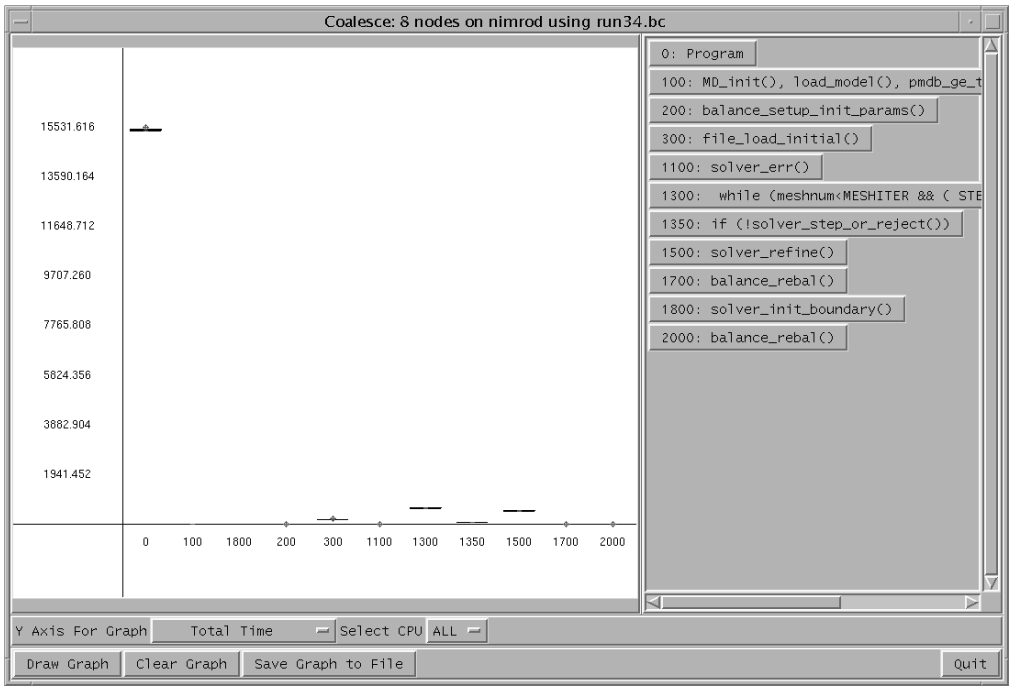


Figure 7.12: Visualization of COALESCE mode experiment for 56 node run of RTBOX using Octree partitioning for TSTOP = 0.100.

problem is derived from study of perforated muzzle brakes (PMB) for large calibre guns. The domain can be cut in half through the center of the vent by symmetry. The simulation starts with a Mach 1.23 flow through the cylinder, and begins as if a diaphragm between the two cylinders was removed. Figure-7.3 shows the problem domain [22], [20].

7.4.1 Performance Experiments

We demonstrate how the same *Loco* reference instrumentation can be used to measure performance of PMB with respect to ITB, PSIRB, and OCTPART on 8 and 16 processors. To this end, we conducted the performance experiments listed in Table-7.4.

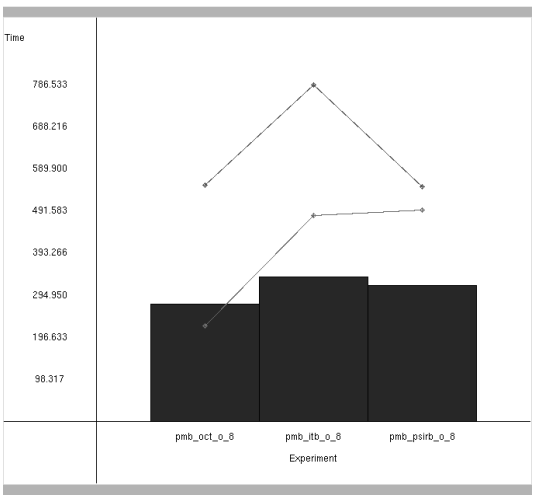
Experiment Name	Nodes	Load Balancer	Simulation Time (TSTOP)
pmb_itb_o_8	8	ITB	0.0100
pmb_psirb_o_8	8	PSIRB	0.0100
pmb_oct_o_8	8	OCTPART	0.0100
pmb_itb_o_16	16	ITB	0.0100
pmb_psirb_o_16	16	PSIRB	0.0100
pmb_oct_o_16	16	OCTPART	0.0100

Table 7.4: Perforated Muzzle Brake performance experiments.

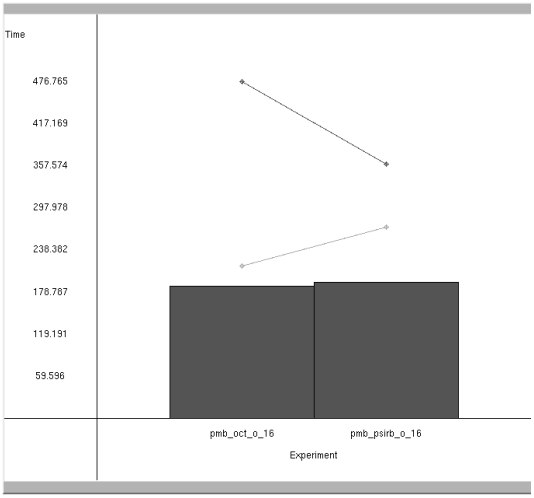
Initial performance experiments involved using a relatively small mesh on 8 processors. Three runs using ITB, PSIRB, and OCTPART are shown in Figure-7.13a.

On 8 processors, the OCTPART version had the best overall run-time. Total execution time of the three versions were ITB = 16866.57s, PSIRB = 15083.84s, and OCTPART = 13529.72s. The same problem on 16 processors shows Octree partitioning was surpassed by PSIRB and ITB (see Figure-7.14b). When we look closely at instrumentation data collected from the OCTPART, PSIRB, and ITB runs (shown in Figure-7.14), solver and load balance are faster with PSIRB. Refinement, however, is significantly faster using Octree partitioning. The two factors contributing to this are:

- A pre-refined mesh was initially loaded.

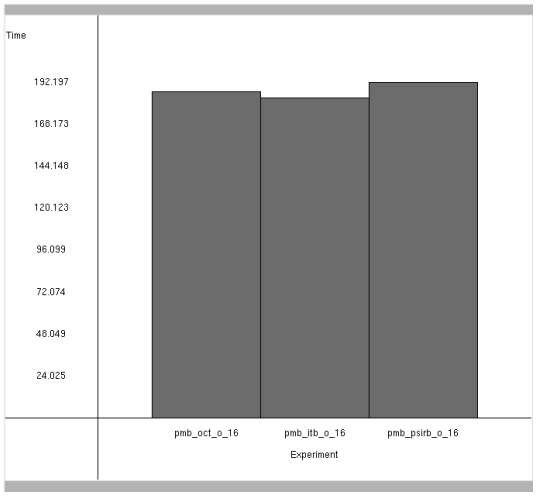


(a). PMB on 8 nodes using ITB, PSIRB, and OCTPART

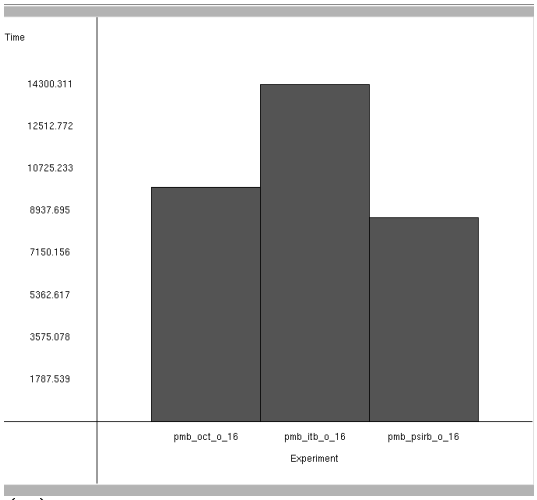


(b). PMB on 16 nodes using PSIRB and OCTPART

Figure 7.13: PMB run on 8 and 16 nodes using ITB, PSIRB, and OCTPART. The bars show solver execution time. The bottom line shows load balancing time, and the top line shows time spent performing mesh enrichment operations.



(a). Solver time on 16 nodes



(b). Total time on 16 nodes

Figure 7.14: PMB run on 16 nodes using ITB, PSIRB, and OCTPART.

- Mesh size is small relative to the number of processors.

The diffusive nature of ITB works well for meshes that are mostly balanced. Also, the compact balancing that PSIRB can quickly provide works well for the relatively small number of mesh elements in this example.

7.5 Discussion

Future applications of IDB include comparing the OCTPART implementation that we considered in these examples to the implementation used in the Zoltan [15] [16] library, which provides a common interface for selection of load balancing algorithms, that is being developed at Sandia National Labs. The Zoltan implementation of Octree partitioning differs in that subtrees are regenerated, as opposed to updated, between iterations. Also, Zoltan supports different subtree traversals using gray code or Hilbert orderings [27] in addition to the depth-first traversal used by the OCTPART implementation that we considered. Updating the subtrees, instead of regenerating them, introduces small errors that cause the MLSI and GSI to slowly degrade; thus affecting overall performance, as shown by IDB.

These examples show that the IDB approach to experiment management is powerful enough to permute a wide range of run-time and compile-time parameters such as number of processors, input size, simulation time, as well as whole program modules, specifically load balancing algorithms, problem types, and implementations. In this chapter, we considered a subset of the different problem types that exist. IDB can be used to fully explore this large problem space.

Load Balance	x	Predictive Balancing	x	Equation Type
ITB		ON		Elliptic
PSIRB		OFF		Parabolic
OCTPART				Hyperbolic

The SCOREC tools include support for different partitioning schemes and implementations, predictive load balancing, and problem types. IDB provides the systematic ability to explore how these attributes affect one another.

CHAPTER 8
Discussion and Conclusions

8.1 General

We have shown that the instrumentation database approach provides a useful means of collecting, archiving, and querying performance data collected from parallel and object oriented scientific applications. Moreover, we provide a formal framework for experiment management through the use of relational database and experiment definition files. This framework provides the basic tools with which it is possible to develop more sophisticated instrumentation, visualization, and analysis tools in a scalable way.

8.2 Emergent Methodologies

Performance tuning and optimization is an iterative process. In the past, the structure of these iterations was loosely defined by the user. The introduction of the database and tools for comparative analysis provide the user with the means to approach optimization in a more systematic manner. Commonalities exist in how IDB was used to instrument and analyze the various scientific codes we presented. As familiarity with the IDB approach increases, these emergent methodologies will hopefully continue to evolve and define a new framework for performance analysis. These include:

- **Reference Instrumentation.** Introduction of probes at the first level of the control flow hierarchy (CFH). The resulting performance database is queried to view which branch or branches are on the critical path, or consume the most time.

- **Growing the tree.** Introduction of probes on subsequent levels of the CFH. This process is repeated until the performance events that comprise the critical path are instrumented. Cues that a probe can be further optimized include:

  - Instrumented loops or functions that iterate or are invoked many times.
  - Probes showing a significant difference between CPU and total time. This may indicate synchronization or blocking delays.
  - Probes that include large segments of code, namely long functions or loop bodies. In many cases, such segments usually contain a performance critical event that may reside on the critical path.

- **Pruning the tree.** Removal of probes that are not on the critical path. The resulting control flow hierarchy defines the *reference instrumentation version.* This is a minimally instrumented version of the program with probes along the critical path of the CFH.

- **Experimentation.** Optimization of instrumented code or permutation of run-time and compile-time parameters.

Analysis methodologies will continue to evolve as the IDB framework and user base grows. This approach resulted in an identification tool designed to aid the user in localizing bottlenecks or probing the effects of modifications to the code or its environment. IDB collects data and presents it in an intelligent manner. It is still incumbent on the user to derive meaningful conclusions and optimization strategies.

8.3 Research Contributions

This Instrumentation Database approach addresses several issues in the area of performance analysis; among these are scalability, cost of invocation, ease of use, accuracy, object-oriented support, multi-language compatibility, platform independent comparative analysis, and experiment management. IDB uniquely addresses these issues by leveraging database technology in conjunction with a novel approach to instrumentation.

Scalable instrumentation. We demonstrated that a minimal set of performance events (PROC, CALL, LOOP, and COMM) coupled with the control flow graph and aggregate statistics (MIN, MAX, AVG, COUNT, and SDEV) are sufficient for deriving performance attributes. The amount of instrumentation data

collected is bound by the path, or paths, traversed through the control flow graph. Database size is not affected by run time, input size, or number of nodes.

Multi-language and object-oriented support. We developed a POSIX compliant instrumentation API that ensures availability across multiple platforms. *++-izing* enables C codes to use the C++ API. Similarly, compiler and linker support enable FORTRAN 90 codes to call IDB probes. The Java Native Interface (JNI) provides a mechanism for future instrumentation of Java applications. Also, extensions to the control flow hierarchy involving multiple instances of a CFH within a node enable object-oriented codes to collect performance data within each object contributing to the critical path.

Mapping program structure onto relation database schema. Archival of statistical probe data, control flow hierarchy connectivity, and static data into a traditional relational database guarantees a standard interface to performance data. Data collected across multiple versions of the same program, across multiple architectures, compiler options, or run-time options can be compared or archived for future analysis. Moreover, the instrumentation database enables analysis on architectures or systems other than those where performance data was collected. Standard Query Language (SQL) provides a powerful interface where data-mining and querying can be cast against multiple program executions. This interface, along with language specific extensions for embedded SQL, provides the foundation for developing sophisticated visualization and graphical query tools.

Automated instrumentation. To ensure a low cost of invocation, the process of introducing probes at the source code level is automated. Target applications are lexically analyzed and instrumentation is added based on information provided by the analyst in a graphical format. The design of this tool was guided exclusively by user feedback.

Visualization front-end. Embedded SQL and graphical toolkits simplify development of sophisticated visualization tools. For example, IDB provides *Vistool* for presenting data collected within a single execution graphically. Similarly, *EDFtool* presents data in a graphical means; however, data spans multiple executions of the same or modified versions of an application. Queries are constructed

graphically through a user interface and the results returned from the database are presented graphically or numerically in tabular form.

Experimental view of performance analysis. IDB is unique in that it forces the analyst to adhere to basic experimental techniques for optimizing and analyzing code. Program executions are defined as performance experiments. *Experiment Definition Files* (EDFs) created during instrumentation are referenced during visualization. All aspects of the IDB framework (instrumentation tool, database, and visualization front-end) are integrated around these definition files. Experiment modes include comparison across multiple versions or coalescing data collected from multiple runs of the same code. Coalescing is useful for *dusty deck* [34] and gauging regularity; that is to say that it is useful on multi-user systems where performance can be affected by external factors or by asynchronous events during execution that can alter performance across multiple runs.

The IDB approach represents a significant change in performance analysis of large parallel and object-oriented systems. This approach is readily extensible to codes outside of the scientific computing domain.

8.4 Future Work

IDB will be used to analyze additional applications to further show that it is a viable and powerful approach to analysis of large codes. The scope of IDB can be extended to provide formal support for large single processor and distributed codes. The approach is applicable to mainstream applications outside of the numerically intensive scientific computing domain. Part of making IDB available to this user base involves enhancing existing tools and providing mechanisms for dealing with single processor and distributed codes.

8.4.1 PAPI Integration

IDB probes are currently implemented around the `MPI_Wtime()` and `clock()` calls. The disadvantage of this is that it limits IDB to parallel codes based on the Message Passing Interface (MPI). Furthermore, probes are uncoupled from the underlying hardware to the extent that there is no easy way to monitor performance

critical issues such as cache utilization, floating point operations, etc. We plan to integrate the IDB approach with the Performance Application Programmer Interface (PAPI) currently being developed by Dr. Jack Dangara at the University of Tennessee at Knoxville. PAPI provides a platform independent interface to vital hardware performance statistics through a hardware specific *substrate*, or interface.

This integration will involve modification of the current probe implementation; the database schema will also be extended to capture additional aggregate information, such as cache utilization or sustained Mflops. The probes will provide more robust information about how the underlying architecture is being utilized while preserving platform independence.

8.4.2 Experiment Automation

The IDB approach provides the user with all the tools required to instrument and analyze performance for the purpose of localizing bottlenecks in an application. We will explore ways to use these tools and develop new ones such that bottleneck localization can be fully automated. The process of growing the control flow hierarchy and pruning excess probes can be automated.

An *instrumentor* will parse the code in the same way the automated instrumentation tool does, and then it will introduce probes automatically. The system will then execute the application and populate the instrumentation database as per invocation syntax provided by the user. The system will be able to query the database for probes that consume some percentage of the critical path. The system can do this iteratively, adding new probes and removing old ones. Key issues include defining a probe's contribution to the critical path, automatically executing and populating the database, and embedding knowledge of the tolerances to set to stop the process. The result will be a reference version of the code containing the minimal number of probes along the critical path.

LITERATURE CITED

[1] Cray user guide. Technical Report JPL 400-508, Jet Propulsion Laboratory, May 1998.

[2] Postgresql administrator's guide, programmer's guide, developer's guide, user's guide, 1998.

[3] Mark W. Beall and Mark S. Shephard. A general topology-based mesh data structure. *Int. J. Numer. Meth. Engng.*, 40(9):1573–1596, 1997.

[4] C. K. Birdsall and A. B. Langdon. *Plasma Physics via Computer Simulation. The Adam Hilger Series on Plasma Physics.* Adam Hilger, New York, 1991.

[5] Rupak Biswas, Karen D. Devine, and Joseph E. Flahe rty. Parallel, adaptive finite element methods for conser vation laws. *ANM*, 14:255–283, 1994.

[6] Daniel K. Blanks, Gerhard Klimeck, Roger Lake, R. Chris Bowen, Manhua Leng, Chenjing Fernando, William R. Frensley, and Dejan Jovanovic. Nemo quantum device simulator. In *Government Microcircuit Applications Conference Digest of Papers (GOMAC)*, page 218, March 1998.

[7] Pradip Bose and Thomas M. Conte. Performance analysis and its impact on design. *Computer*, 31(5):41–49, May 1998.

[8] Jon Chen. Automated instrumentation independant study project, January 2000.

[9] Doreen Y. Cheng. A survey of parallel programming languages and tools. Technical Report RND-93-005, NAS Systems Division, NASA Ames Research Center, March 1993.

[10] K. Clark, J. E. Flaherty, and M. S. Shephard. *Appl. Numer. Math., special ed. on Adaptive Methods for Partial Differential Equations*, 14, 1994.

[11] Mark J. Clement and Michael J. Quinn. Analytical performance prediction on multicomputers. In *Proceedings Supercomputing '93*, pages 886–894, Los Alamitos, CA, USA, 1993. IEEE Comput. Soc. Press.

[12] B. Cockburn, S.-Y. Lin, and C.-W. Shu. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws III: One-Dimensional systems. *JCP*, 84:90–113, 1989.

[13] B. Cockburn and C.-W. Shu. TVB Runge-Kutta local projection discontinuous Galerkin finite element method for conservation laws II: General framework. *Math. Comp.*, 52:411–435, 1989.

[14] Deitel and Deitel. *How to Program Java, Second Edition.* Prentice Hall, 1998.

[15] Karen D. Devine, Bruce A. Hendrickson, Eric Boman, Ma tthew St. John, and Courtenay Vaughan. *Zoltan: A Dynamic Load Balancing Library for Parallel Applicat ions; Developer's Guide.* Sandia National Laboratories, Albuquerque, NM, 1999. Tech. Report SAND99-1376.

[16] Karen D. Devine, Bruce A. Hendrickson, Eric Boman, Ma tthew St. John, and Courtenay Vaughan. *Zoltan: A Dynamic Load Balancing Library for Parallel Applicat ions; User's Guide.* Sandia National Laboratories, Albuquerque, NM, 1999. Tech. Report SAND99-1377.

[17] Elmasri and Navathe. *Fundamentals of Database Systems Second Edition.* The Benjamin/Cummings Publishing Company, Inc., 1994.

[18] Thomas Fahringer. Estimating and optimizing performance for parallel programs. *Computer*, 28(11):47–56, November 1995.

[19] R. Fatoohi and S. Weeratunga. Performance evaluation of three distributed computing environments for scientific applications. In *Proceedings Supercomputing '94*, pages 400–409, Washington, DC, USA, November 1994.

[20] J. E. Flaherty, R. M. Loy, P. C. Scully, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Load balancing and communication optimization for parallel adaptive finite element methods. In *XVII International Conferece of the Chilean Computer Society*, Valparaiso, Chile, November 10 - 15 1997.

[21] J. E. Flaherty, R. M. Loy, M. S. Shephard, B. K. Szymanski, J. D. Teresco, and L. H. Ziantz. Adaptive local refinement with octree load balancing for the parallel solution of three-dimensional conservation laws. *Journal of Parallel and Distributed Computing*, 47:139–151, 1997.

[22] Joseph E. Flaherty, Raymond M. Loy, Can Özturan, Mark S. Shephard, Boleslaw K. Szymanski, James D. Teresco, and Louis H. Ziantz. Parallel structures and dynamic load balancing for adaptive finite element computation. *Applied Numerical Mathematics*, 26:241–263, 1998.

[23] Joseph E. Flaherty, Raymond M. Loy, Mark S. Shephard, and James D. Teresco. Software for the parallel adaptive solution of conservation laws by discontinuous Galerkin methods. In B. Cockburn, G.E. Karniadakis, and S.-W. Shu, editors, *Discontinous Galerkin Methods Theory, Computation and Applications*, volume 11 of *Lecture Notes in Compuational Science and Engineering*, pages 113–124, Berlin, 2000. Springer.

[24] Patrick H. Fry, Jeffrey Nesheiwat, and Boleslaw K. Szymanski. Computing twin primes and brun's constant: A distributed approach. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing*, pages 42–49, Chicago, IL, July 1998. IEEE Computer Society.

[25] Rob Gordon. *Essential JNI: Java Native Interface.* Prentice Hall, 1998.

[26] Bill Gustafson. Orbital thermal imaging spectrometer (otis). Technical report, University of Washington, March 1999.

[27] Karl E. Gustafson. *Introduction to Partial Differential Equations and Hilber Space Methods, Second Edition.* John Wiley and Sons, Inc., New York, 1987.

[28] Michael T. Heath. *Visualization of Parallel and Distributed Systems*, chapter 31. McGraw-Hill, 1996.

[29] Michael T. Heath and Jennifer A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, pages 29–39, September 1991.

[30] Michael T. Heath, Allen D. Malony, and Diane T. Rover. The visual display of parallel performance data. *Computer*, 28(11):21–28, November 1995.

[31] Jukka Helin and Kimmo Kaski. Performance analysis of high-speed computers. In *Proceedings Supercomputing '89*, pages 797–808, New York, NY USA, November 1989. ACM.

[32] Jeffrey K. Hollingsworth, James E. Lumpp Jr., and Barton P. Miller. Techniques for performance measurement of parallel programs. Computer Sciences Department, University of Wisconin and Department of Electrical Engineering, University of Kentucky.

[33] Jeffrey K. Hollingsworth, Barton P. Miller, Marcelo J. R. Gonalves, Oscar Naim, Zhichen Xu, and Ling Zheng. Mdl: A language and compiler for dynamic program instrumentation. In *International Conference on Parallel Architectures and Compilation Techniques*, San Francisco California, November 1997.

[34] Anna Hondroudakis. Performance analaysis tools for parallel programs. Technical report, Edinburgh Parallel Computing Centre, The University of Edinburgh, July 1995.

[35] Raj Jain. *The Art of COmputer Systems Performance Analysis – Techniques for Experimental Design, Measurement, Simulation, and Modeling.* John Wiley and Sons, Inc., 1991.

[36] Boleslaw K. Szymanski Jeffrey Nesheiwat. Scalable performance analysis for parallel scientific computations. *Electronic Modeling*, 22(2):25 – 43, 2000.

[37] Claes Johnson. *Numerical Solution of Partial Differential Equations by the Fi nite Element Method*. Cambridge University Press, 1987.

[38] Simon Karpen. Performance visualization independant study project, January 2000.

[39] George Karypis, Kirk Schloegel, and Vipin Kumar. Parmetis: Parallel graph partitioning and sparse matrix ordering library version 2.0, June 1998.

[40] D. S. Katz, T. Cwik, B. H. Kwan, J. Z. Lou, P. L. Springer, T. L. Sterling, and P. Wang. An assesment of a beowulf system for a wide class of analysis and design software. In *Advances in Engineering Software*, volume 26, pages 451–461, July 1998.

[41] Gerhard Klimeck, Roger K. Lake, R. Chris Bowen, William R. Frensley, and Ted Moise. Quantum device simulation with a generalized tunneling formula. *Appl. Phys. Lett*, 67:2539, 1995.

[42] David J. Kuck. What do users of parallel computer systems really need? *International Journal of Parallel Programming*, 22:99–127, 1994.

[43] Kei-Chun Li and Kang Zhang. Stamp: A stopwatch approach for monitoring performance of parallel programs. Department of Computing, Macquarie University, 1996.

[44] J. Lou, C. D. Norton, and T. Cwik. A robust and scalable software library for parallel adaptive refinement on unstructured meshes. *To appear in NASA Computational Aerosciences Workshop*, 1998.

[45] R. M. Loy. *Adaptive Local Refinement with Octree Load Balancing for the Parallel Solution of Three-Dimensional Conservation Laws*. PhD thesis, Rensselaer Polytechnic Institute, Troy, New York, May 1998. UMI Company.

[46] Victor W. Mak and Stephen F. Lundstrom. Predicting performance of parallel computations. *IEEE Transactions on Parallel and Distributed Systems*, 1(3):257–270, July 1990.

[47] A.D. Malony, D. A. Reed, and H. A. G. Wijshoff. Performance measurement intrusion and perturbation analysis. *IEEE Transactions on Parallel and DIstributed Systems*, 3(4):433–50, July 1992.

[48] Jem Melton and Alan R. Simon. *Understanding SQL: A Complete Guide*. Morgan Kaufmann Publishers, 1993.

[49] Barton P. Miller, Mark D. Callaghan Jeffrey K. Hollingsworth, Jonathan M. Cargille, R. Bruce Irvin, Karen L. Karavanik, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tools. *IEEE Computer*, 28(11), November 1995. Special Issue on Performance Analysis Tools for Parallel and Distributed Computer Systems.

[50] Zhichen Xu Barton P. Miller and Oscar Naim. Dynamic instrumentation of threaded applications. In *7th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Atlanta, Georgia, May 1999.

[51] Douglas C. Montgomery, George C. Runger, and Norma F. Hubele. *Engineering Statistics*, chapter 2. John Wiley and Sons, Inc., New York, 1998.

[52] Philip J. Mucci, Shirley Browne, Christine Diane, and George Ho. Papi: A portable interface to hardware performance counters. In *Department of Defence HPCMP Users Group Conference*, Montery, CA, June 7-10 1999.

[53] Jeffrey Nesheiwat and Boleslaw K. Szymanski. Instrumentation database for performance analysis of scientific applications. In *Lecture Notes in Computer Science: Fourth International Workshop on Languages, Compilers, and Run-Time Systems for Scalable Computers*, May 1998.

[54] C. D. Norton. *Object Oriented Programming Paradigms in Scientific Computing*. PhD thesis, Rensselaer Polytechnic Institute, Troy, New York, December 1996. UMI Company.

[55] G. J. Nutt, A. J. Griff, J. E. Mankovich, and J. D. McWhirter. Extensible parallel program performance visualization. In *Proceedings of the International Workshop on Modeling Analysis and Simulation of Computer and Telecommunication Systems 3rd*, pages pp. 205–211, Durham, NC, January 1995.

[56] David R. O'Hallaron. Spark98: Sparse matrix kernels for shared memory and message passing systems. Technical Report CMU-CS-97-178, School of Computer Sciences, Carnegie Mellon University, October 1997.

[57] Can Özturan. *Distributed Environment and Load Balancing for Adaptive Unstructure d Meshes*. PhD thesis, Computer Science Dept., Rensselaer Polytechnic Institute, Troy, 1995.

[58] Peter S. Pacheco. *Parallel Programming with MPI*. Morgan Kaufmann Publishers, Inc., 1997.

[59] Terrence W. Pratt. Design of thhe godiva performance measurement system. In *LCR98: Fourth Workshop on Languages, Compilers, and Run-time Systems for Scalable Computers*, pages pp. 171–176. Springer and Verlag, 1998.

[60] Daniel A. Reed, Keith A. Shields, Will H. Scullin, Luis F. Tavera, and Christopher L. Elford. Virtual reality and parallel systems performance analysis. *Computer*, 28(11):57–67, November 1995.

[61] Diane T. Rover. Performance evaluation: Integrating techniques and tools into environments and frameworks. In *Proceedings Supercomputing '94*, pages 277–278, Los Alamitos, CA, USA, 1994. IEEE Comput. Soc. Press.

[62] Mark S. Shephard, Joseph E. Flaherty, Carlo L. Bottasso, Hugues L. de Cougny, Can Özturan, and Michelle L. Simon e. Parallel automatic adaptive analysis. *Parallel Comput.*, 23:1327–1347, 1997.

[63] Mark S. Shephard, Joseph E. Flaherty, Hugues L. de Cougny, Can Özturan, Carlo L. Bottasso, and Mark W. Beall. Parallel automated adaptive procedures for unstructured meshes. In *Parallel Comput. in CFD*, number R-807, pages 6.1–6.49. Agard, Neuilly-Sur-Seine, 1995.

[64] Amitabh B. Sinha and Laxmikant V. Kale. Towards automatic performance analysis. In *Proceedings of the 1996 International Conference on Parallel Processing*, volume 3, pages 53–60, Los Alamitos, CA, USA, August 1996. IEEE Comput. Soc. Press.

[65] Marc Snir, Steve W. Otto, Steven Huss-Lederman, David W. Walker, and Jack Dongarra. *MPI: The Complete Reference*. The MIT Press, 1996.

[66] Michael Stonebraker and Paul Brown with Dorothy Moore. *Object-Relational DBMSs: Tracking the Next Great Wave*. Morgan Kaufmann Publishers, Inc., 2 edition, 1999.

[67] J. D. Teresco. Pmdbtool. Personal communication with J. D. Teresco, 1998,2000.

[68] J. D. Teresco. *A Hierarchical Partitioning Model for Parallel Adaptive Finite Element Computation*. PhD thesis, Rensselaer Polytechnic Institute, Troy, New York, August 2000. UMI Company.

[69] Abdul Waheed, Vincent F. Melfi, and Diane T. Rover. A model for instrumentation system management in concurrent computer systems. In *Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences*, pages 432–441, Los Alamitos, CA, USA, 1995. IEEE Comput. Soc. Press.

[70] Hui Wang. Automated Instrumentation of Scientific Applications . Master's thesis, Rensselaer Polytechnic Institute, Troy, New York, May 1999.

[71] J. Wang, P. Liewer, and V. Decyk. 3d electromagnetic plasma particle simulations on a mimd parallel computer. *Computer Physics Communications*, 87:35 – 53, 1995.

[72] J. C. Yan. Performance tuning with aims – an automated instrumentation and monitoring system for multicomputers. In *Proceedings of the 27th Hasaii International Conference on System Sciences*, volume II, pages 625–633, January 1994.

APPENDIX A
Program Database Definitions

A.1 Database Schema

Field	Type	Length
cfh	varchar()	80
name	varchar()	80

Table A.1: CFH Table. This table was added to support multiple control flow hierarchy instances. Relations are joined with this table to lookup the name of a specific control flow hierarchy ID. This is analagous to Ttable-A.4.

Field	Type	Length
cfh	varchar()	80
lid	int8	8
id	int4	4
min	float8	8
max	float8	8
avg	float8	8
sdev	numeric	30.6
count	int8	8
noise	float8	8
type	int8	8
proc	int8	8
cpu_acc	float8	8
cpu_min	float8	8
cpu_max	float8	8

Table A.2: PROBE Table. This table contains the statistical data for all active probes. The *cfh* attribute indicates to which CFH instance a probe belongs. This was added to support the case of multiple CFH instances. A composite key (lid, proc) uniquely identify all probes.

Field	Type	Length
cfh	varchar()	80
proc	int8	8
parent	int4	4
child	int4	4

Table A.3: CONNECTIONS Table. This table describes the connectivity of the control flow hierarchy. This table is meant to be queried to ascertain ancestry relationships betwen multiple probes.

Field	Type	Length
name	varchar()	80
lid	int8 not null	8

Table A.4: LOOKUP Table. This table maps probe IDs to their descriptions in the same way as Table-A.1

A.2 Sample Queries

This query returns statistical information for a probe on processor 0. Note that the total time spent is a derived attribute resulting from the product of the average time and the number of probe activations.

```
Q1 =  SELECT   probes.lid, name, count, cpu_acc,
              (avg*count) AS total_time
      FROM     probes, lookup
      WHERE    probes.lid = lookup.lid AND proc=0;
```

Q1 returns the following relation when run against the program database derived from runing the OTIS thermal emissivity code.

```
lid|name            |count|cpu_acc|total_time
---+----------------+-----+-------+----------
  0|Program         |    1|   1.96|  5.071973
200|SetUp(void)     |    1|      0|  0.001788
700|Master(void)    |    1|    1.9|  5.010868
900|RowMaster(void)|    1|    1.9|  5.010798
(4 rows)
```

This next query returns the total time spent in the foo() function across all processors.

```
Q2 =  SELECT    name, avg*count AS total
      FROM      probes, lookup
      WWHERE    probes.lid=lookup.lid AND lookup.name='foo()'
```

Q3 returns the processor on which foo() took the longest time to execute.

```
Q3 =  SELECT    proc
      FROM      probes, lookup
      WHERE     probes.avg=max('foo()') AND probes.lid=lookup.lid
```

A.3 SQL Function Definitions

```
CREATE FUNCTION   sdev(int4) returns numeric
AS                'SELECT sqrt( abs( avg((avg*avg))-(avg(avg) * avg(avg))))
                  FROM probes
                  WHERE id = $1;'
LANGUAGE 'sql';
```

Standard Deviation Definition

```
CREATE FUNCTION   minprobe(int4) returns float8
AS                'SELECT min(avg)
                  FROM probes
                  WHERE id = $1;'
LANGUAGE 'sql';
```

Returns the minimum time the given probe spent executing.

```
CREATE FUNCTION   maxprobe(int4) returns float8
AS                'SELECT max(avg)
                  FROM probes
                  WHERE id = $1;'
LANGUAGE 'sql';
```

Returns the maximum time the given probe spent executing.

```
CREATE FUNCTION   maxnoise(int4) returns float8
AS                'SELECT max(noise)
                  FROM probes
                  WHERE id = $1;'
LANGUAGE 'sql';
```

Returns the maximum noise for all probes on all processors.

APPENDIX B
Experiment Definition Files

```
#
# otis_coalesce.edf
#
[PREAMBLE]
Experiment      4 Node OTIS run on RPI CS Cluster using small.cfg
Mode            COALESCE
Analyst         Jeffrey Nesheiwat

[DESCR]
        Determine regularity of OTIS by running 6 times consecutively
        on ba2.cs.rpi.edu

        Commandline:    mpirun -np 4 otis -f small.cfg
[AUTOINST]
        [              Binary Data Omitted              ]
        [       --automated instrumentation tool--      ]
        [          --saved state information--          ]
[DBLIST]
        run1    otis_run1 using small.cfg on ba2
        run2    otis_run2 using small.cfg on ba2
        run3    otis_run3 using small.cfg on ba2
        run4    otis_run4 using small.cfg on ba2
        run5    otis_run5 using small.cfg on ba2
        run6    otis_run6 using small.cfg on ba2
```

Figure B.1: Experiment definition file describing a suite of six runs of OTIS using 4 processors.

```
#
# otis_compare.edf
#
[PREAMBLE]
Experiment      4 Node OTIS runs on RPI CS Cluster using master.cfg,
                masterlu.cfg, equal.cfg and equallu.cfg
Mode            COALESCE
Analyst         Jeffrey Nesheiwat

[DESCR]
        Compare performance of four processor OTIS run on ba2.cs.rpi.edu.
        master: Decomposes input acros (n-1) processors. The remaining
        processor is used to colate results.  masterlu: Same as master
        with lookup tables enabled.  equal: Decomposes input across all
        processors such that workload is equally distributed.  Results
        colated upon completion.  equallu: Same as equal with lookup
        tables enabled.

        Commandline:    mpirun -np 4 otis -f myconfig.cfg
[AUTOINST]
        [              Binary Data Omitted              ]
        [       --automated instrumentation tool--      ]
        [          --saved state information--          ]
[DBLIST]
        master          master/slave decomposition
        masterlu        master/slave decomposition with lookups
        equal           equal distribution
        equallu         equal distribution with lookups
```

Figure B.2: Experiment definition file describing comparison of 4 OTIS runs on 4 processors.

```
#
# pic3d_pow.edf
#
[PREAMBLE]
Experiment      Compare 8 node run on SP2 with POW removal
Mode            COMPARE
Analyst         Jeffrey Nesheiwat

[DESCR]
Removed 10 cals to pow() from plasma_advance() for loop by
doing multiplication manually.  The loop iterates 10e7
times, thus 10e8 calls are avoided.  How does this change
effect performance of the function, and of the program?

[AUTOINST]
        [              Binary Data Omitted             ]
        [       --automated instrumentation tool--     ]
        [          --saved state information--         ]
[DBLIST]
        plasma4  Original
        plasma5  Removed POW
```

Figure B.3: Experiment definition file describing two 3D PIC runs to test how removal of `pow()` function calls effect performance.

```
#
# rtbox-short-8.edf
#
[PREAMBLE]
Experiment      8 Nodes on SP2 with ITB, PSIRB & OCTPART
Analyst         Jeffrey Nesheiwat
Mode            COMPARE
[DESCR]
This compares 3 runs consisting of 8 processors
using ITB, PSIRB, and OCTPART load balancing.

[AUTOINST]
        [              Binary Data Omitted             ]
        [       --automated instrumentation tool--     ]
        [          --saved state information--         ]
[DBLIST]
rt_itb_o_8      8node ITB
rt_psirb_o_8    8node PSIRB
rt_oct_o_8      8node OCTPART
```

Figure B.4: Experiment definition file describing 3 RTBOX runs to test ITB, PSIRB and OCTPART load balancing schemes effect performance.

```
#
# nemo.edf
#
[PREAMBLE]
Experiment      NEMO: 12 nodes on nimrod.jpl.nasa.gov
Analyst         Jeffrey Nesheiwat
Mode            COMPARE
[DESCR]
This experiment compares a reference version of NEMO with
3 optimized version.  Optimizations include:
o Declare local data as static
o #define enumerated types
o In-lining

[AUTOINST]
        [              Binary Data Omitted             ]
        [       --automated instrumentation tool--     ]
        [          --saved state information--         ]
[DBLIST]
nemo9 Reference implementation
nemoa Static version
nemob Enumerated type as macro version
nemoc In-line version
```

Figure B.5: Experiment definition file describing reference NEMO run and three optimized versions.

APPENDIX C
Class Definitions

```
class CFH {

    // Public methods define end user's API
    public:
        char *obj_name;           // Name associated with CFH
        CFH();
        void init(char*,int,int); // Name CFH and processID
        void start(int,int);      // Start probe given type and ID
        void name(int, char*);    // Name probe based on ID
        void stop(int);           // Stop probe with ID
        void dump(int);           // Output all or specific probe
        static void dumpall();    // Dump all active CFH's.
    private:
        static int num_CFH;     // Number of CFHs on node
        int CFH_id;             // Unique CFH ID (on node)
        int proc;               // Processor ID
        int nproc;              // Total processors
        double w1,w2;           // Noise collection
        IDB_Probe_ptr *array;   // Array of probe pointers
        int num_probes;         // Number of probes
        IDB_Stack active;       // Active probe stack
        int add(int);           // Create a new probe
        void link(int, int);    // Connect parent/child probes
        int map(int);           // Map logical ID to real ID
        static CFH **cfh_list;
        static int num_cfhlist;
};
```

Figure C.1: cfh.h

```
class Probe {
public:
    List parents, children;   // parent and children lists
    int id;                   // logical id -- user defined
    Probe();
    void start(int,int,int);
    void stop();
    void setname(char *);
    void addnoise(double);
    void dump(int,char*);

private:
    int xid;                  // internal probe id
    int type;                 // probe type (LOOP, CALL, etc.)
    double t1, t2;            // measured time
    double w1, w2;            // measured noise
    double c1, c2, c;         // measured CPU time
    double cpu_acc;           // accumulated CPU time
    char *name;               // description of probe

    // statistical data
    long count;
    double avg_sq;
    double time, instr_acc, avg, min, max, sdev;
    double cpu_min, cpu_max;
};
typedef Probe * Probe_ptr;    // define pointer to probe type
```

Figure C.2: probe.h

```
class IDB_List {
public:
    IDB_List();
    void addunique(int);
    void print();
    int  exists(int);
    int  getfirst();
    int  getnext();
    int  size();

private:
    int *list;
    int num;
    int current;
};
```

Figure C.3: idb_list.h

```
class IDB_Stack {
public:
    IDB_Stack();
    void push (int);
    int  pop();
    int  size();
    void print();
    int  tos();
private:
    int num;
    int *stack;
};
```

Figure C.4: idb_stack.h

```
#define MAX 128
#define TRUE 1
#define FALSE 0

#define BUNDLE 64

#define INIT    0
#define START   1
#define STOP    2
#define DUMP    3
#define DUMPALL 4

#define PROC    1
#define LOOP    2
#define CALL    3
#define COMM    4
/*  DUMP MODES */
#define TXT 0
#define SQL 1
#define SQLFILE "populate.sql"
```

Figure C.5: idb_defs.h