EXTENSIBLE SOURCE-LEVEL INSTRUMENTATION TOOL FOR PERFORMANCE MEASURMENTS

By

Jonathan Chen

A Thesis Submitted to the Graduate
Faculty of Rensselaer Polytechnic Institute
in Partial Fulfillment of the
Requirements for the Degree of
MASTER OF SCIENCE

Approved:

_____

Dr. Boleslaw K. Szymanski
Thesis Adviser

Rensselaer Polytechnic Institute
Troy, New York

May 2007

CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

In the area of parallel computing, the complex and computationally demanding nature of scientific applications as well as the increasing prevalence of parallel computers has dramatically increased the number of parallel applications. As optimizing parallel code can be time consuming, there are often ineffiencies that are left unresolved in parallel application codes. The purpose of this thesis is to describe the automated instrumentation tool that the author contributed to the Instrumentation Database (IDB) [8] framework.

The IDB framework is a scalable approach to collect performance data in such a way that the problem size and run-time environment do not affect the amount of information collected. In order to collect data, IDB probes must be added to the program being instrumented. These probes collect information about the control flow of the program as well as various performance data. The data collected by the probes is subsequently stored in a database for further visualization and analysis.

Manually inserting probes into existing code can easily be the most tedious task in the process of using the IDB framework. This thesis focuses on improvements to the IDB framework by automating the task of source code instrumentation. We present our reasons for instrumentation at the source code level and demonstrate its effectiveness through an implementation of the automated instrumentation tool. We show that our instrumentation tool, when coupled with the IDB framework, enables easy analysis of parallel codes.

CHAPTER 1
Introduction

The complex and computationally demanding nature of scientific applications as well as the increased prevalence of parallel computers has fueled research in the area of parallel computing. Moving from conventional uniprocessor systems to multiprocessor systems, and now to multicore architectures in which a single computer contains many cores (processors), 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 [15], 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 [3] 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**, which affords more accuracy than analytic modeling. In this approach, the target system's response to the executing program is simulated. It requires fewer 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 [4].

- **Measurement**, which involves instrumenting an executing application. This is the focus of the research described in this thesis. This approach can be time consuming, but yields the most accurate results since measurements are taken on the target system [7].

1.1 Background

Traditional methods of performance instrumentation often rely on hardware or kernel support to gather performance data. Other performance instrumentation methods, such as compiler instrumentation, do not allow the programmer to specify levels of granularity of the experiments, while instrumentations involving binary modifications are often platform specific. Programmers could always insert customized probes in their source code, but the iterative process of performance tuning makes this process laborious and unscalable for large pieces of code. Manually introducing instrumentation at critical locations is a tedious and time consuming process that is also prone to errors. To keep instrumentation cost low, an automated instrumentation tool is needed to selectively instrument large codes and manage performance experiments.

1.2 Motivation and Objectives

As part of the Instrumentation Database framework [8], a method is needed to insert instrumentation probe API calls at critical locations of source code. Since our instrumentation framework needs to work for large systems, our probe insertion method must be scalable to large programs, easy to use, as well as error-proof. As scalability is a primary concern, our instrumentation tool must also allow the user to select or deselect probes at multiple levels of granularity. Additionally, instrumentation of large systems often require multiple test runs of the program, either with varying inputs to the program or with varying probes within the source code. To support this, our instrumentation tool must be able to manage multiple experiments on the same program, allowing the user to add or remove arbitrary probes between test runs. Finally, our instrumentation tool should support instrumentation of multiple programming languages.

In addition to the IDB research, other activities were ongoing that required a flexible instrumentation tool to insert custom probes into a variety of existing code [1]. Since we are creating an instrumentation tool, it became obvious that extending our automated instrumentation tool to support arbitrary probes would be beneficial.

CHAPTER 2
Previous Work

2.1 AIMS

AIMS, or Automated Instrumentation and Monitoring System [17] [16], is a tool developed by NASA Ames Research Center. It is a performance analysis framework and tool suite, including tools for source code instrumentation, run-time performance monitoring, noise reduction, and trace post-processing.

The source code instrumentation tool within AIMS automate the tedious process of source code instrumentation by offering the option to automatically insert instrumentation at each subroutine and PVM communication call. Finer grain instrumentation must be manually configured by adding or removing probes from the source code by hand. The instrumentation tool within AIMS is also only able to automatically instrument PVM programs.

As AIMS invokes the preprocessor on the source code before instrumentation, instrumentation must be done on the platform on which the code will be compiled, and the trace results from experiments are not as tightly coupled with the source code as the Instrumentation Database framework. In order to map the collected run-time data to source code constructs, source level instrumentation is stored in a flat file that resides at the beginning of run-time instrumentation trace files.

AIMS gathers many performance metrics at every instrumentation point, including time stamps, processor ID, event type and additional event-specific data. These event types include:

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

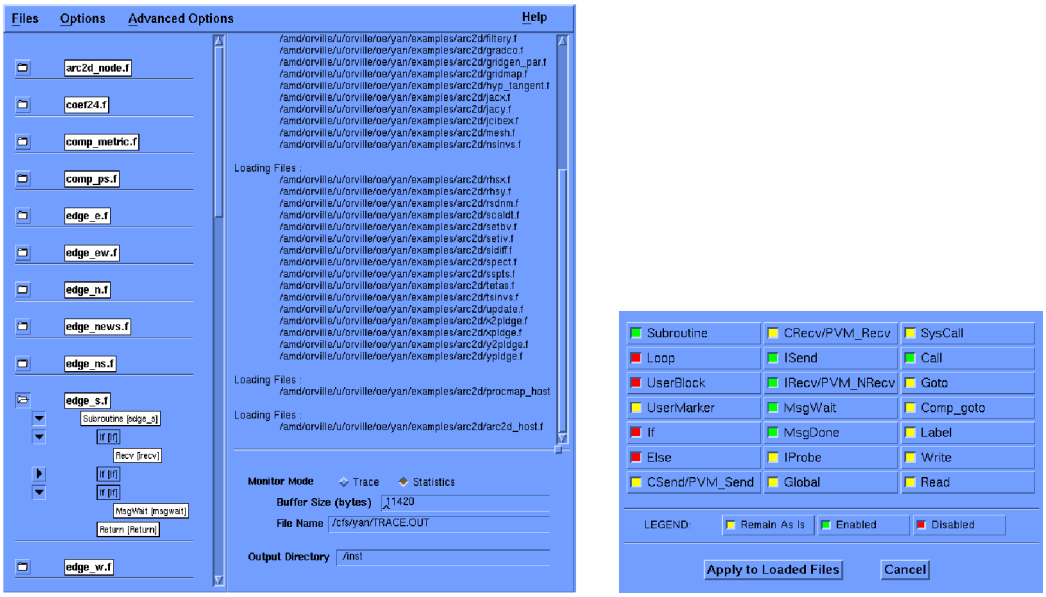

Figure 2.1: AIMS xinstrument with dialog box for type selection

As AIMS keeps track of trace information for every instrumentation point, it is not scalable. Long running programs can make the trace file very large. While AIMS uses a powerful noise reduction postprocessor on its trace file outputs, it does not have a facility to compare or coalesce trace files from different experiments, which can be useful for comparing performance across different platforms.

2.2 Paradyn

Paradyn [6] uses dynamic instrumentation to reduce instrumentation overhead. Paradyn is ideal for analysis of long running programs. When a program is instrumented, Paradyn dynamically adds and subtracts probes based on a user configurable metric-focus grid. This grid specifies the metric to be considered, such as CPU utilization or memory usage, and the focus of the search, which identifies the resources of the program that the user is interested in. The user can restrict the focus in a hierarchical manner to any synchronization objects, portions of the source code, threads, processors or time. In addition to instrumenting and visualizing performance, Paradyn attempts to draw analytic conclusions about performance bottlenecks. Once a performance bottleneck is found, Paradyn can dynamically add finer grain instrumentation in order to pinpoint the location of the bottleneck.

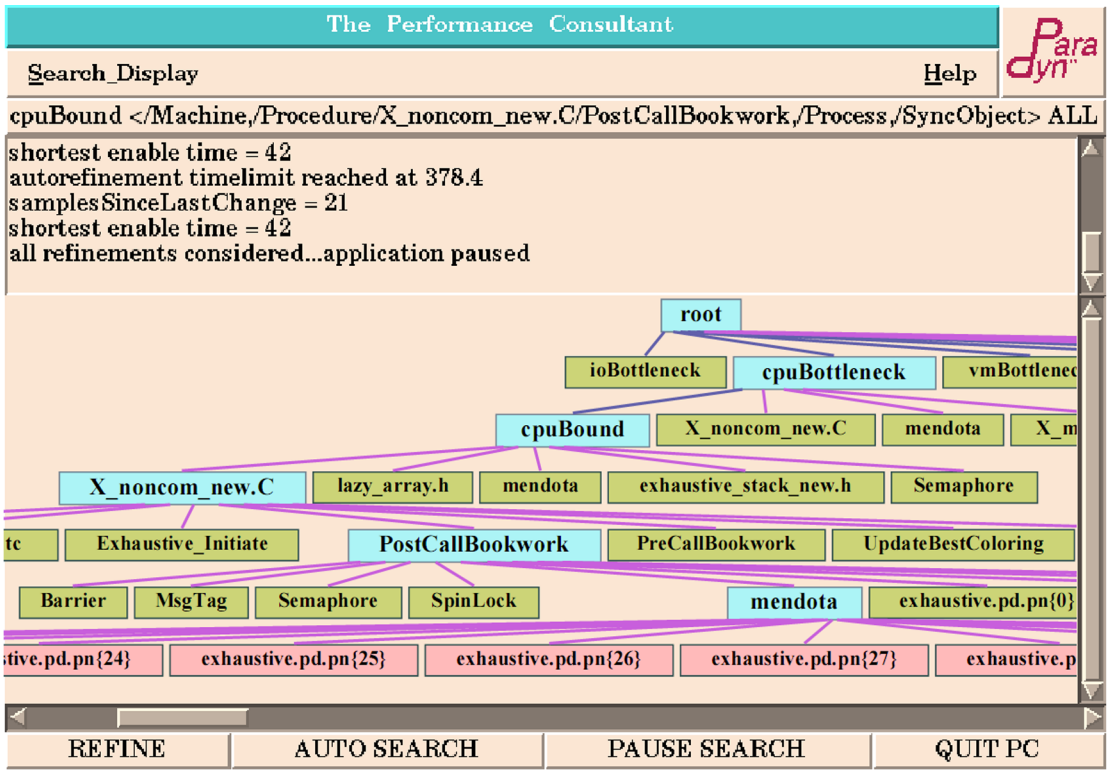

Figure 2.2: Paradyn Performance Consultant Search through W^3 for Graph Coloring Program

Once instrumentation has been inserted into an application, counters are periodically sampled by Paradyn's Data Manager. The data collected can then be analyzed by the Performance Consultant or visualized through any number of applications using Paradyn's visualization interface.

To determine why the application is performing the way it is, Paradyn represents potential performance problems using hypotheses and tests. Each of these fundamental 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. To determine where a performance problem occurs, a program is organized into a hierarchy, where the location of a bottleneck can be refined by searching a node's children. Finally, Paradyn determine when a performance bottleneck occurs by focusing on testing the hypotheses during different intervals of time in a program's execution. The Why, Where and When questions represent Paradyn's W^3 search model.

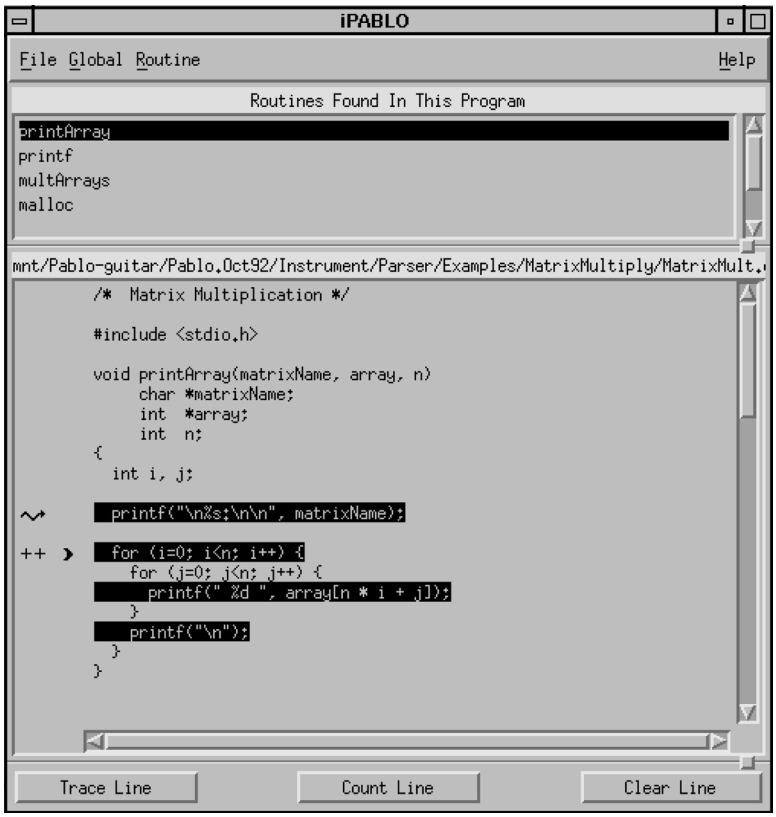

Figure 2.3: iPablo instrumentation software

2.3 Pablo

Pablo [12] [11] is a performance analysis environment that focuses on supporting portability, scalability and extensibility. Pablo supports three classes of instrumentation events: tracing, counting and time intervals. In order to reduce the likelihood of malignant perturbations caused by instrumentation probes, the Pablo trace library can automatically substitute the initially configured probes with less invasive data recording techniques if the event rate exceeds a user specified threshold. Pablo can also use dynamic statistical clustering techniques to reduce the amount of data collected. Of particular note is Pablo's data analysis and presentation software, which includes four-dimensional scatter plots seen through immersion in a virtual world, as well as sonification components, in addition to a myriad of analysis modules.

Pablo source code instrumentation is done through the iPablo [14] software, a graphical instrumentation environment that allows users to interactively select rou-

tines to instrument. It also allows the user to manually select lines in the source code for instrumentation. Unlike IDB, iPablo does not distinguish between procedures that are defined and procedure calls. iPablo also does not highlight other functional blocks such as loops and synchronization calls. iPablo can parse C source code and uses the system C preprocessor to aid in parsing. The instrumented files that it generates are preprocessed.

CHAPTER 3
Instrumentation Tool Implementation

Our instrumentation tool framework includes several functional components. In order to instrument at the source code level, a parser is needed to identify the critical blocks within the program and to locate the points of code insertion. Additionally, a user interface component makes adding and removing of probes a trivial task. Furthermore, an Experiment Definition File format is defined to ease the management of multiple experiments done on the same program. Lastly, a standard probe instrumentation library and visualization tool completes the performance instrumentation suite.

3.1 Usage Overview

The user begins by selecting C, C++ or Java files from the source tree using the file selection window shown in Figure 3.1. Selected files are then parsed using the tool's internal C/C++ or Java parser. The instrumentation window shown in Figure 3.2 presents the user with several views:

- **Function Selection View** – Lists all functions and methods defined in the selected files. In this view, the user can select the functions to be instrumented. When a function is highlighted, the source code of the function is shown in the Source Code View. The Loop and Call selection views are also updated to reflect the loops and function calls in the function highlighted. Left clicking on the function selects or deselects it for instrumentation, while right clicking on a function highlights a function without changing its instrumentation state.

- **Loop Selection View** – Lists all loop structures within the currently highlighted function. Here, a user can select the loops to be instrumented. When a loop is selected, its corresponding source code is highlighted in the source code view. In addition to turning instrumentation of a loop on and off, a probe can be designated as internal or external. Internal instrumentation of a

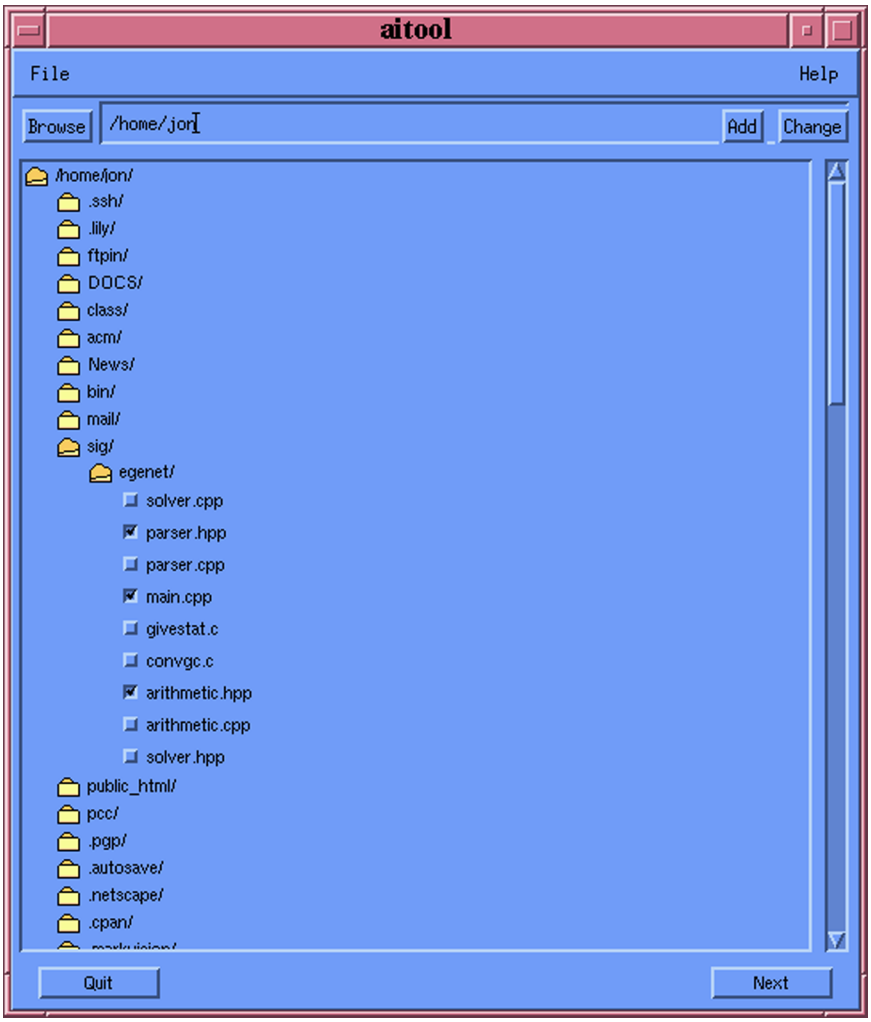

Figure 3.1: Automated Instrumentation Tool: File selection dialog

loop involves placing the API probe calls within the body of 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 highlighted function. When a function call is selected to be instrumented, probes are added immediately before and after the function selected by the user. Here, a user can select instrumentation down to a very fine granularity. Individual instances of a function call can be selected for instrumentation. Selecting a function call from this list will also highlight its corresponding source code in

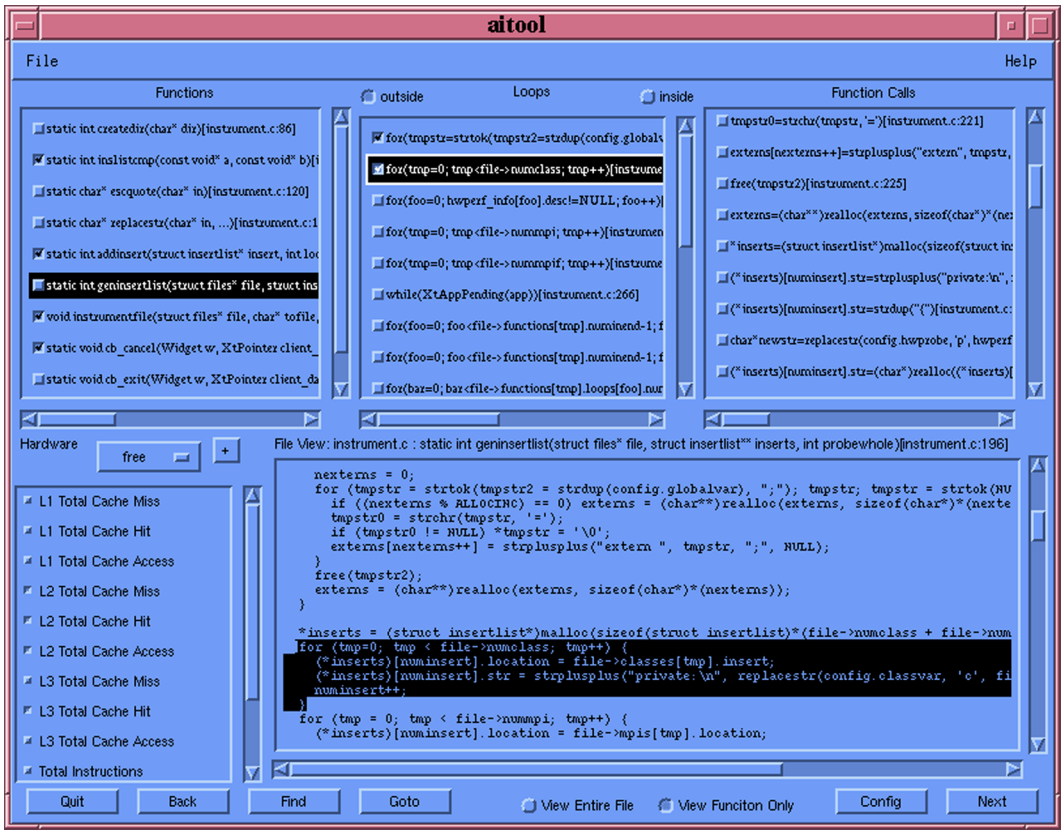

Figure 3.2: Automated Instrumentation Tool: Instrumentation window

the source code view.

- **Source Code View** – Presents a context-sensitive view of application source code. This view can be constrained to individual functions or files. When loops or function calls are selected, their corresponding locations in the source code are highlighted, allowing users to view the code being instrumented in its context.

In addition to the four main views, the instrumentation window provides a search feature to allow for keyword search within all selected source files. The Goto feature also allows for the quick location of a function or loop based on the file name and line number of the source code. Because the automated instrumentation tool uses a built in parser that understands pre-preprocessed source code, searches can be performed on the actual source file and will match macros and comments. Line numbers of the parsed files are also consistent with their input files.

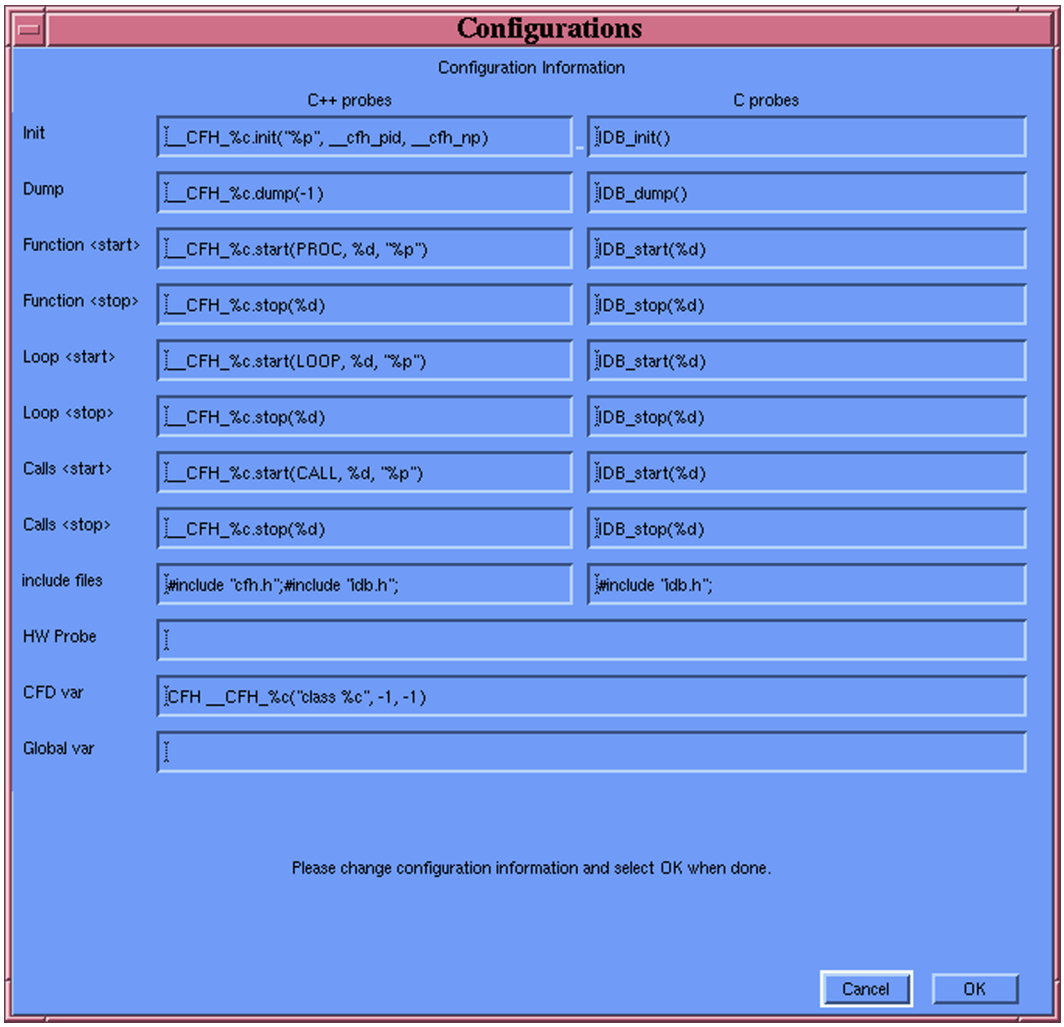

Figure 3.3: Automated Instrumentation Tool: Configuration window

In the instrumentation window, there is also a list of hardware counters. In addition to the normal collection of control flow data and execution times, IDB can leverage PAPI [2] for the collection of hardware instrumentation counters. Users can select from this list the counters they are interested in, and the automated instrumentation tool will configure IDB to collect the requested counters.

The automated instrumentation tool is designed to work with IDB and will normally insert IDB probes into the instrumented file. However, it can be configured to insert arbitrary API calls to meet other needs. As shown in Figure 3.3, settings in the Configuration screen allow the user to specify the probe API inserts before and after each instrumented block. Any initialization, finalization calls as well as

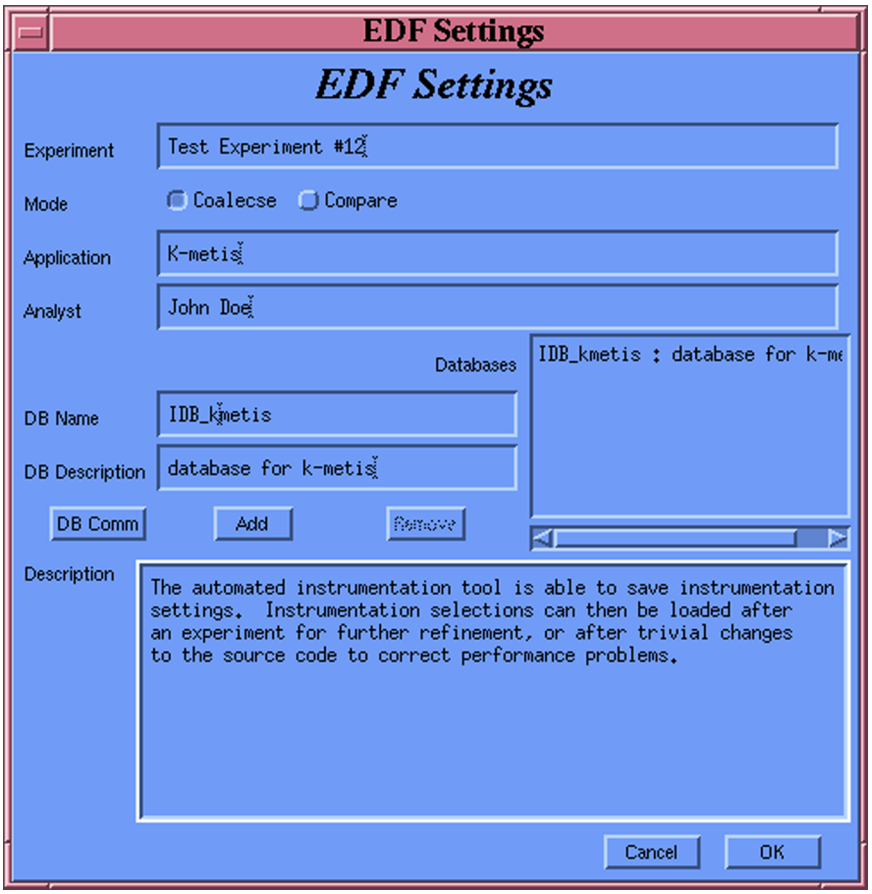

Figure 3.4: Automated Instrumentation Tool: EDF save dialog

other necessary additions can also be added to an instrumented file.

One strength of the automated instrumentation tool is its ability to save an Experiment Definition File. The save function in the file menu (figure 3.4) allows a user to save basic experiment information as well as their instrumentation selections in an Experiment Definition File. Once the information is saved, the user can then reload their settings at a later time. This allows a user to tune and refine the instrumentation after an experiment. Because the automated instrumentation tool recognizes instrumentable blocks based on functions, loops and function calls, a user can even reload the same settings on source code that has been modified. Trivial changes, such as those made to correct performance bottlenecks, often maintain the same basic structure as the original code, thus the automated instrumentation tool can automatically locate the same functions, loops and calls and select them for

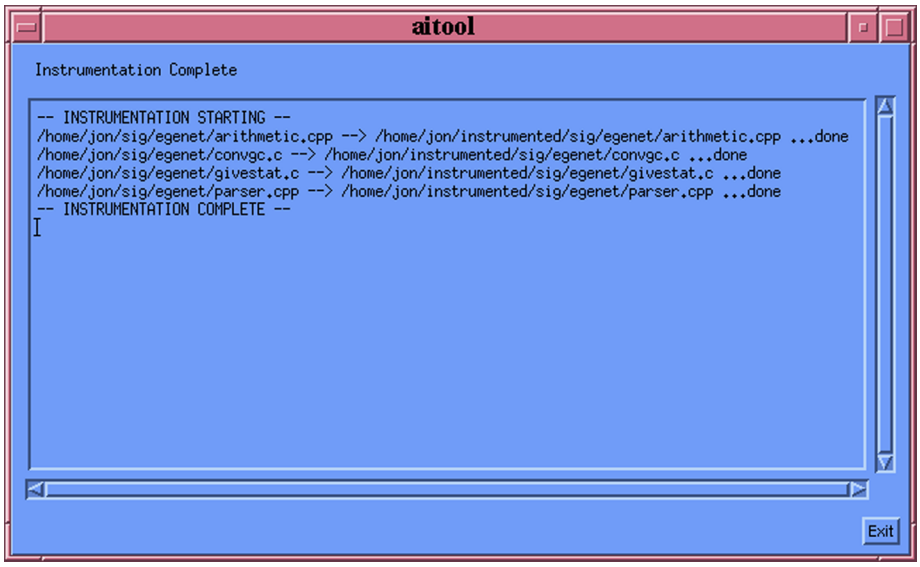

Figure 3.5: Automated Instrumentation Tool: Completion dialog

instrumentation. The Experiment Definition File and its uses are further described in section 3.4.

Once a user finishes selection of what to instrument, instrumentation is started by clicking the Next button. The resulting dialog (Figure 3.5) shows the user the status of the instrumentation, the placement of the instrumented files as well as any errors that may have occurred during the instrumentation. Once instrumentation is complete, the instrumented code can then be compiled and ran. Because the automated instrumentation tool uses an internal parser, the file structure, comments and line numbers in the original source code are maintained in the instrumented output. This can be useful if the user needs to further add, subtract or modify probes after the automated instrumentation tool has generated its output. This also aids in debugging as the source code associated with the executable is highly readable to humans.

3.2 Source code parser

Choosing to instrument at the source code level allows a program to be instrumented without regard to the hardware platform, operating system or compiler usage. Unlike binary instrumentation, source code instrumentation allows us to

locate important source level language constructs without regard to optimization settings at the compiler. Instrumentation at the source code level also allows for the highest possible degree of mapping between performance events and the source code. While instrumentations at the binary level can be dynamic, they are also more costly due to the additional overhead incurred by the needed support of dynamic instrumentation. Source level instrumentation can be easily implemented and is applicable at variable levels of granularity with minimal impact on the application.

Two source code parsers are included with the automated instrumentation tool, supporting the C/C++ and Java languages. Both of these parsers are LALR(1) parsers and are designed to accept as liberally as possible. While it is easier and more correct to only accept code strictly according to the language definition, this would require preprocessing of include files and macros to be done prior to our parsing. To present the programmer with the closest possible view to the original source, parsing of include files and other preprocessor evaluations are not attempted. Due to the extensibility of the instrumentation tool framework, additional parsers could easily be implemented that more robustly parse the existing sets of languages, as well as to parse new languages.

Adding instrumentation to a syntactically correct program before preprocessing is not a straight-forward process. Several challenges need to be overcome to ensure maximum usability and correctness of results.

One problem is how to ensure that a probe is stopped before the function or program terminates. In C, special attention needs to be paid to language constructs such as *break*, *continue*, *return()* and *exit()*. Additionally, some library functions may have side effects that affect program flow and need to be treated specially, such as *fork()*, *pthread_*()* or *MPI_Init()*. The automated instrumentation tool will automatically locate these functions and automatically insert appropriate API calls around these functions.

Another problem is caused by the fact that inserting API calls at arbitrary locations in the source may alter program semantic. Consider the following C code snippet:

```
if (test)
```

```
  a = foo();
else
  b = bar();
```

Adding a probe around foo() may alter the semantics of the program. The two ways around this problem is either with the use of the comma operator, or by wrapping the instrumented code in a new block. Since the use of the comma operator is not allowed everywhere a semicolon can be (particularly at the end of loops), we put all Call probes into new blocks:

```
if (test)
  { PROBE_START; a = foo(); PROBE_END; }
else
  { PROBE_START; b = bar(); PROBE_END; }
```

Other problems are caused by the fact that we do not preprocess the code prior to parsing it. As we do not know what preprocessor macros will be defined at the compile time, we must process all **#if** and **#ifdef** blocks – both the positive and the **#else** sides (the exception being **#if 0**, which we always ignore.) As a result, the programmer must ensure that the source code is syntactically parsable when all sides of **#if** blocks are parsed in their written order. When necessary, a programmer may manually insert blocks of dummy code in **#ifdef NEVER_DEFINED** sections to manually correct the parsing of the automated instrumentation tool. In practice, we find that this is rarely necessary.

Another preprocessor difficulty is caused by the use of macros. Since the automated instrumentation tool does not expand macros, it may fail to parse around macro invocations that do not conform to the expected syntax of the language at the context at which the macro was invoked. Parsing difficulties may also arise if a macro contains an unbalanced open brace or if the invocation of a statement-like macro is not followed by a semicolon. In most macro use cases, programmers tend to follow the generic syntax of the language and this is not an issue. The parser is also often able to automatically recover from such errors and to continue parsing on its own.

Other uses of macros may obscure important elements from the parser, such as the use of a macro to declare a function or a loop. Additionally, macros may contain important language constructs such as *return()*, which will not be properly recognized by the parser. While an obscured function or loop generally poses very little problem (probes can always be inserted manually), missed *return()* and similar calls may cause a probe to never be closed. Underlying probe libraries, such as IDB [8], can detect this condition and automatically corrects it.

3.3 Instrumentation

After an input file is parsed and instrumentation is selected, probes are added to the source code by the automated instrumentation tool. In order to maintain maximum human readability of generated code, changes from the original file are kept at a minimum. Macros are not expanded, and comments spacing and indentation are kept intact. When the input file is parsed, information concerning the character offset of blocks of code is obtained, thus probes can be inserted in the same line immediately preceding and following the instrumented code. New lines are in general not added to files, except as required by the language (such as new #include directives).

3.4 Experiment Definition File

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.
- **Execution** – Permuting run-time parameters across multiple systemic 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 a formal structure by centralizing information for conducting performance experiments. The automated instrumentation

tool provides a mechanism where EDF files can be loaded and saved. 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 for storage of results 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.
- **Instrumentation tool state** – State information is automatically generated by the automated instrumentation tool and saved as part of the Experiment Definition File. These states describe where instrumentations were introduced. Specifically, it contains a snapshot of which functions, loops, and calls are selected for instrumentation. This allows the user to save a snapshot of the current work and resume adding or removing instrumentation later. Users can also use this feature to add, subtract or modify instrumentation between experiments, or even after modifications to the source code to address a performance problem.

CHAPTER 4
Application

4.1 Particle in Cell Simulation Code

In this chapter, we describe an example of the application of IDB and its instrumentation tool to a parallel object oriented application. In this example, we show the benefits of an instrumentation tool that is able to iteratively refine placements of instrumentation probes, as well as compare instrumentation data after a refinement to the program.

4.1.1 Background

Particle in cell, or PIC, [10] codes are used to simulate spatial non-linear kinetic systems. The 3D PIC code in this example [9] models plasma as particles in a self-consistent electro-magnetic field. In each time-step of its execution, our PIC code updates each particle position and recomputes their charge / current density. Then, the electromagnetic field is recomputed based on particle positions computed in the previous step.

4.1.2 Instrumentation

Our 3D PIC code was instrumented using the automated instrumentation tool. Since we begin the process with no knowledge of the performance characteristics of the PIC code, minimal instrumentation was initially added to determine the scalability of the application. Gradually, probes were added to determine the critical path and to locate potential bottlenecks. This is done by instrumenting all loops and function calls within the main event loop, and proceeding further down the critical path once it is located.

4.1.3 Results

Initial analysis shows that the PIC code is scalable with 99% efficiency when ran on an IBM SP2 with 32 nodes. Further, investigation down the critical path of

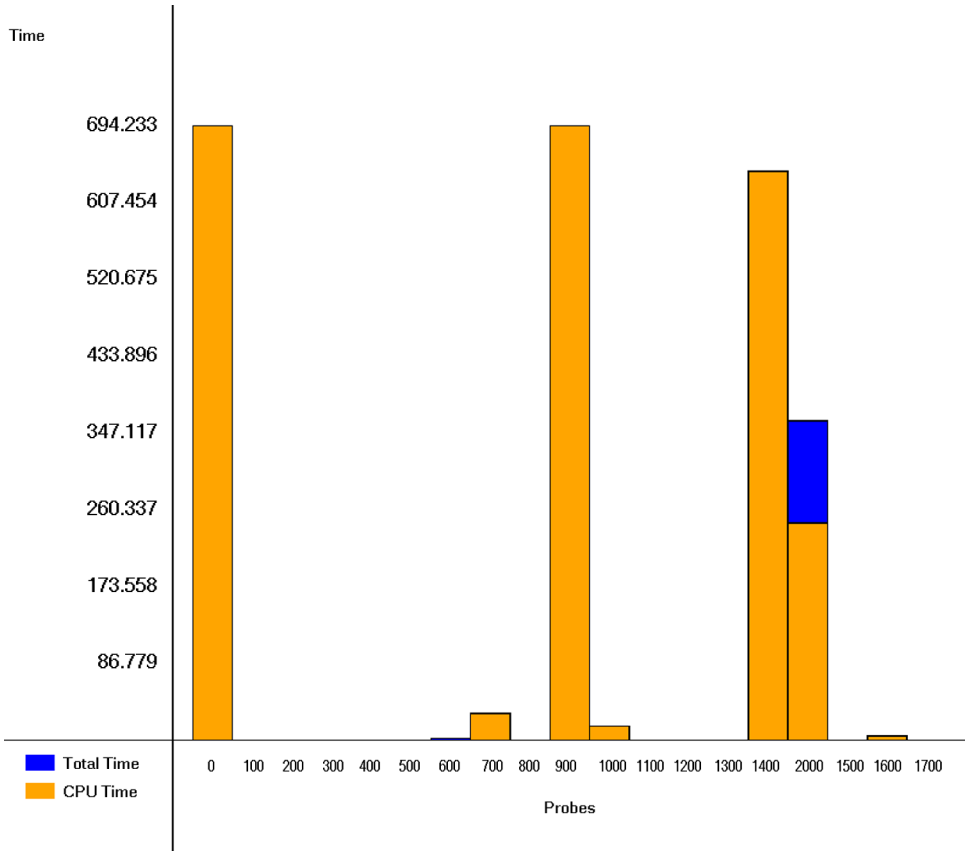

Figure 4.1: Total execution time and CPI time for 3D PIC code

the program reveals that the plasma_advance() function (probe 1400 in figure 4.1) consumes 92.6% of the execution time. Within that function, there is a main loop (probe 2000 in figure 4.1) which is executed close to 10^7 times and consumes almost half the program's execution time, with a significant difference between clock time and CPU time. Our optimization efforts thus focus on this loop. Examination of the source code within this loop reveals that the pow() function is used multiple times to square simple mathematical expressions. Replacing the costly function with simple multiplication results in an 18.1% speedup in the function, which translates to a 6.1% speedup in total execution time.

4.2 Improving Spatial Locality of Programs

The following example, based on [13], demonstrates the use of the automated instrumentation tool to collect much more data than the original Instrumentation

Database tool intended. We can see here the ease of obtaining customized instrumentation data by plugging in a different set of instrumentation libraries.

4.2.1 Background

A computer's main memory is typically fast and relatively small, while the secondary storage (disk) is slower and larger. Since a large program may not completely fit in the main memory, performance improvements can be made to a program by grouping certain functions together within a page. Performance gains can also be obtained with smaller programs as cache behaves in a very similar way. The experiments described here are designed to find an optimal placement of code within a binary executable with respect to spatial locality. The algorithms investigated data mine the temporal locality information from some sample executions as input.

4.2.2 Instrumentation

In addition to the standard "when" and "how long" aggregate probes, our need for detailed temporal locality information necessitates an execution trace that includes all functions. A new probe library was written that collects and logs the necessary information. As the sizes of functions are also needed to determine the best fit, the symbol table of an uninstrumented executable is consulted for a list of functions to be instrumented together with their sizes. Using the automated instrumentation tool, the required probes are then inserted into the start of each function automatically. The new instrumented executable is then run with a variety of input parameters to produce a set of function traces.

Once all the traces have been gathered, they are fed into a data mining algorithm to obtain an optimal code placement. Two placement algorithms were tested: (i) a Greedy Clustering algorithm, which chooses functions to fill the pages based on the expected contribution towards minimization of page faults per unit byte occupied by the function in the page; and (ii) a Stochastic Optimization approach, which attempts to minimize the estimated page fault ratio.

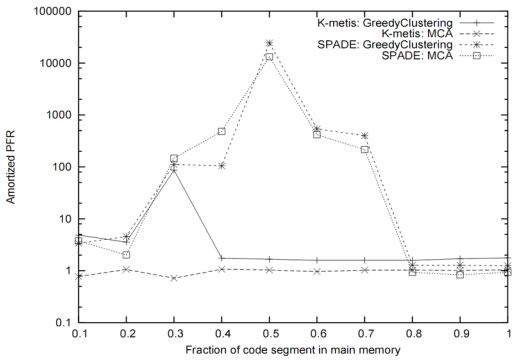

Figure 4.2: GreedyClustering v/s MCA on PFR

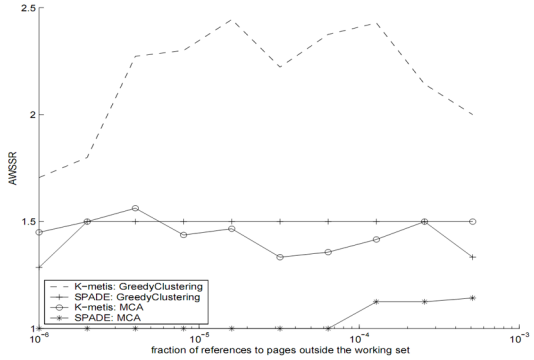

Figure 4.3: GreedyClustering v/s MCA on AWSSR

4.2.3 Results

Two data mining applications were used to validate the placement algorithms – SPADE [18] and K-metis [5]. Instrumented versions of both of these applications were ran with a variety of randomly generated input. The output of the instrumented code is a sequence of program-specific function identification names/numbers. The success of our algorithms is measured using two metrics:

- Page Fault Ratio (PFR), where if y is the fraction of code segments in main memory, then $PFR(y) = \frac{pfr_{default}(y)}{pfr_{optimized}(y)}$
- Approximate Working Set Size Ratio (AWSSR). Let $f(x)$ be a function, which returns the minimum number of pages whose cumulative references account for $(1-x)$ of the total references. As $x \to 0$, $f(x)$ is an approximation to the working set size:

$$AWSSR(x) = \frac{f_{default}(x)}{f_{optimized}(x)}$$

A high PFR implies that the optimized assignment has a considerably lower PFR and a high AWSSR implies that the optimized assignment has a smaller approximate working set than the default assignment. As we can see from figures 4.2 and 4.3, the Greedy Clustering algorithm was able to outperform the default assignment in both of our metrics.

4.3 Functional Level Memory Optimization Tool

The final example described here [19] demonstrates the suitability of the automated instrumentation tool to collect data that is quite different from the original purpose of the tool. Instead of optimizing a parallel application by examining timing information and the program control flow graph, in this example we optimize a single-threaded long-time running program.

4.3.1 Background

Similar to the previous example, the functional level memory optimization tool seeks to improve performance by rearranging a function's placement within the binary executable. In addition to rearranging a function's placement to improve temporal locality, this optimizer also attempts to reduce page faults incurred when loading functions by splitting functions such that they do not span two or more pages, and also by stuffing the binary executable in such a way that larger functions begin at page boundaries. Additionally, functions that have a high called frequency are placed together to improve the effectiveness of paging, thus decreasing frequency of page faults.

4.3.2 Instrumentation

For this optimizer, the only information needed from a test run of the program is the execution trace containing a hierarchical and ordered list of function calls. As execution speed and perturbations are not a concern, a simple probe library which wrote to a trace file on every function entry and exit was constructed. The automated instrumentation tool was then used with these probe parameters to instrument the source code. Once the instrumented code is ran and a trace file is obtained, the analysis and function stuffing and splitting applications are then used to obtain an optimized binary executable.

To show the effectiveness of this optimization tool, the Apache HTTP server was used as an example of a longer running program. This presents a challenge to the automated instrumentation tool, as many Apache function prototypes, or even entire functions, are generated by macros in the source code. To overcome

this obstacle, a limited C preprocessor was first ran on all the Apache HTTP server source code. This preprocessor expanded the troublesome functions and produced code that the instrumentation tool can understand. Once this initial preprocessing was done, the automated instrumentation tool proved able to parse and instrument all the necessary source code.

4.3.3 Results

To determine the effectiveness of the optimizer, Apache access logs were replayed against both a randomly linked server and a server that was optimized with the tool. Experiments were done on a single processor computer with 256MB of memory and a quad processor machine with 1GB of memory. The results show that for shorter runs on the machine with less memory, the performance improvement can be significant, while longer runs on the machine with more memory yield less improvement. The machine with more memory would see less speed up since its abundant memory means an executable could be paged in to main memory more rapidly, perhaps all at once at the application's execution, whereas the weaker machine may be more conservative at paging in the program. The length of a run also affects the percentage of the speedup, since memory that is paged in is less likely to be swapped out when it is used frequently.

Size of log file	Linking method	Runtime (sec)
73,519	Random	26
	Optimized	20 (-23%)
519,492	Random	206
	Optimized	175 (-18%)
3,451,435	Random	1365
	Optimized	1128 (-21%)

Table 4.1: Apache HTTP server on single processor with 256MB RAM

Size of log file	Linking method	Runtime (sec)
73,519	Random	10
	Optimized	9 (-10%)
519,492	Random	62
	Optimized	60 (-3%)
3,451,435	Random	441
	Optimized	432 (-2%)

Table 4.2: Apache HTTP server on quad processor with 1GB RAM

CHAPTER 5
Conclusions

We have shown that the automated instrumentation tool is useful for performing both iterative analysis using the IDB framework and to validate performance enhancements. Furthermore, the automated instrumentation tool was validated to work with a number of existing scientific and commercial applications. Designed using a modular design that was guided by user feedback, our tool also shows potential to be extended to support other instrumentation frameworks that require source code instrumentation. With this easy to use automated instrumentation tool, the speed at which a program can be instrumented and the chance of accidentally introducing errors is thus reduced, and the cost of improving performance of programs is decreased.

[1] B. Bouchra, C. Carothers, M. Zaki, and B. Szymanski. Understanding Filesystem Performance for Data Mining Applications. In *Proceedings of the 6th International Workshop on High Performance Data Mining: Pervasive and Data Stream Mining (HPDM:PDS'03) at the Third International SIAM Conference on Data Mining*, San Francisco, CA, May 2003.

[2] S. Browne, J. Dongarra, N. Garner, G. Ho, and P. Mucci. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*, 14(3):189–204, Fall 2000.

[3] Raj Jain. *The Art of Computer Systems Performance Analysis: Techniques for Experimental Design, Measurement, Simulation, and Modeling.* Wiley-Interscience, 1991.

[4] W. Kaplow, P. Tannenbaum, B. Szymanski, and V. Decyk. Run Time Reference Clustering for Cache Performance Optimization. In *Proc. Second Aizu Int. Symposium on Parallel Algorithms/Architectures Synthesis, Aizu-Wakamtsu, Japan*, pages 42–49, Los Alamitos, CA, March 1997. IEEE Computer Science Press.

[5] George Karypis and Vipin Kumar. Multilevel k-way hypergraph partitioning. *Proceedings of the 36th ACM/IEEE conference on Design automation*, pages 343–348, 1999.

[6] Barton P. Miller, Mark D. Callaghan, Joanthan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, November 1995.

[7] S Moore, D Cronk, F Wolf, A Purkayastha, P Teller, R Araiza, M Aguilera, and J Nava. Performance Profiling and Analysis of DoD Applications using PAPI and TAU. In *Proceedings of the DoD High Performance Computing Modernization Programs User Group Conference*, Nashville, TN, June 2005. IEEE.

[8] Jeffrey Nesheiwat and Boleslaw Szymanski. Instrumentation database system for performance analysis of parallel scientific applications. *Parallel Computing*, 28(10):1409–1449, August 2002.

[9] M. Nibhanapudi, C. Norton, and B. Szymanski. Plasma simulation on networks of workstations using the bulk-synchronous parallel model. In *Proc. Int. Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'95)*, pages 13–22, November 1998.

[10] C. Norton, B. Szymanski, and V. Decyk. Object-oriented parallel computation for plasma simulation. *Communication of the ACM*, 38(10):88–100, October 1995.

[11] Daniel A. Reed, Ruth A. Aydt, Tara M. Madhyastha, Roger J. Noe, Kieth A. Shields, , and Bradley W. Schwartz. An overview of the Pablo performance analysis environment, 1992.

[12] Daniel A. Reed, Ruth A. Aydt, Roger J. Noe, Phillip C. Roth, Keith A. Shields, Bradley W. Schwartz, and Luis F. Tavera. Scalable performance analysis: the pablo performance analysis environment. In *Proceedings of the Scalable Parallel libraries Conference, 1993.*, pages 104–113. IEEE Computer Society, October 1993.

[13] Karlton Sequeira, Mohammed Zaki, Boleslaw Szymanski, and Christopher Carothers. Improving spatial locality of programs via data mining. In *Proceedings of the ninth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD-03)*, pages 649–654, August 2003.

[14] Keith A. Shields. iPablo User's Guide, November 1992.

[15] Boleslaw Szymanski. Scalable computers. In A. Kent and J.G. Williams (exec. edts), editors, *Encyclopedia of Computer Science and Technology*, volume 39, pages 211–228. Marcel Dekker Inc., New York, 1998.

[16] Jerry Yan, Melisa Schmidt, and Cathy Schulbach. The Automated Instrumentation and Monitoring System (AIMS) Version 3.2 Users' Guide. Technical Report NSA-97-001, NASA Ames Research Center, January 1997.

[17] Jerry C. Yan. Performance Tuning with AIMS – An Automated Instrumentation and Monitoring System for Multicomputers. In *Proceedings of the Twenty-Seventh Annual Hawaii International Conference on System Sciences*, volume II, pages 625–633, January 1994.

[18] Mohammed J. Zaki. SPADE: An Efficient Algorithm for Mining Frequent Sequences. *Machine Learning Journal, special issue on Unsupervised Learning (Doug Fisher, ed.)*, pages 31–60, 2001.

[19] Lizhuang Zhao. A Functional Level Memory Optimization Tool – The Slicer. Master's thesis, Rensselaer Polytechnic Institute, July 2004.