Paradyn Parallel Performance Tools

User's Guide

Release 5.1
May 2007

Paradyn Project
Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685
paradyn@cs.wisc.edu

Table of Contents

1 Overview....1
1.1 Release notes (version 5.1)....2
1.2 Release notes (version 5.0)....2
1.3 Release notes (version 4.2)....2
1.4 Release notes (version 4.1)....2
1.5 Release notes (version 4.0)....3
1.6 Release notes (version 3.3)....3
1.7 Release notes (version 3.2)....4
1.8 Release notes (version 3.1)....4
1.9 Release notes (version 3.0)....4
1.10 Supported hardware and software platforms....6
1.11 Other documentation: Manuals....7
1.12 Other documentation: Technical papers....8
1.13 Contacting the Paradyn developers....9

2 Running Paradyn....1
2.1 Overview of major steps....1
2.2 Setting up Paradyn and the Paradyn daemons....1
2.2.1 Paradyn environment variables....1
2.2.2 Overriding Paradyn's default fonts....2
2.3 Preparing your application program....5
2.3.1 Generation of debug information (all platforms)....5
2.4 Running Paradyn....5
2.5 Running applications with Paradyn....7
2.5.1 Defining a new process....7
2.5.2 Attaching to a process....9
2.6 Architectural issues....10
2.6.1 Common Platforms....10
2.6.2 MPI....11
2.6.3 Microsoft Windows....12
2.6.4 Multi-threaded....13

3 Main Control window....1
3.1 Main menubar....1
3.1.1 File menu....1
3.1.2 Setup menu....1
3.1.3 Phase menu....2
3.1.4 Visi menu/button....2
3.1.5 Help menu....2
3.2 Status lines....3
3.3 Buttons....3

4 Tunable Constants....1
4.1 Overview....1
4.2 User Tunable Constants....2
4.3 Developer Tunable Constants....2

5 Selecting resources 1
5.1 Resources (The "Where" Axis) 1
5.2 The Where Axis display 3
5.3 How to select foci using the Where Axis 4
5.4 The Where Axis GUI 5
5.5 Call Graph display 6

6 Selecting metrics 1
6.1 How to select metrics 1
6.2 Metric Descriptions 2

7 Controlling visis 1
7.1 Starting 1
7.2 Stopping 2

8 Phases 1
8.1 Starting a new phase 1
8.2 Visualizations and Phases 1
8.3 The Performance Consultant and phases 1

9 Performance Consultant 1
9.1 The W3 search model 1
9.1.1 The Why Axis 2
9.1.2 The search strategy 3
9.2 Running the Performance Consultant 4
9.2.1 The Performance Consultant window 4
9.2.2 Starting and stopping a search 5
9.2.3 The Search History Graph display 6
9.3 Interpreting the results of callgraph-based search 7
9.4 Interpreting the results of module-then-function search 8
9.5 Customizing the search parameters 13

10 Standard visi modules 1
10.1 Time Histogram visi 1
10.1.1 File menu 2
10.1.2 Curve menu 2
10.1.3 Panning and zooming 2
10.2 Barchart visi 4
10.2.1 Changing metrics and foci being viewed 5
10.2.2 Viewing data 5
10.3 Table visi 6
10.3.1 Actions menu 7
10.3.2 View menu 7
10.4 3D Terrain visi 8
10.5 Viewing the application output display 9

11 Exporting Paradyn Data 1
11.1 Saving Performance Data 1

11.1.1 Saving Performance Data From Front-End 1
11.1.2 Saving Performance Data From the Visis 3
11.2 Saving the Where Axis 4
11.3 Saving Performance Consultant Search Data 4

12 Paradyn Configuration Language 1
12.1 Notation 1
12.2 Lexical conventions 1
12.3 Language structure 2
12.4 Daemon definition 3
12.5 Process definition 4
12.6 Tunable constant definition 5
12.7 Visi definition 6
12.8 Exclude definition 6
12.9 Metric Description Language 7
12.9.1 Metric definition 8
12.9.2 Variables 9
12.9.3 Types 9
12.9.4 Predefined variables 10
12.9.5 Resource lists 11
12.9.6 Constraints 12
12.9.7 Metric definitions 13
12.9.8 Metric statements 15
12.9.9 Metric expressions 15
12.9.10 Function calls 16
12.9.11 Instrumentation requests 17
12.9.12 Instrumentation code 17
12.9.13 Interaction of constraints and metrics 18
12.9.14 A complete example 19

List of Figures

1 Overview
Figure 1: Platforms on which Paradyn can monitor application processes 1-7

2 Running Paradyn
Figure 2: Files needed to run Paradyn 2-2
Figure 3: Environment variables used when running Paradyn 2-3
Figure 4: Starting Paradyn 2-6
Figure 5: Defining a new application process 2-7
Figure 6: Paradyn ready to run the application 2-8
Figure 7: Specifying a process to attach to. 2-10
Figure 8: Attach completed and application execution continuing. 2-11
Figure 9: Sample Makefile for Windows. 2-12

3 Main Control window
Figure 10: Paradyn Main Control window 3-1

4 Tunable Constants
Figure 11: The Tunable Constants Window 4-1
Figure 12: Tunable Constants Descriptions Window 4-2
Figure 13: User-level Tunable Constants 4-3
Figure 14: Developer-level Tunable Constants. Use at your own risk! 4-4

5 Selecting resources
Figure 15: Where Axis window. 5-1
Figure 16: Showing all resources in the Where Axis display 5-3
Figure 17: A single focus selected 5-4
Figure 18: Multiple foci selection 5-5
Figure 19: Callgraph display 5-7

6 Selecting metrics
Figure 20: Metrics dialog box 6-1
Figure 21: Metrics dialog box with several metrics selected 6-2
Figure 22: Metrics defined in Paradyn 6-3
Figure 23: Developer Mode Metrics defined in Paradyn 6-7

7 Controlling visis
Figure 24: Paradyn Main Control window 7-1
Figure 25: Start A Visualization menu 7-1
Figure 26: Message box indicating instrumentation has been deferred 7-2

8 Phases
Figure 27: Phase Table Display 8-1
Figure 28: Time Histogram: Global Phase 8-2
Figure 29: Time Histogram: Local Phase (3) 8-2

9 Performance Consultant
Figure 30: The Why Axis 9-2
Figure 31: A sample Performance Consultant window 9-4
Figure 32: The Performance Consultant's search begins 9-6

List of Figures

Figure 33: The Performance Consultant refines bottleneck to CPUbound 9-8
Figure 34: Search History Graph tunable constants for saving screen space 9-9
Figure 35: The Performance Consultant refines bottleneck beyond CPUbound 9-10
Figure 36: The second set of Search History Graph refinements 9-11
Figure 37: Final Search History Graph bottleneck refinement 9-12

10 Standard visi modules
Figure 38: Time Histogram with selected curve 10-1
Figure 39: Time Histogram with unsmoothed and hidden curves 10-3
Figure 40: Zoomed Time Histogram 10-3
Figure 41: Barchart visualization window 10-4
Figure 42: Barchart showing total values 10-5
Figure 43: Table visualization window 10-6
Figure 44: Table visualization showing short focus names 10-7
Figure 46: 3D Terrain visualization 10-9
Figure 45: Table visualization with values shown to two significant digits 10-9
Figure 47: Application output window 10-10

11 Exporting Paradyn Data
Figure 48: Export dialog window 11-1
Figure 49: Performance Data File Header and Data Format 11-2
Figure 51: Visi Menu to Export Paradyn Data 11-3
Figure 50: The Paradyn Main Window after saving performance data. 11-3
Figure 52: The Paradyn Main Window after saving resource names. 11-4
Figure 53: Format for exported search data in shg.txt. 11-5
Figure 54: Paradyn Main Window after saving Performance Consultant data. 11-5

12 Paradyn Configuration Language
Figure 55: List of MDL keywords 12-2
Figure 56: Predefined variables 12-10
Figure 57: Metric labels. 12-14

1 OVERVIEW

Paradyn is a tool for measuring the performance of parallel and distributed programs. When run with Paradyn, instrumentation is dynamically inserted into an executing application program and its performance is reported in real-time. Paradyn's features include:

- Run-time program instrumentation: you do not have to modify your source code or use a special compiler. Paradyn directly instruments the binary image of your running program.
- Performance data visualizations: Paradyn currently provides visualizations to present performance data in time-plots, bar graphs, and tables.
- Automated search for performance bottlenecks: Paradyn's Performance Consultant has a well-defined notion of bottlenecks and directs Paradyn's instrumentation in search of them.
- Multi-platform support: Paradyn currently can measure programs running on Solaris (SPARC), Linux (x86), Linux (ia64), Linux (x86-64), Windows 2000 and XP (x86), and AIX (RS6000).
- Paradyn can measure programs running on heterogeneous combinations of the above systems.
- Support for MPI message-passing.
- The ability to monitor and display performance data, and isolate performance problems to particular intervals ("phases") of program execution.
- An open interface for defining new performance metrics: the Metric Description Language allows the advanced Paradyn user/programmer to define new performance metrics. These metrics can be based on application specific performance data.
- An open interface for adding new run-time visualizers and external analysis: using Paradyn's Visilib, programmers can interface new or existing display routines to Paradyn performance data, and Paradyn's measurement data export capability supports analysis with external tools.

Paradyn differs from many performance tools in that it can decide what performance data to collect while the program is running. When you select some performance metric to be displayed for some part of your program, at that moment Paradyn will insert the necessary data gathering instrumentation into your application program. This method allows you to have direct and dynamic control over the overhead of data collection (so you don't pay for what you don't use).

A tool based on dynamic instrumentation can control instrumentation overhead and data volume while still being able to collect information about the time-varying behavior of long-running complex application programs.

Dynamic instrumentation may seem a bit unusual at first. When you (or the Performance Consultant) are not requesting a particular kind of performance data, it is usually not being collected. This means that there may be intervals of time for which you cannot display data: if you display a time-plot, there will be gaps in the curves. Paradyn tries to keep you informed of these details, so that you can use this information to your advantage.

***Note**: this manual contains color figures with detail which may not be easy to distinguish when printed/viewed in grayscale.*

1.1 Release notes (version 5.1)

Release 5.1 of the Paradyn Parallel Performance tools is provided as both binary and source distributions, along with extensive documentation.

Paradyn is now built using the Dyninst API and Symtab API.

1.2 Release notes (version 5.0)

- Integrated MRNet and Paradyn for improved scalability
- Full support for Linux/x86-64
- Support for parsing stripped binaries on all platforms
- Improved effeciency when instrumenting loops

1.3 Release notes (version 4.2)

- Support for parsing and handling stripped binaries on PowerPC
- Support for instrumenting multi-threaded applications on Linux 2.4/x86 systems with NPTL
- Support for MPI-2 applications using LAM orMPICH
- Support for MPI-1 applications using LAM
- Full support for Linux/IA64
- AIX 5.2 support.
- Improved build/install process.

1.4 Release notes (version 4.1)

Key improvements for Paradyn 4.1 include

- Support for parsing and handling stripped binaries on x86
- Support for instrumenting multi-threaded applications on Linux 2.6/x86 systems with NPTL
- MRNet 1.0 with a test suite, Windows support, and Dynamic Filter Management
- Unified the daemons for multi-threaded and single-threaded applications
- Improved MPI support
- Improved handling of application fork and exec calls
- Performance improvements
- Paradyn front-end uses true Win32 threads on Windows and supports customized fonts
- Paradyn now uses tcl version 8.4.0
- AIX 4.1 support has been dropped

1.5 Release notes (version 4.0)

Key improvements for Paradyn 4.0 include:

- Support for instrumenting multi-threaded applications on AIX and Solaris.
- Support for Performance Consultant searches of multi-threaded applications
- Initial support for PAPI library which can aid in finding memory bottlenecks
- Partial support for new PMAPI interface
- Support for handling application fork and exec calls
- Paradyn and KernInst now use same visualizations
- Enhanced histogram save functionality
- Improved support for using ssh for startup of monitored mpi applications
- Paradyn now uses tcl version 8.3.4
- The Paradyn front-end is preemptively multithreaded, enabling it to run on MP workstations.
- The Paradyn runtime library and DynInst runtime libraries are fully decoupled

1.6 Release notes (version 3.3)

Key improvements for Paradyn 3.3 include:

- can instrument many more application functions on the x86 and SPARC platforms
- now handle applications with multiple names for the same function
- separate window to display application stderr/stdout output
- ability to export data from visualizations for off-line analysis
- optimization of instrumentation based on sharing of common subexpressions
- handle MPI applications that create communicators and tags at a rapid rate
- CPU time can be based on hardware counters on AIX, if they're availables
- sampling of wall time has been made much more efficient for AIX
- 64 bit data path now from data collection to main Paradyn process
- scalability improvements when aggregating sampling data

Platform changes and notes for Paradyn 3.3:

- glibc versions 2.1 and 2.2 are supported on Linux.
- MPI message passing is supported for MPICH 1.2.2 with the ch_p4 device.
- support for instrumenting multithreaded applications is deferred until the next minor release.
- Windows 2000 is now supported.
- IRIX (MIPS), Solaris (x86) and Tru64 (Alpha) are no longer supported by Paradyn. The last release for these platforms was Release 3.2

- Paradyn 3.3 is the last release for Linux 2.2, Solaris 7, and Windows NT 4.0.

1.7 Release notes (version 3.2)

Key improvements for Paradyn 3.2 include:

- support for high-resolution wall timers on Linux based on the TSC register
- support for high-resolution CPU timers on Linux when used with a kernel built with the **hrtime** patch. (See the ***Paradyn Installation Guide*** for details about obtaining and installing this patch.)
- support for shared objects on AIX
- support for high-resolution wall timers on Irix
- improved predicted and observed cost estimates on Linux and Solaris
- improved process control and Fortran support on Irix
- function relocation on x86: functions with tight instrumention points are now relocated and expanded to avoid trap-based instrumentation
- improved x86 function parsing
- many performance improvements and bug fixes

Platform changes for Paradyn 3.2:

- The Paradyn daemon is now supported for sequential programs on AIX 4.3
- PVM is no longer supported
- Paradyn 3.2 is the last release for Solaris 2.6

1.8 Release notes (version 3.1)

Key improvements for Paradyn 3.1 include:

- preliminary support for applications running on Tru64 Unix (Alpha): “beta” versions of the Paradyn daemon and run-time library are available on request
- more reliable bootstrapping (for both explicit process creation and attaching to existing processes) and handling of processes as they fork(), exec() and exit()
- support for x86 applications which have their own signal handlers (which would otherwise conflict with Paradyn’s use of signals for instrumentation and control on these platforms)
- more flexible static and dynamic instrumentation heap organization
- cleaner source organization and easier, more configurable build
- many performance improvements, bug-fixes and software revisions.

1.9 Release notes (version 3.0)

New features for Paradyn 3.0 include:

- support for Irix 6.5 (MIPS; N64 & N32 ABIs; Origin MPI), Linux 2.2 (x86) and newer versions of Solaris
- support for applications using MPICH 1.2.0 on Linux and Solaris platforms
- support for Fortran applications on Irix and WindowsNT
- support for multithreaded applications and per-thread metrics (currently only Solaris threads on SPARC/Solaris, through a separate Paradyn daemon and run-time library: see the separate ***Instrumentation of Multithreaded Programs*** document.)
- dynamic instrumentation and notification of dynamic function calls
- an extensive suite of inclusive metrics (i.e., metrics which include called functions)
- program callgraph display and alternative callgraph-based Code hierarchy search based on retroactive ("catchup") instrumentation execution for currently-executing (on-stack) functions and cheaper inclusive metrics
- reorganization of processes and threads under the Machine resource hierarchy
- execution measurement data export capability to support analysis with external tools
- multiple inferior instrumentation heaps, localized to allow use of atomic single-instruction instrumentation points on SPARC and MIPS.
- system-call interruption and restart (Solaris)
- instrumentation trigger guards available on SPARC, MIPS, RS6000 and x86 platforms
- support for remote/wide-area application monitoring
- shared-memory sampling now standard (i.e., added for WindowsNT and AIX)
- dynamic loading of Paradyn runtime library (*libdyninstRT*) now standard on all platforms (except AIX)
- runtime histogram (*rthist*) visi ported to Tk (like most other Paradyn visis and the main GUI)
- Paradyn GUI and visis ported to WindowsNT
- support for Microsoft CodeView debug format (WindowsNT)
- many performance improvements, bug-fixes and software re-engineering.

Further implementation details behind these features (and more) are available in the Paradyn ***Developer's Guide***.

Paradyn releases attempt to make capabilities available as early as possible on a wide variety of platforms, however, there are some limitations in the current version:

- Fortran and C++ application support is being improved on a number of platforms.
- CPU timers are being derived from virtual timers for more efficient management of large groups of metric timers (such as those used for timing functions which are disabled and then re-enabled during message-passing communication operations based on spin-waiting).
- Handling of collections of processes (on the same processor by a single Paradyn daemon) and propagation of associated metrics to fork&exec'd processes are being improved, along with the management of metrics defined on dynamic sets of processes. Support for applications consist-

ing of multiple executables (and distinct callgraphs/resources) is being improved.

- The automated search executed by the Performance Consultant is being streamlined/optimized to avoid redundant instrumentation of nodes, and the callgraph-based version refined to re-evaluate inclusively-exigent nodes with exclusive metrics to verify whether internal (non-leaf) graph nodes are themselves exigent. The search from all such exclusively-exigent nodes will then progress to consider other resource hierarchies.
- AIX application programs that are to be monitored using Paradyn need to be re-linked with explicit code block markers and Paradyn's run-time instrumentation library. This link step is necessary because Paradyn isn't yet able to dynamically load its instrumentation library under AIX, and the peculiar format of libraries makes it difficult to distinguish user and library modules. Details of this link step are described in Section 2.3. Instrumentation of dynamically-linked libraries is not supported on AIX.
- The standard version of Paradyn cannot safely handle some threaded applications or applications that share code space as it currently does not know about threads. If you use a non-preemptive thread package, Paradyn will still work; performance data can be attributed to the Unix processes, but cannot be broken-down by thread. If you use any multiprocessing or preemptive threading package, Paradyn's instrumentation is likely to misbehave (i.e., we make no guarantees on what will happen). A separate Paradyn daemon and run-time library supporting applications based on Solaris threads on SPARC is available: see the separate ***Instrumentation of Multithreaded Programs*** document for further information.
- Paradyn currently uses 32-bit counters as the basis for some of its instrumentation. For very frequent events, such as those triggered by hardware counters (such as instruction counters or memory reference counters), these 32-bit counters will overflow. Future releases will allow larger counters.
- Instrumentation and monitoring of 64-bit applications is not supported. 64-bit SPARC/Solaris support is under development.
- Instrumentation metrics for I/O are based on the Unix `read()` and `write()` system calls. If you use read or write for socket operations, these will appear as I/O. If you use other system calls that do file I/O, these will not be accounted for.

Most (if not all) of these restrictions will be relaxed in an intermediate release or the next major release of Paradyn.

1.10 Supported hardware and software platforms

The Paradyn user interface can run and Paradyn can monitor application programs on any of the types of workstations and parallel computers listed in Figure 1. Paradyn can also monitor application program running on heterogeneous combinations of these platforms.

Note the following qualifiers:

- MPI programs can only be run under the non-threaded POE environment on SP clusters, and under MPICH 1.2.x on Linux and Solaris platforms. Additional support is in development.

System Identifier	Description
sparc-sun-solaris2.8	Sun Solaris operating system version 8 on SPARC processors.
sparc-sun-solaris2.9	Sun Solaris operating system version 9 on SPARC processors.
i386-unknown-linux2.4	Linux operating system with kernel version 2.4 on x86 processors.
i386-unknown-nt4.0	Microsoft Windows NT operating system version 4.0 on x86 processors. This should work, but we no longer test on this platform.
i386-unknown-w2k	Microsoft Windows 2000 operating system on x86 processors.
i386-unknown-winxp	Microsoft Windows XP operating system version on x86 processors.
rs6000-ibm-aix5.1	IBM AIX operating system version 5.1 running on RS6000 processors.
rs6000-ibm-aix4.3[a]	IBM AIX operating system version 4.3.3 on RS6000 processors. This should work, but we no longer test on this platform.

Figure 1: Platforms on which Paradyn can monitor application processes

a. Power3-based AIX systems will only function properly if APAR IY03550 has been applied. This is detailed in the AIX installation notes, below.

1.11 Other documentation: Manuals

In addition to this ***User's Guide***, the following documentation is available for Paradyn:

Installation Guide

The Installation Guide describes how to obtain Paradyn via anonymous ftp and install it on your system(s). It also describes the minimum operating system and system software version numbers needed for compatibility with this release of Paradyn.

Tutorial

The tutorial provides a step-by-step example of the use of Paradyn. It walks you through the main features of starting a program with Paradyn, displaying performance visualizations, and using the Performance Consultant. The tutorial is intended to show you many of the common and most useful features, but is not a complete description of Paradyn's features. This manual (the ***User's Guide***) contains the complete description of Paradyn.

VisiLib Programmer's Guide

Visilib is the standard API interface for external processes that want to collect performance data from Paradyn. Paradyn performance visualizations (Time Histogram, Bar Chart, Table and 3D Terrain) execute as separate processes, using Visilib as their interface to Paradyn. This document describes how to use Visilib in order to develop custom data visualizations.

MDL Programmer's Guide

MDL programming hints and examples for those wishing to write their own metrics.

Developer's Guide

This is intended for those who wish to understand the Paradyn source code—whether to just to browse it or to actually make changes with the intent of rebuilding Paradyn from scratch.

Instrumentation of Multithreaded Programs

Describes implementation of instrumentation for threaded programs, with current usage and status information for using Paradyn with applications on Solaris, AIX, and Linux 2.6.

1.12 Other documentation: Technical papers

Following is a bibliography of currently available papers on the technology contained in or related to Paradyn. These papers and others may be obtained from the Paradyn Project Web home page.

1. "The Paradyn Parallel Performance Measurement Tools", Barton P. Miller, Mark D. Callaghan, Jonathan M. Cargille, Jeffrey K. Hollingsworth, R. Bruce Irvin, Karen L. Karavanic, Krishna Kunchithapadam, and Tia Newhall. *IEEE Computer* **28**, 11, (November 1995). Special issue on Parallel and Distributed Processing Tools.
2. "An Adaptive Cost Model for Parallel Program Instrumentation" Jeffrey K. Hollingsworth and Barton P. Miller. *EuroPar'96 Conference*, Lyon, France, August 1996. Appears as *LNCS 1123*, Vol.I, pp. 88-97, Springer-Verlag.
3. "Dynamic Program Instrumentation for Scalable Performance Tools", Jeffrey K. Hollingsworth, Barton P. Miller, and Jon Cargille. *Scalable High Performance Computing Conf.*, Knoxville, May 1994.
4. "Dynamic Control of Performance Monitoring on Large Scale Parallel Systems", Jeffrey K. Hollingsworth and Barton P. Miller. *International Conference on Supercomputing*, Tokyo, July 19-23, 1993.
5. "The Paradyn Parallel Performance Tools and PVM", Barton P. Miller, Jeffrey K. Hollingsworth, and Mark D. Callaghan. **Environments and Tools for Parallel Scientific Computing**, J. J. Dongarra and B. Tourancheau, eds., SIAM Press, 1994.
6. "Mapping Performance Data for High-Level and Data Views of Parallel Program Performance", R. Bruce Irvin and Barton P. Miller. *International Conf. on Supercomputing*, Philadelphia, May 1996.
7. "A Performance Tool for High-Level Parallel Programming Languages", R. Bruce Irvin and Barton P. Miller. **Programming Environments for Massively Parallel Distributed Systems**, K. M. Decker and R. M. Rehmann editors, Birkhauser Verlag, pp. 299-314, 1994.
8. "Optimizing Array Distributions in Data-Parallel Programs", Krishna Kunchithapadam and Barton P. Miller. *7th Workshop on Languages and Compilers for Parallel Computing*, Ithaca, NY. August 1994.
9. "Integrating a Debugger and Performance Tool for Steering", Krishna Kunchithapadam and Barton P. Miller. *Workshop on Debugging and Performance Tuning for Parallel Computing Systems*. Cape Cod, Massachusetts, USA, October 1994.
10. "What to Draw? When to Draw? An Essay on Parallel Program Visualization", Barton P. Miller. *Journal of Parallel and Distributed Computing* **18**, 2 (June 1993).
11. "Binary Wrapping: A Technique for Instrumenting Object Code", Jon Cargille and Barton P. Miller. *SIGPLAN Notices* **27**, 6 (June 1992).

12. "Finding Bottlenecks in Large-scale Parallel Programs", Jeffrey K. Hollingsworth, August 1994. University of Wisconsin-Madison Computer Sciences Department Tech. Report #1243 (Ph.D. Thesis).

13. "Performance Measurement Tools for High-Level Parallel Programming Languages", R. Bruce Irvin, October 1995. University of Wisconsin-Madison Computer Science Department Technical Report #1292 (Ph.D. Thesis).

14. "MDL: A Language and Compiler for Dynamic Program Instrumentation", Jeffrey K. Hollingsworth, Barton P. Miller, Marcelo J. R. Gonçalves, Oscar Naìm, Zhichen Xu and Ling Zheng. *PACT'97*, San Francisco, California, USA, November, 1997.

15. "A Callgraph-based Search Strategy for Automated Performance Diagnosis," Harold W. Cain, Barton P. Miller and Brian J. N. Wylie. *EuroPar'2000*, München, Germany, August 2000.

1.13 Contacting the Paradyn developers

There are various ways to get in touch with us. We are glad to answer questions and appreciate feedback.

e-mail: `paradyn@cs.wisc.edu`

This is our project e-mail address. Use this address for technical questions or requests.

Web: `http://www.cs.wisc.edu/~paradyn`

This is our home page. From this page, you can find out how to get a binary or source version of Paradyn. You can also get updates and news on the current release of Paradyn.

FTP: `ftp://ftp.cs.wisc.edu/pub/paradyn/`

This is our ftp site. In the "paradyn" directory, you will find subdirectories containing the binary and source versions of the Paradyn release. Make sure to look at the README files!

FAX: +1-608-262-9777

Postal:

```
Paradyn Project
c/o Prof. Barton P. Miller
Computer Sciences Department
University of Wisconsin
1210 W. Dayton Street
Madison, WI  53706-1685
U.S.A.
```

2 RUNNING PARADYN

In this section, we describe the steps that you should follow to run Paradyn. First we give you an overview of the major steps and then we explain each one in detail. For this section, we are assuming that you have already installed Paradyn as documented in the ***Installation Guide***.

2.1 Overview of major steps

To run Paradyn, follow the steps:

1. *Set up Paradyn and daemons (Section 2.2):* You need to specify the location of the Paradyn executable and configuration files and some external libraries. You may also wish to override the default fonts used in the Paradyn GUI.
2. *Run Paradyn (Section 2.4):* Paradyn has several options that you may use during execution, such as adding a new process to your application. These options may be specified directly on the command line or in a Paradyn configuration file for the application.

Sections 2.2 through 2.4 explain these steps in more detail.

2.2 Setting up Paradyn and the Paradyn daemons

Paradyn has two main parts: the Paradyn front-end and user interface ("paradyn") and the Paradyn daemons ("paradynd"), which are the agents that run on each remote host where your application program is running. Paradyn contains the user interface that allows you to display performance visualizations, use the Performance Consultant to find bottlenecks, start or stop your application, and monitor the status of your application. The Paradyn daemons operate under the control of Paradyn to monitor and instrument the application processes. Paradyn also uses configuration files to specify details of Paradyn configuration, instrumentation and application programs. You must have Tcl and Tk library files installed to be able to use the Paradyn front-end. Also, a special RPC package is required to use Paradyn on Windows systems.

For the details of installing Paradyn, its daemons, Tcl/Tk and other external software, refer to the ***Paradyn Installation Guide***. This section details settings for environment variables required for Paradyn's operation, and describes how to override the default fonts used by the Paradyn GUI.

2.2.1 Paradyn environment variables

After you have installed Paradyn, you need to specify the location of Paradyn's executable and configuration files. The files needed to run Paradyn are listed in Figure 2, along with explanations of their use. The environment variables that are needed or helpful when running Paradyn are listed in Figure 3, along with a description of their use.

File	Use
`paradyn` `paradyn.exe`	The executable that starts a Paradyn session and provides the main user interface. There are versions for each supported platform and an appropriate version should be placed in a location that will be found by your shell's search path (or you can specify the full path name to run it).
`paradynd` `paradynd.exe`	The executable for a Paradyn daemon. Versions exist for each of the supported target application environments, and an appropriate version should be placed in a location that will be found by your shell's search path (or you can specify the full path name to run it).
`paradyn.rc`	Contains crucial information, such as metric and daemon definitions. The following steps are used to try to find this file (in the order listed): 1. Look for the file `paradyn.rc` in the directory specified by the environment variable "`PARADYN_ROOT`" (i.e., `$PARADYN_ROOT/paradyn.rc`). 2. Look in your current working directory for the file `paradyn.rc`.
`.paradynrc`	In addition to `paradyn.rc`, Paradyn will also look in your account's home directory for a file named `.paradynrc` (note the slightly different form). Should it exist, this file is processed after, and in addition to, `paradyn.rc`.

Figure 2: Files needed to run Paradyn

2.2.2 Overriding Paradyn's default fonts

Paradyn's graphical user interface (GUI) uses fonts that may not be available on all systems or be visually appealing to all users, so Paradyn allows users to specify alternative fonts to use instead of its defaults. Alternative fonts are specified to Paradyn using Tk's options database. (Paradyn uses Tk to implement its GUI.) Tk initializes its options database differently depending on the type of

Environment Variable	Use
PARADYN_ROOT	Specifies the location of the `paradyn.rc` configuration file. In source code distributions of Paradyn, it is also used to locate the root of the Paradyn code tree. (Not required if you are running Paradyn from your current working directory or from your home directory.)
PARADYN_LIB	Used to specify the Paradyn run-time instrumentation shared object file (`libparadynRT.so.1` or `libparadynRT.dll`). The full path name of this file must be specified. E.g., `setenv PARADYN_LIB /usr/home/me/lib/libparadynRT.so.1`
DYNINSTAPI_RT_LIB	Used to specify the Dyninst run-time instrumentation shared object file (`libdyninstAPI_RT.so.1` or `libdyninstAPI_RT.dll`). This file is required for use with Paradyn. The full path name of this file must be specified. E.g., `setenv DYNINSTAPI_RT_LIB /usr/home/me/lib/libdyninstAPI_RT.so.1`
PARADYN_LIB_MT	This environment variable is needed only if paradyndMT, the daemon for monitoring multi-threaded applications, is used. Used to specify the Paradyn run-time instrumentation shared object file for multi-threaded applications (`libparadynMT.so.1`). The full path name of this file must be specified. E.g., `setenv PARADYN_LIB_MT /usr/home/me/lib/libparadynMT.so.1`
LD_LIBRARY_PATH (UNIX only)	Update this variable so that the directory that contains the libpdthread.so file is specified, which is used by the front-end. On Linux, update so that the directory that contains the libdwarf.so file (which can be retrieved off of the release web page) is specified. The libdwarf.so file is used by the Paradyn daemon. E.g., `setenv LD_LIBRARY_PATH /usr/home/me/lib:/usr/home/libdwarf`
AIXTHREAD_SCOPE (AIX ONLY)	This is needed only on AIX if paradyndMT is being used. This environment variable should be set to 'S' on AIX.
TCL_LIBRARY TK_LIBRARY	These environment variables specify the location of the Tcl and Tk command files needed to implement the basic Tcl/Tk object types. If you have been using a current installed version of Tcl/Tk, you probably already have these correctly set. If not, see the instructions in the ***Paradyn Installation Guide*** for information on setting them.
LIBPATH (AIX ONLY)	This is needed only on AIX. This environment variable specifies the location of the file `libDyninstText.a`.

Figure 3: Environment variables used when running Paradyn

Environment Variable	Use
PARADYN_RSH	This environment variable will specify the program to use when starting daemons on remote hosts. The pcl file variable, remoteShell is an alternative way to specify the shell used for starting remote daemons. For example, one might set it to 'ssh'.
PD_TIMER_SHOW_INFO	This environment variable is not required to be defined. If this environment variable is set, Paradyn will display the time querying method it will be using in the termWin visi. For example, if Paradyn is querying the cpu time with a method that uses a hardware counter, "Chosen cpu timer level: 1" will display. If Paradyn uses a software based time querying method for cpu time, "Chosen cpu timer level: 2" will appear.

Figure 3: Environment variables used when running Paradyn

system. On UNIX systems, Tk initializes its options database from the X resource database, so alternative Paradyn font specifications should be added to whatever file the system's X installation uses to specify X resources (e.g., `$HOME/.Xresources`). On Microsoft Windows systems, Tk adds the contents of the `$HOME/.Xdefaults` to its options database, so specifications for alternative Paradyn fonts should be added to that file.

To specify an alternative font, add a line of the form:

resource_path.Font: *font_specification*

to the appropriate Tk options database configuration file. Like all Tk-based applications, each graphical resource (e.g., button, menu, or label) can be identified with a path. Similar to resource paths in the X Window System, each component in the path can be either the name of a Tk GUI component class such as Menu, Button, and Label, or it can be the name of a particular instance of a Tk GUI component class. Paths are read left-to-right, with the outermost GUI component specified first in the path. Components in a path are separated by a period '.' or an asterisk '*'. If two components are separated by a period ("x.y"), the first component ("x") is the immediate parent of the second component ("y"). Because it is tedious to specify all GUI components in each resource path, the '*' indicates zero or more components are not specified. For example, the resource path `Paradyn*status.Button` names all buttons in all windows of the Paradyn class, but only if they are immediately enclosed by some other component whose instance name is `status`. The subtleties of resource paths are outside the scope of this document; because resource paths are specified the same whether on UNIX or Microsoft Windows systems, we refer the reader to the X Window System documentation for complete details.

The font specification can be given in Tk's shorthand notation instead of the X Window System's font naming scheme. For example, a font specification can be given as "`times 10 italic`" instead of "`-*-times-medium-i-*-*-10-*-*-*-*-*-*-*`". For details on font naming, we refer the reader to the Tk documentation for details.

In the Paradyn GUI, all top-level windows have the class `Paradyn` so that alternative fonts can

be specified easily that apply to all windows in the Paradyn GUI. Although we do not list the class and instance names of all components of the Paradyn GUI here, alternative font specifications similar to the following should suffice for most users:

Paradyn*Font:	helvetica 10 roman
Paradyn*Button.Font:	helvetica 10 bold roman
Paradyn*Menu.Font:	helvetica 10 bold roman
Paradyn*Menubutton.Font:	helvetica 10 bold roman
Paradyn*Label.Font:	helvetica 10 bold roman
Paradyn*Entry.Font:	helvetica 10 roman
Paradyn*Message.Font:	helvetica 10 roman
Paradyn*Text.Font:	helvetica 10 roman

2.3 Preparing your application program

On most platforms, Paradyn is able to instrument unmodified executable files (`a.out` or `*.exe`). However, some platforms require preparation of executables, and Paradyn will benefit from the inclusion of debug information on all platforms. This section details the application preparation required or recommended for use with Paradyn. Additional platform-specific build and execution details are documented in Section 2.6

2.3.1 Generation of debug information (all platforms)

Paradyn will benefit from access to debug information for the application under study, so we recommend that executables be built to contain debug information if possible. For most compilers, this means passing the -g compile flag to generate debugging information. For the Microsoft Visual C++ compiler, use -Z7 or -Zi.

Note that often this does not require disabling any compiler optimizations, and while generating debug information may result in a slightly slower build and larger executable, there are no execution performance implications.

2.4 Running Paradyn

At this point, your should be ready to run your application program with Paradyn. You start Paradyn by entering the following command at a command prompt[1]

```
% paradyn
```

Several optional command line arguments can be used when invoking Paradyn:

- `-f <pcl-configuration-filename>`
  specifies a file from where Paradyn can read configuration commands (see Section 12);
- `-default_host <host name>`

1. A command prompt is available under Windows from the “Command Prompt” item under the Programs submenu of the Start menu.

specifies the default host where Paradyn should start an application when no host name is given. (If the `-default_host` option is not used, the default host is the local host.)

- `-x <connect-filename>`
  specifies a file to which Paradyn daemon start-up information will be written, which may be used by external programs to explicitly start Paradyn daemons on different hosts which will connect to this Paradyn front-end. (This file is created if it doesn't already exist.)
- Paradyn should start running and display the Paradyn Main Control Window, shown in Figure 4. This window has five menus, **File**, **Setup**, **Phase, Visi,** and **Help**. These menus allow you to:

1. **File**: In this menu there is an option to get information on how to start up a daemon (**Daemon start-up info**) and an option to exit the program (**Exit Paradyn**.)
2. **Setup**: This menu has selections to allow you to describe a new application program to run from scratch (**Define a Process**, described below) or attach to an already-running application process (**Attach to a Process**, below). In addition, you can bring up windows which allow you to start the Performance Consultant (**Performance Consultant**, described in Section 9), change Paradyn's tunable constants (**Tunable Constants Control**, described in Section 4), bring up the call graph of the program (**Call Graph**), and view the WhereAxis (**Where Axis**).
3. **Phase**: start and define a new local phase for visualizations and analysis (see Section 8).
4. **Visi**: start visualizations of your application performance (see Section 7).
5. **Help**: get additional information about Paradyn.

Additionally, there are four buttons in this window: **RUN**, **PAUSE**, **EXPORT** and **EXIT**. **RUN** and **PAUSE** are disabled when there is no application currently defined. These two buttons allow you to run or stop execution of your application as you wish. **EXPORT** will open a dialog offering to save the data from current measurements to files for off-line analysis (see Section 11). Finally, **EXIT** will exit Paradyn, terminating the application program if necessary, and end the session.

The Paradyn Main Control Window can contain several status lines. Each status line represents information about some part of Paradyn or your application. In the initial window, there is a status line labeled "UIM status". This line shows the current state of Paradyn's User Interface Manager ("ready" in this case).

Figure 4: Starting Paradyn

2.5 Running applications with Paradyn

There are two ways to give Paradyn an application program to monitor: defining a new process to start, and attaching to an already-running process. These two methods are described below.

2.5.1 Defining a new process

One way to measure a program with Paradyn is to select the option **Define A Process** from the **SetUp** menu. A new window appears, as shown in Figure 5.

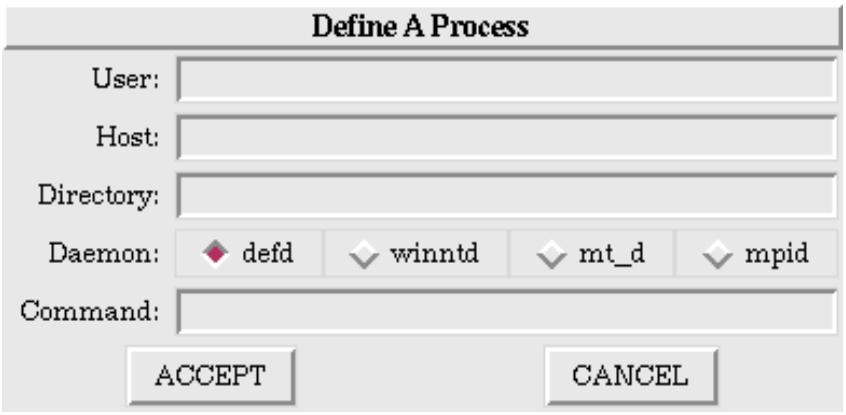


Figure 5: Defining a new application process

From this window, you can specify the following parameters:

1. **User**: This is your login name on the host on which Paradyn will run your application process. If you leave this field blank, the login will default to your current login name.
2. **Host**: This is the name of the host on which Paradyn will run your application. If you leave this field blank, it will default to the host specified with the `-default_host` command line option to paradyn, or to the current host (the one on which the Paradyn front-end is running), if the option `-default_host` is not used.
3. **Directory**: Paradyn runs paradynd and your application as follows. First, it performs a remote login operation using the "User" and "Host" fields specified above. The current directory (on the remote machine) at this point is the root directory—not usually where your application program resides. The "directory" entry box allows you to specify a directory to change to before executing the command specified in the "Command" entry box. Note that the path-name given is interpreted on the field named in the "Host" field. For UNIX hosts, the allowed syntax is familiar: the path specified may start with a slash ("/") (specifying an absolute path name, starting from the file system root directory), or it may start with a tilde ("~") followed by a user name (specifying a path name rooted at the specified user's home directory). A tilde not followed by a user name is the same as a tilde followed by the current user name. For Windows hosts, the path may start with a drive letter, for example "d:\myprograms\bubba". Both forward and backslashes are accepted in Windows paths.
4. **Command**: The command that will start this instance of your application program. If the

Directory entry has been filled in, the command is executed with the current directory set to the specified path. If the Directory entry is left blank, then the command will be executed with the current directory set to the home directory of the specified user. Under Windows, any backslashes in the command must be escaped with another backslash, for example "..\\bubba example5".

5. **Daemon**: This option allows you to specify which Paradyn daemon to run. For most uses, the default daemon ("defd") is appropriate. For Windows applications, use the "winntd" daemon. For MPI applications, select either "mpid". For multi-threaded applications (only on Solaris and AIX), use the "mt_d" daemon. If you specify additional daemons in the Paradyn configuration file, they will appear here.

Once you have made your selections, click on **Accept** and Paradyn will start the application program and initialize it. When the status of the Paradyn window is like that in Figure 6, the program is ready to run and be measured.

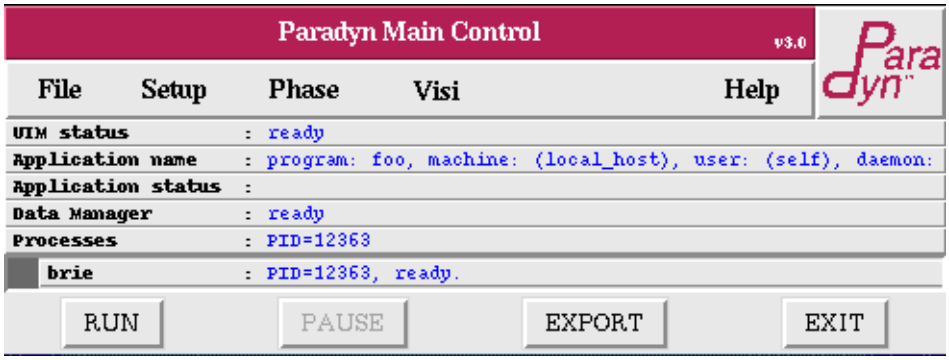


Figure 6: Paradyn ready to run the application

The window in Figure 6 shows several new status lines with the following information:

1. *Application name*: This is the name of the application program (`foo`), the host machine where it has been started (if remote), the user identifier which it is running as (if different), and the type of daemon which is monitoring it (`defd`).
2. *Application status*: This is the overall application status (either `PAUSED` or `RUNNING`).
3. *Data Manager*: This is the status of Paradyn's Data Manager.
4. *Processes*: This is the process identifier of the controlling process in your application.
5. *brie*: There is one status line on each host or node on which you are running your application; here there is the status line for host "brie". It shows the current status of your application process on this host/node.

Notice that since you have defined a new process the RUN button is enabled and you are ready to run and measure your program!

The information in the "Define a Process" window can be stored in a Paradyn Configuration

Language (PCL) file. In this file, the user can specify information such as: user application, new visualizations to be added to the system, new metrics, and additional paradyn daemons. The complete details of the Paradyn Configuration Language are given in Section 12.

As a simple example, if we want to run an application called "bubba", with an executable file named "bubba_pd" indicating special Paradyn support, a file called "bubba.pcl" might contain:

```
process bubba {
      dir "/p/paradyn/applications/sequential/bubba";
      command "bubba_pd example.dat";
      daemon defd;
}
```

and the command to automatically start Paradyn with this application would be like this:

```
% paradyn -f bubba.pcl
```

This command tells Paradyn to run the application "bubba" in the directory specified by "dir" using the command line specified by "command" with the Paradyn daemon specified by "daemon" (defd or default daemon in this case).

2.5.2 Attaching to a process

Sometimes, defining a new process from Paradyn as shown in the previous sub-section is not convenient. The main limitation of defining a new process is that a new process is launched every time you run Paradyn (and killed every time you exit Paradyn). Many programs you may wish to measure are not amenable to starting up and shutting down every time you wish to measure them. Typically these are server-type programs, which are meant to run for an indefinite amount of time. In such cases, it is more convenient to attach to an already-running program when you wish to measure it with Paradyn, and to detach from it when you exit Paradyn.

Paradyn currently does not offer the option to detach from the application leaving it running when you exit; on exit, Paradyn kills the application it is monitoring and all its associated processes. This limitation will be removed in a future release.

To attach to a running process, choose **Attach to a Process** from the **Setup** menu of the Paradyn main window. A dialog box (Figure 7) will appear.

The **User**, **Host**, and **Daemon** items have the same meaning as in Section 2.5.1. The most important box is **Pid**, where you specify the process identifier of the process (on the **Host** machine) you wish to attach to. The **Executable file** item lets you specify a full pathname to the executable file corresponding to the process id. The Paradyn Daemon needs to find the executable file on disk in order to extract symbols (procedures, modules) that will go in the **Code** portion of the Paradyn **Where Axis**. Obtaining symbols from the executable file is also done when defining a new process (Section 2.5.1). However, it can be burdensome to enter the full path name of a process that you want to attach to; it is possible that you might not even know the disk directory from which it was launched. Therefore, if you leave the **Executable file** item blank, the Paradyn Daemon will make an effort to locate its value automatically. (It obtains the program name by examining the process' first argument, `argv[0]`. It then looks in several directories for this program name; it searches the pro-

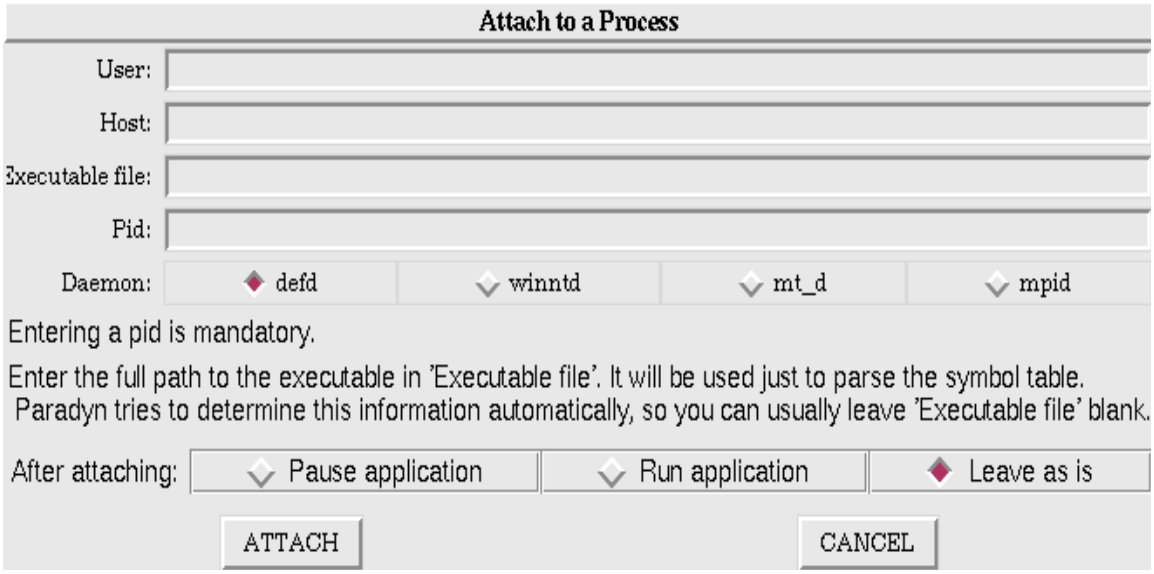


Figure 7: Specifying a process to attach to.

cess' current directory and all items in its **PATH** environment variable. For those interested, further technical details on how attach is performed can be found in the separate ***Paradyn Developer's Guide***.) If Paradyn reports that it cannot locate the executable file, you will have to enter the full path name in the **Executable file** field.

The Paradyn daemon can attach to a process, whether it is currently running or stopped. After it has attached, you may wish to have the daemon automatically pause or run the application. To do this, choose either **Pause application** or **Run application** items from the dialog box. The default is **Leave as is**, which detects whether the program was running or stopped at the time of attach. Note that the process is necessarily paused for a short time while the Paradyn daemon initializes it (parses its symbol table, parses any shared libraries it has been linked with, etc.)

When you have entered the desired parameters, click on **ATTACH** to perform the attach operation. When ready, the Paradyn main window should look like Figure 8.

2.6 Architectural issues

Certain platforms require slight modifications to the procedures discussed above. In this subsection, we describe each of them in turn.

2.6.1 Common Platforms

These notes apply for Solaris, Linux (x86/ia64), and AIX (RS/6000). Variations for Windows and MPI programs follow.

On these platforms we support instrumenting shared objects (dynamically-linked libraries). Dynamic executables are executables that are linked with shared object files, and are the default

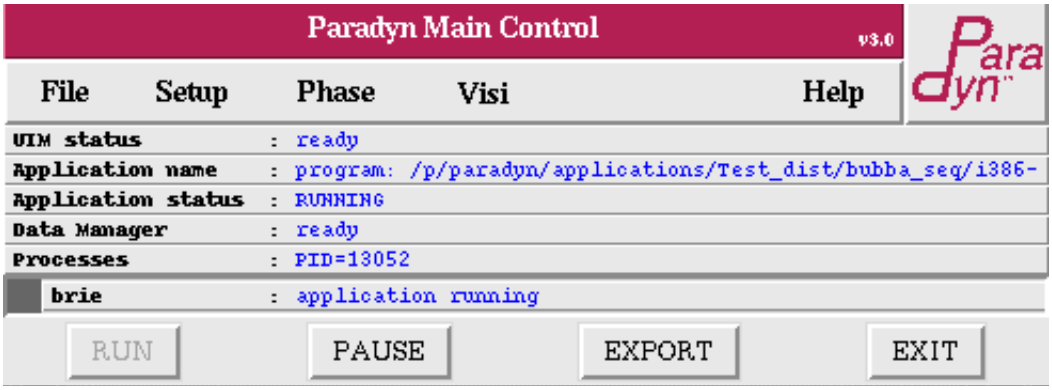


Figure 8: Attach completed and application execution continuing.

output generated by the link-editor, therefore no special flags are needed to create dynamic executables. Paradyn's run-time instrumentation library is a shared object (`libparadynRT.so.1`) which is dynamically loaded at run-time, and does not need to be linked with the executable.

Shared objects will show up on the Paradyn Where Axis and performance data can be collected for functions from shared objects. Also, the Performance Consultant will include functions in shared objects in its search for bottlenecks. The MDL **exclude** option can be used to specify shared objects and/or functions from shared objects that should not be included in the Performance Consultant's search. This is discussed in more detail in Section 12.8.

When using the Sun C or Fortran compilers on Solaris, you should also specify the **`-xs`** option together with **`-g`**. The **`-g`** option alone will direct the compiler to place debugging information in the object files (**`.o`** files), but it will not place the debugging information on the executable (**`a.out`**) file. You must use the **`-xs`** option so that the compiler will add the debugging information to the a.out file. The **`-xs`** option is not needed if you are using GNU compilers..

Inter-library calls on AIX appear to be made by a "shadow" function with the same name as the function being called. Paradyn detects these shadow functions and appends "_linkage" to their names. This is simply for clarification purposes.

2.6.2 MPI

To run an MPI application under Paradyn one should follow the steps described in Section 2.5.1. The user should set *Daemon* to *mpid*. The *Command* field should contain the same command line one uses to launch the application without Paradyn(e.g., "`mpirun -np 2 hello`"). Note that the *Processes* field for MPI will identify the type of MPI being used (POE, IRIX or MPICH) and not the processes involved in the job. Platform-specific details are given below.

- **MPICH**: Currently, Paradyn supports MPICH version 1.2.2 with the ch_p4 device. The user should use the "`mpirun`" command to start an application. Unsupported mpirun options include "`-p4pg file`", "`-gdb`" and "`-dbx`". The "`mpirun`" command will be started through a remote shell on the machine specified in the *Host* field of the dialog box. If no name is specified, it will run on the frontend machine.

- **POE MPI on AIX**: the POE job launcher `poe` can be entered in the command field or omitted.

2.6.3 Microsoft Windows

The way Paradyn works in Windows is similar to other platforms, however there are a few small differences.

On Windows the run-time instrumentation library (`libparadynRT.dll`) is loaded dynamically. You must either define `PARADYN_LIB` with a full path to `libparadynRT.dll` or have it in a directory that is listed in your PATH environment variable, so that it can be found by the dynamic linker.

Paradyn needs symbolic debug information, so you must compile your application with debugging information enabled. We currently handle CodeView (also called "Microsoft style") and COFF symbol formats, though we recommend CodeView format since it provides more complete and accurate information than with COFF symbols. The option to enable a CodeView symbol table will depend on the compiler used. For the Microsoft compiler this option is `/Z7 or /Zi`. You must also direct the linker to generate symbolic information in the executable file. The options `/debug` must be passed to the linker. Figure 9 shows a sample Makefile for the Microsoft Visual C++ compiler.

```
CC = cl /Z7
OBJECTS = main.obj this.obj that.obj

bubba.exe:  $(OBJECTS)
      link -out:bubba.exe -debug $(OBJECTS)
```

Figure 9: Sample Makefile for Windows.

The files which are needed to run on Windows are `paradynd.exe` (the paradyn daemon), `libparadynRT.dll` (the run-time dynamic instrumentation library), and `oncrpc.dll` (a version of the Sun RPC library for Windows, included with the Paradyn binary release, which is used by Paradynd to communicate with the Paradyn front-end). All of these files should be in directories that are listed on your "path" environment variable.

In order to have a Paradyn daemon started automatically by the Paradyn front-end (as for the other platforms), you need to have a remote shell daemon (**rshd** or **sshd**) running on the Windows machine(s), and you must be able to execute commands on Windows from the Unix machine where the Paradyn front-end is running. If you don't have an **rshd** running on the Windows machine, you must start the Paradyn daemon manually. Either refer to the `-x` command-line option for Paradyn to automatically get this information (Section 2.4) or use the information from the "Daemon start-up info" menu item under the "File" menu in the Paradyn user interface. You must start paradynd giving the exact arguments shown in that dialog but specifying the appropriate "flavor" (which will be `winntd` for a Paradyn daemon and application processes running on Windows): note that for each session the port identifier (and possibly also host machine) arguments will be slightly different, so you can't reuse exactly the same command line for different Paradyn sessions. The command line to start paradynd on Windows will look like:

```
paradynd -zwinnt -l2 -mmyhostmachine.domain.org -p12345
```

Once the Paradyn daemon is started, it connects to the existing Paradyn front-end session, and everything else will work as usual.

Note that currently Paradyn is not expected to work with gcc-compiled application programs under Windows.

2.6.4 Multi-threaded

Paradyn currently supports multithreaded applications on all platforms.

3 MAIN CONTROL WINDOW

In this section we discuss features of the Paradyn main control window (an example is shown in Figure 10). The Paradyn main window is the interface though which a user can access all parts of the Paradyn tool. The main window is divided into three sections; the top section contains a menu bar, the middle section contains a dynamic set of status lines (split into a generic part and a part for per-process status information which is both resizable and scrollable), and the bottom section contains a set of menu buttons. We discuss the details of each of these below.

Figure 10: Paradyn Main Control window

3.1 Main menubar

The menu bar in the Paradyn main control window contains five items; four of these display a sub-menu when selected, and the other opens a dialog, as follows:

3.1.1 File menu

The **File** sub-menu contains two menu items. The option **Daemon start-up info** gives information on how to manually start a daemon from a terminal. When the option **Exit Paradyn** is selected, the Paradyn process and all currently-associated application, daemon and visualization processes exit. The same effect can be achieved by clicking on the **EXIT** button (Section 3.3).

3.1.2 Setup menu

The **Setup** menu contains items to define an application process, to attach to an already-running application process, to create a Performance Consultant window, to bring up the Tunable Constants dialog, and to bring up the Where Axis display. Selecting **Define A Process** displays the **Define A Process** window (this window is shown in Figure 5 in Section 2.5.1). This is a mechanism through which a user can provide information about their application so that Paradyn can start it. A description of how to use the **Define A Process** window is given in Section 2.5.1.

Using **Define A Process** creates (i.e. starts) a new application process, which Paradyn can begin monitoring right away. Sometimes, however, it is more convenient to ask Paradyn to attach to an already-running process (supported since Paradyn release 1.2). This is especially useful for server-type processes such as database servers or file servers, for which re-launching every time you wish to measure with Paradyn would be inconvenient. To attach to an already-running process, select **Attach to a Process** from the **Setup** sub-menu. A description of how to use **Attach to a Process** is given in Section 2.5.2.

The **Performance Consultant** menu item will bring up the Performance Consultant window. This window provides an interface for the user to start automated performance bottleneck searches. The Performance Consultant is described in Section 9.

The **Tunable Constants** menu item will bring up the Tunable Constants dialog, through which the user can set values for any tunable constants defined in Paradyn. Information about the set of tunable constants and how they can be modified is given in Section 4.

The **Where Axis** menu item will bring up the Where Axis display, through which the user makes resource hierarchy selections. Information about the Where Axis is given in Section 5.2.

The **Call Graph** menu item will bring up the Call Graph display, which provides an interactive representation of the callgraph of each executable in the application. Information about the Call Graph is given in Section 5.5.

3.1.3 Phase menu

Phases may be started using the **Phase** menu. There are presently four items under this menu: **Start**, **Start with Perf Consultant**, **Start with Visis**, and **Start with Perf Consultant & Visis**. Each item under this menu will create a new phase; they differ in what additional actions they take. The first item, **Start**, does nothing additional. **Start with Perf Consultant** will have Paradyn's Performance Consultant module (Section 9) commence searching on this phase, as opposed to simply defining the new phase. *Note:* **Start with Visis** *and* **Start with Perf Consultant & Visis** *are not yet implemented.* Complete information about phases is provided in Section 8.

3.1.4 Visi menu/button

Visualization processes can be started by selecting them from the **Start A Visualization** dialog which appears upon pressing the **Visi** button in the main menubar. A complete description of how to start a visualization process is given in Section 7.1, and documentation on the standard visualization modules is given in Section 10.

3.1.5 Help menu

The **Help** menu options offers basic information about Paradyn in separate displays. **General Info** has summary information about Paradyn capabilities and supported platforms, along with pointers to project Web pages and the `paradyn@cs.wisc.edu` maintainers' account for further information or to report problems. **License Info** contains a copy of the license agreement governing use of the Paradyn Parallel Performance Tools. **Release Info** provides information related to the current

Paradyn release (and obtaining other releases). Finally, **Version Info** displays build/release information about the version of Paradyn which is running: it is more detailed than the abbreviated version identifier appearing in the upper-right of the display title, and you may be asked to provide this information when reporting any problems with special versions of Paradyn.

3.2 Status lines

The middle section of the Paradyn main window consists of a dynamic set of status lines which are updated as Paradyn runs and learns about new application processes. Each line displays status information about some part of Paradyn, the application, or the Paradyn daemons monitoring application processes.

The main window in Figure 10 contains status lines that were created after a sequential (single process) application was defined. Some of the status lines contain information about the application program, such as its name (`foo`), the process identifier(s) associated with the application on the host (`PID=19271`) and an indented/offset area with status lines for each host machine or processor node on which the application is running (in this case, only on one host, `beaufort`). There are also lines displaying the status of the UI Manager and Data Manager (`ready`).

The indented/offset area grows additional lines as hosts or nodes join the set which constitutes the application managed by Paradyn. After a certain number of lines is reached, this area no longer grows automatically and a scrollbar appears in the indent area to manage this region of the display. If desired, the window can be vertically resized to display more (or all) of the host/node status lines, or shrunk to display fewer (down to a minimum number which can still be displayed).

3.3 Buttons

There are four buttons at the bottom of the Paradyn main window.

The **RUN** and **PAUSE** buttons allow the user to run or pause execution of the application. When the application is running, the **RUN** button is disabled and the **PAUSE** button enabled. Conversely, when the application is paused the **PAUSE** button is disabled and the **RUN** button enabled. Before an application has been defined, both buttons are disabled.

The **EXPORT** button dumps the application execution data Paradyn currently maintains to files for off-line analysis. This is useful for exporting execution data from Paradyn to other analysis tools. Complete details are in Section 11.

The **EXIT** button, when selected, will exit Paradyn and terminate all associated application and visualization processes.

4 TUNABLE CONSTANTS

4.1 Overview

Users can customize Paradyn's operation through *tunable constants*. Paradyn defines several tunable constants that may be altered by the user, ranging in scope from user-interface window layout issues to tuning the automated search parameters of the Performance Consultant (see Section 9).

Tunable constants are either boolean or floating-point. Paradyn's tunable constants are listed in Sections 4.2 and 4.3.

To change the value of a tunable constant, choose **Tunable Constants Control** from the **Setup** menu of the **Paradyn Main Control** window. This brings up the window shown in Figure 11.

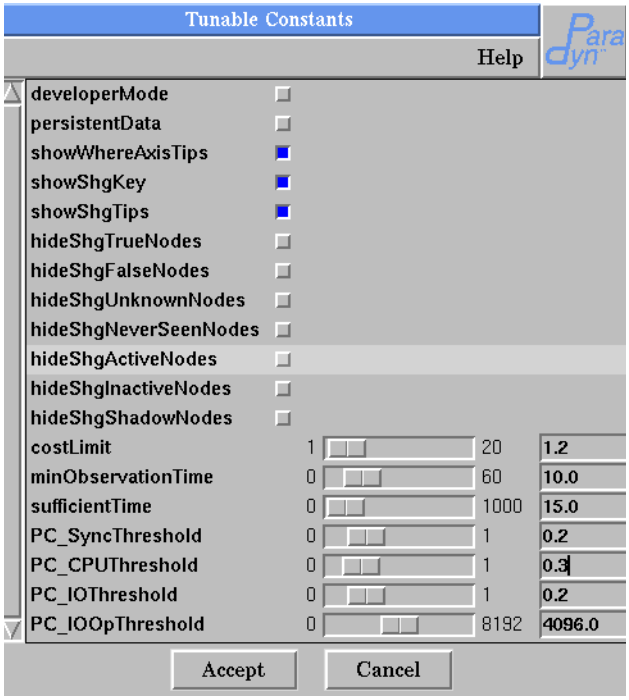


Figure 11: The Tunable Constants Window

Boolean tunable constants (**developerMode, showWhereAxisTips, showShgKey, showShgTips,** and so on through **hideShgShadowNodes** in Figure 11) are shown before floating-point numeric constants. The checkbox to the right of a boolean tunable constant is colored gray if the tunable constant's setting is false, and blue if the tunable constant's setting is true. Floating-point tunable constants with bounds on their acceptable values have a slider widget between the name and the entry field. A new value can be typed into the entry field or click on the slider and "drag" it to the desired value. The minimum and maximum allowable values are displayed on the left and right sides of the slider as a convenience: attempts to set values outside this range are truncated.

Changes made to tunable constant values do not take effect until the window is dismissed by

clicking on **Accept**. Clicking on **Cancel** will dismiss the window without making any changes. Tunable constant settings remain in effect for the duration of this Paradyn session; that is, until you explicitly change the value again through this dialog or quit the Paradyn process.

Each time a new Paradyn session is started, tunable constants are reset to their default values. This can be an inconvenience if the default values of certain tunable constants are not to your liking. The Paradyn Configuration Language (PCL) allows you to create files read by Paradyn on startup. Among many other things, such files can contain tunable constant settings to your liking. See Section 12.6 for particulars on how to set tunable constant values in a PCL file.

Under the Tunable Constants Window **Help** menu is an entry **Show Tunable Descriptions**. Invoking this menu item brings up the **Tunable Descriptions** window, giving a concise description of each tunable constant. An example is shown in Figure 12.

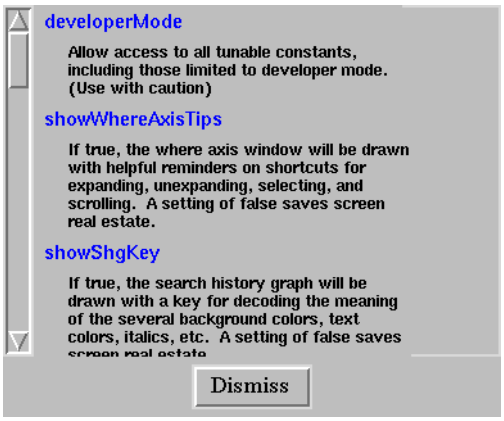


Figure 12: Tunable Constants Descriptions Window

4.2 User Tunable Constants

Each tunable constant is classified as either **User** or **Developer** mode. User tunable constants are intended for everyday use. User tunable constants are listed in Figure 13.

4.3 Developer Tunable Constants

Developer tunable constants are not intended for everyday use. **If you change a developer mode tunable constant, you are presuming a detailed knowledge of the internal workings of Paradyn**. We provide no guarantees on how system behavior changes, nor can we offer support if any developer tunable constant has been altered from its original setting. In addition, developer tunable constants are subject to significant change from release to release. Nevertheless, we realize that some experienced users may benefit by occasional access to these tunable constants.

To access developer tunable constants set the tunable constant **developerMode** to true and click **Accept**: the **Tunable Constants** window will re-present itself containing both the user and developer tunable constants. Setting the tunable constant **developerMode** to false will "hide" the developer

tunable constants once again. Developer tunable constants are listed in Figure 14.

Tunable Name	Description
showWhereAxisTips (bool)	If true, the Where Axis window is drawn with several user-interface tips on how to select and expand where axis items. Setting to false saves screen real estate.
persistentData (bool)	If true, all performance data remains stored internally in Paradyn histograms after data collection has been halted by removing the instrumentation. If false (the default), internal data is deleted at the time the related instrumentation is removed (for example, when a visualization is exited).
costLimit (float)	Maximum allowable perturbation of the application when running the Performance Consultant (Section 9). Paradyn keeps track of an estimate of the extent to which its instrumentation is perturbing the application under execution; this tunable constant allows users to set a maximum upper-bound on such perturbation, as a percentage of execution time.
minObservationTime (float)	Specifies a lower bound on the time (in seconds) before the Performance Consultant will begin using data collected to evaluate hypotheses. This time guards against the effects of transient data values at the start of a phase.
sufficientTime (float)	Specifies the minimum amount of time (in seconds) before the Performance Consultant can conclude that a hypothesis is false.
showShgKey (bool)	If true, the Performance Consultant window includes a key to the meaning of the node and text colors shown.
showShgTips (bool)	If true, the Performance Consultant window includes a key to relevant mouse functions.
hideShgTrueNodes (bool)	If true, the Performance Consultant's Search History Graph (SHG) will not show true nodes.
hideShgFalseNodes (bool)	If true, the SHG will not show false nodes.
hideShgUnknownNodes (bool)	If true, the SHG will not show nodes which haven't been determined true or false yet.

Figure 13: User-level Tunable Constants

hideShgNeverSeenNodes (bool)	If true, the SHG will not show nodes which it has not begun to evaluate yet.
hideShgActiveNodes (bool)	If true, the SHG will not show nodes which are active (instrumented).
hideShgInactiveNodes (bool)	If true, the SHG will not show nodes which are inactive (un-instrumented).
hideShgShadowNodes (bool)	If true, the SHG will not show shadow nodes.
useLoops (bool)	If true, the SHG will contain nested loops.
PC_SyncThreshold (float)	Percentage Performance Consultant uses as threshold for all synchronization hypotheses (such as *ExcessiveSyncWaitingTime*). For example, selecting 20% here will cause any synchronization-related hypothesis-focus pair testing above 0.20 to conclude "true."
PC_CPUThreshold (float)	Percentage Performance Consultant uses as threshold for determining CPU bottlenecks (*CPUbound*).
PC_IOThreshold (float)	Percentage Performance Consultant uses as threshold for determining I/O blocking time bottlenecks (*ExcessiveIOBlockingTime*).
PC_IOOpThreshold (float)	Number of bytes Performance Consultant uses as threshold for determining small I/O operation bottlenecks (*TooManySmallIOOps*).
developerMode (bool)	If set, additional tunable constants and metrics are made available to the user. NB: USE AT YOUR OWN RISK!!

Figure 13: User-level Tunable Constants

Tunable Name	Description
hysteresisRange (float)	Represents the fraction above and below threshhold that a test should use.
PCuseCallGraphSearch (bool)	If true, the Performance Consultant uses a callgraph-based search of the Code hierarchy, otherwise uses a module-then-function search.
PCprintDataTrace (bool)	If true, the Performance Consultant prints a full trace to stdout of all PC-related data events: data arrival at the PC, data values after filtering, etc.
PCprintTestResults (bool)	If true, the Performance Consultant prints data to the console window every time it computes a result value for an experiment.
PCprintDataCollection (bool)	If true, the Performance Consultant prints out trace information on PC-initiated instrumentation requests and disables.
PCuseIndividualThresholds (bool)	If true, the Performance Consultant will ignore the user-level tunable constants **PC_SyncThreshold,PC_CPUThreshold,PC_IOThreshold,PC_IOOpThreshold**,and use a set of hypothesis-specific developer-level tunable constants instead.
PCprintSearchChanges (bool)	If true, the Performance Consultant prints data to the console window every time it draws a conclusion for a hypothesis, or starts or stops an experiment.
PCcollectInstrTimings (bool)	Times all instrumentation requests, saving result in `TESTresult.out`
printChangeCollection (bool)	If true, the name of each metric/focus pair is printed to the console window any time it is enabled or disabled.

Figure 14: Developer-level Tunable Constants. *Use at your own risk!*

printSampleArrival (bool)	If true, the arrival of each sample from `paradynd` is printed out to the console window.
tclPrompt (bool)	If true, a Paradyn prompt is presented in the start-up shell window, allowing the user to type in and execute arbitrary Tcl language commands.
enableRequestPacketSize (float)	It represents the length of the packet sent when batching enable requests. The default value is 5.
highSyncThreshold (float)	If **PCuseIndividualThresholds** is set to true, this will be used as the Performance Consultant test threshold for *ExcessiveSyncWaitingTime*.
highCPUtoSyncRatioThreshold (float)	If **PCuseIndividualThresholds** is set to true, this will be used as the Performance Consultant test threshold for *CPUbound*.
lockOverhead (float)	If **PCuseIndividualThresholds** is set to true, this will be used as the Performance Consultant test threshold for *lockOverhead*.
minLockSize (float)	If **PCuseIndividualThresholds** is set to true, this will be used as the Performance Consultant test threshold for *minLockSize*.
highIOthreshold (float)	If **PCuseIndividualThresholds** is set to true, this will be used as the Performance Consultant test threshold for *ExcessiveIOBlockingTime*.
diskBlockSize (float)	If **PCuseIndividualThresholds** is set to true, this will be used as the Performance Consultant test value for *TooManySmallIOOps*.
seekBoundThreshold (float)	If **PCuseIndividualThresholds** is set to true, this will be used as the Performance Consultant test threshold for *seekBound*.

Figure 14: Developer-level Tunable Constants. *Use at your own risk!*

When the **Developer Mode** tunable constant is set, Paradyn makes available a number of additional "developer-mode metrics" for selection. For further details, see Section 6.

5 SELECTING RESOURCES

You specify performance data for Paradyn to collect in two parts: the type of performance data and the part(s) of the program for which you want this data collected. The parts of your program are called *resources* in Paradyn. This section discusses how to select resources. (Section 6.1 discusses how to select *metrics*—the type of performance data.)

5.1 Resources (The "Where" Axis)

The *Where Axis* is used to describe the parts of your program for which Paradyn can report performance data. It is a visual representation of different ways to specify these parts. A simple example of a Where Axis is given in Figure 15. The Where Axis is used to make all *resource*-related selections. For example, users will use the Where Axis for adding resources to a visi, and in the future for manual refinements in the Performance Consultant. This section describes the Where Axis, its visual representation, and how to make selections.

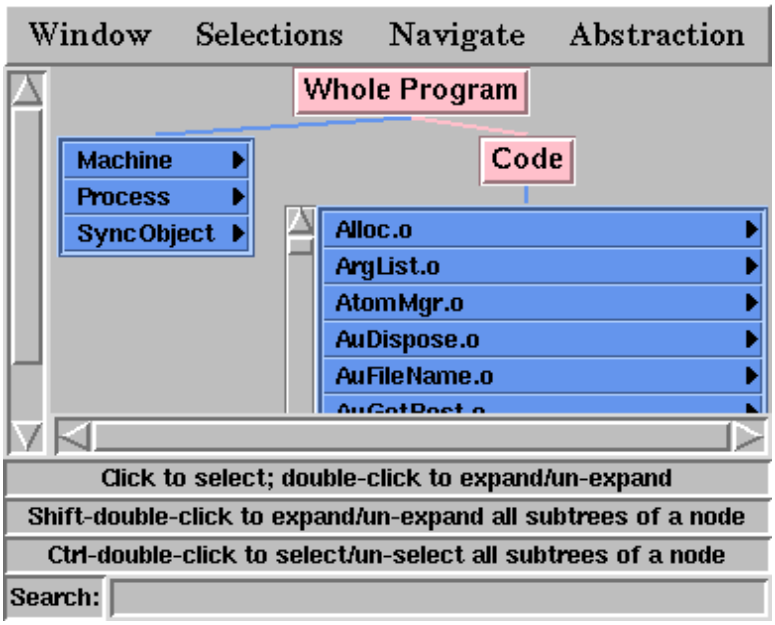


Figure 15: Where Axis window.
"Whole Program" has three unexpanded subtrees and one expanded subtree (Code)

Before we delve into specific examples of usage, a few definitions are in order:

Resources:

Resources are program components. Examples include modules, procedures, processes, barriers, locks, processor nodes, and disks. Some of the resource types are common to all programming platforms. Examples of these common resources include Modules and Procedures. Some resources are only supported on particular platforms. An example of this type of resource would be the Barrier synchronization object.

Resource Hierarchy:

Paradyn organizes all of a program's resources into hierarchies (trees). Each hierarchy represents a broad class of objects in an application. Typically, a parallel program has at least four hierarchies: Code (under which we have an application's modules, then individual functions), Process (under which we have each node in a parallel machine), Machine (these are the nodes or hosts in the parallel or distributed environment), and SyncObject (that includes such types as message tags, semaphores and barriers).

The code hierarchy contains a hierarchical representation of the code which comprises the program under examination. It is a two level hierarchy. The code space as a whole is separated into modules, which represent a high level grouping of program functionality. In general, a module corresponds to an individual source file in a higher level language, or to a single library. A module contains all of the functions located in the corresponding original source file (or files, for libraries).

There are a few instances in which the set of modules displayed in the code hierarchy will not correspond exactly to the set of source files and modules linked into the program, as discussed next.

The "`DEFAULT_MODULE`" module holds all functions which could not be assigned to any other module, either because the necessary information could not be found, or because the functions are not rightfully assigned to any of the input files or libraries which make up the given executable. On most supported platforms, this module should include only functions which are built-in by the compiler or environment, in the sense that they do not come from any user specified source files or libraries (e.g., the `_start`, `__do_global_ctors_aux`, and `__do_global_dtors_aux` functions provided in **`crt0`** by most Unix C compilers).

Some compiler and linker settings do not generate enough information to resolve functions into modules: e.g., when compiled/linked without the '`-g`' compilation flag which requests a symbol table be included in the object/executable for the use of tools such as a debugger. When parsing a file which does not contain this information, Paradyn assigns all functions to the "`DEFAULT_MODULE`". In particular, we are not aware of any compilers and linkers on the AIX platform which provide the necessary information. As such, when Paradyn is used on AIX, all functions are generally placed in that module. Note that this affects the MDL "exclude" directive (Section 12.8).

In Paradyn versions 2.1 and above, it is no longer necessary to explicitly delineate "interesting" user code with `DYNINSTstartCode` and `DYNINSTendCode` block objects. However, later Paradyn versions should still correctly parse executables which have been so built. To maintain compatibility with older Paradyn versions, when an application is linked with `DYNINSTstartCode` and `DYNINSTendCode`, any statically linked code which is outside of the range delimited by `DYNINSTstartCode` and `DYNINSTendCode` is placed in the "`DYN_MODULE`" module.

Focus:

A focus is a set of selections from the Where Axis containing exactly one resource from *each* resource hierarchy. For example, in the Where Axis of Figure 15, a focus might be the set

{/Code/Alloc.o, /Machine, /Process, /SyncObject}. The selection /Code/Alloc.o means restrict our performance data collection to only the code contained in module `Alloc.o`. The selection /Machine means all machines (nodes) on which your program is running. /Process means all processes in the program and /SyncObject means for all types of synchronization used. If you select the root node in each hierarchy, this means that Paradyn will collect data for a metric for the whole program (all nodes, processes, modules, etc.).

Performance data is collected for a particular focus. For example, suppose we made the following focus selection and requested that CPU time data be collected for this focus: {/Code/Alloc.o/XtCalloc, /Machine, /Process/psicm.pd.pn{123657_mendota}, /SyncObject}. This selection means "measure the CPU time spent in function XtCalloc while it's being executed on any machine, only when executed by process psicm.pd.pn{123657_mendota}, and for any type and instance of a SyncObject." In this example, CPU time is a *metric*. Paradyn metrics are functions that describe how the behavior of your application program changes over time. Metrics and their selection are presented in Section 6.

5.2 The Where Axis display

Resources for your application program are displayed in the Where Axis display. The resource hierarchy in Figure 15 is an example of a such a window. Many programs will have hundreds, if not thousands of resources; displaying the complete tree for all of their hierarchies and their nodes (as in Figure 16) is cumbersome to the user, who will have difficulty finding desired items.

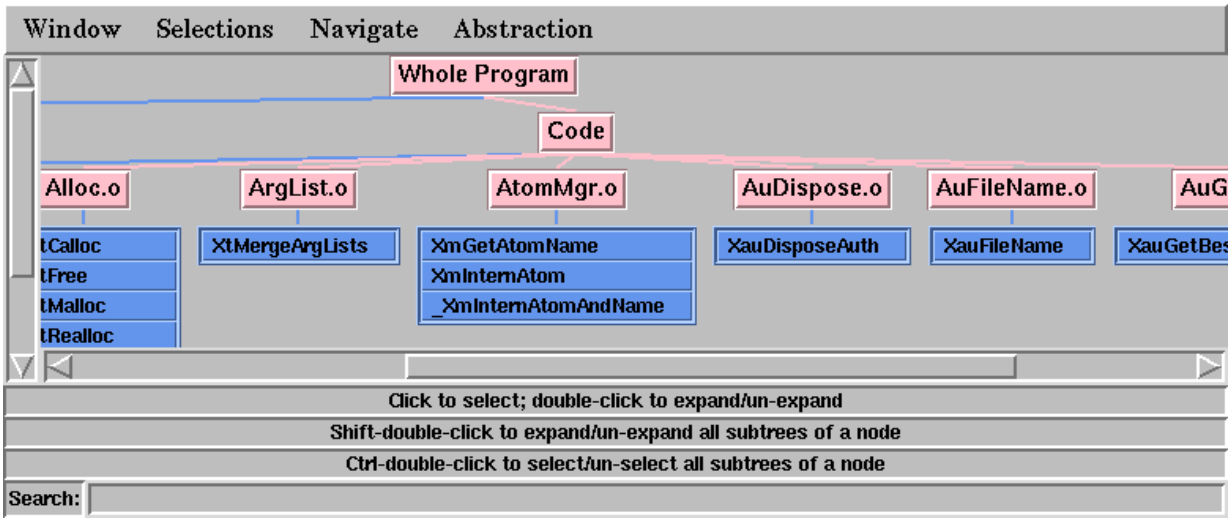


Figure 16: Showing all resources in the Where Axis display

Paradyn allows the user to control how much of the Where Axis is visible at any one time. The children of a node may be displayed as separate single nodes or be displayed together in a single *listbox*. The listbox is a compact way of representing many children of a node. For example, in Figure 15, the root node (Whole Program) has four (yes, four!) child nodes. Three of these nodes (Machine, Process, and SyncObject) are combined in the blue listbox. The fourth child of **Whole Program** is the salmon colored single node, **Code**.

If the listbox contains a large number of nodes, then it may even have a scroll bar on the side. A

triangle beside a node in a listbox means that it is not a leaf node—that the subtree is presently un-expanded to conserve screen real estate. Double-clicking on such a node will expand it from the listbox as a single node. This new single node will be salmon colored with a blue listbox below, containing its child nodes. The Code node in Figure 15 was originally displayed as a node in the listbox Whole Program. Double-clicking on the Code resulted in Code being displayed as single node. Since Code is not a leaf node, its children (a list of modules) are displayed as a listbox below.

After expanding a node, the resource desired may still be buried lower in the hierarchy. You can continue to double-click on appropriate nodes. Shift-double-click on the parent of a listbox (that is, on a pink node showing a listbox under it) will expand *all* listbox items one level.

5.3 How to select foci using the Where Axis

A focus is a selection of one resource from each resource hierarchy. To choose a focus, click the left mouse button over a resource name, thereby selecting it. Performing this operation on one resource in each hierarchy selects a single focus. An example of such a selection is shown in Figure 17. The focus selected in this figure is:

{/Code/anneal.c/a_cost, /Machine/goat, /process, /SyncObject}.

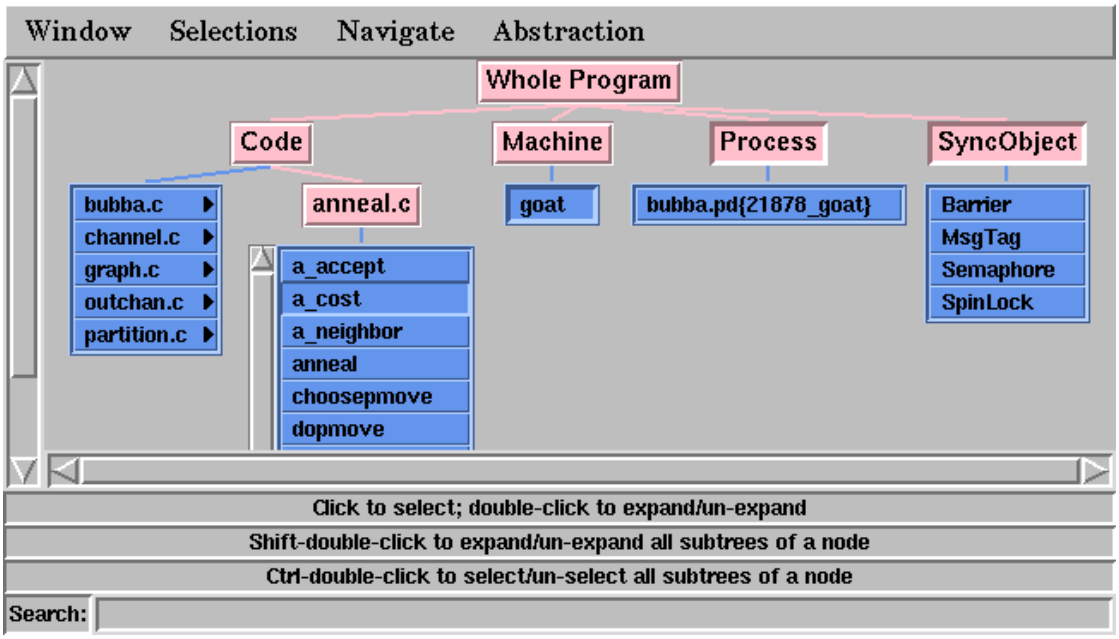


Figure 17: A single focus selected

The Where Axis also can be used to select multiple foci at the same time. Multiple selections are done by making more than one selection in a given hierarchy. The set of foci currently selected is the cross-product of all resource hierarchy selections. For example, in Figure 18 there are three resources selected from the Code hierarchy (/Code/channel.c, /Code/anneal.c, and /Code/anneal.c/a_cost), one resource selected from the Machine hierarchy (/Machine/goat), one resource selected from the Process hierarchy (/Process), and two resources selected from the SyncObject hierarchy (they are /SyncObject and /SyncObject/Semaphore). The total number of foci currently selected is therefore ($3 \times 1 \times 1 \times 2 =$

6). They are:

- {/Code/channel.c, /Machine/goat, /Process, /SyncObject}
- {/Code/channel.c, /Machine/goat, /Process, /SyncObject/Semaphore}
- {/Code/anneal.c, /Machine/goat, /Process, /SyncObject}
- {/Code/anneal.c, /Machine/goat, /Process, /SyncObject/Semaphore}
- {/Code/anneal.c/a_cost, /Machine/goat, /Process, /SyncObject}
- {/Code/anneal.c/a_cost, /Machine/goat, /Process, /SyncObject/Semaphore}

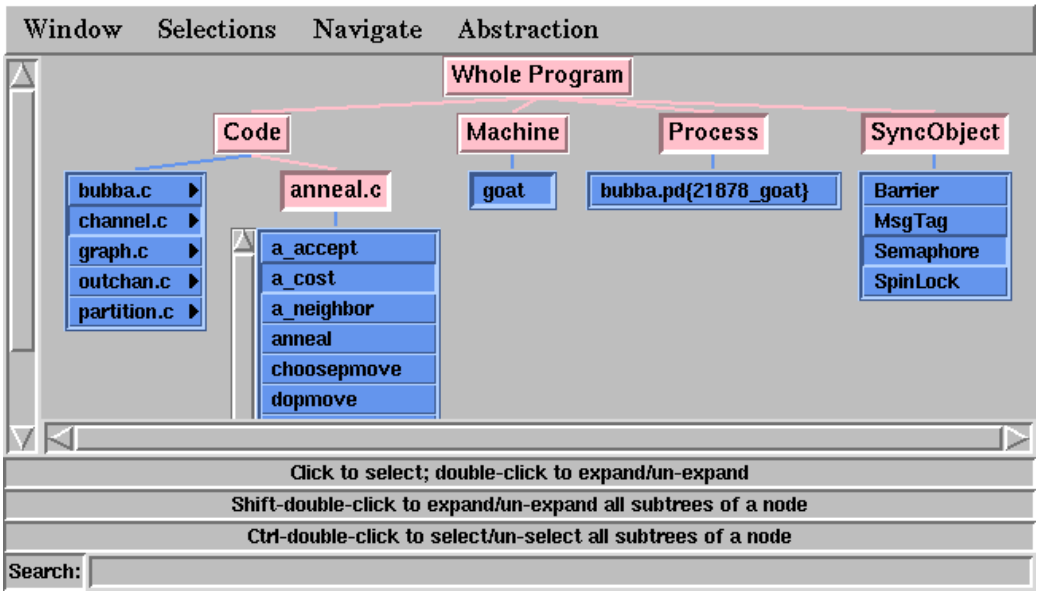


Figure 18: Multiple foci selection

5.4 The Where Axis GUI

Locating a resource

Resource names are sorted in every listbox, to ease locating resources. All sibling expanded subtrees are sorted left-to-right on screen. A subtree's sibling listbox is always leftmost, followed by its expanded items, if any. If all of a subtree's children are expanded, then no listbox is drawn.

To quickly locate a resource, you may type a resource prefix into the "Search" entry box at the bottom of the Where Axis window and press return. This feature finds, expands, and scrolls to the first resource with that prefix (if any). Continuing to press return will find the next prefix match. The search wraps around to the beginning when no more matches are found. The search is case-sensitive.

Selecting a resource

Clicking on a resource name (whether in a listbox or expanded) selects it. Clicking again deselects it. A <ctrl-dbl-click> on the root of an expanded subtree will select all of its children (but not its children's children; i.e., not recursively). Another <ctrl-dbl-click> deselects the same. Note that

when a function is selected or deselected in the code hierarchy, the selection state of the corresponding function in the Call Graph is changed (see Section 5.5 for more details).

To deselect every node in the Where Axis, choose **Clear** from the **Selections** menu.

Listbox expansion

As previously mentioned, double-clicking on a non-leaf listbox item will expand it. To quickly expand *all* (non-leaf) items of a given listbox, <shift-dbl-click> on the listbox's parent node (which is always salmon colored). Another <shift-dbl-click> on the listbox's parent will un-expand all children back into the listbox.

The navigate menu

If many Where Axis items have been expanded (e.g. a <shift-dbl-click> on a listbox containing 100 elements), it may be difficult finding your way around the Where Axis. The **Navigate** menu can help with this. After clicking on any node (whether or not it was a listbox node), the **Navigate** menu will contain every ancestor of that node (i.e., its parent, its parent's parent, and so on up to the root node). Selecting any item from the **Navigate** menu will scroll the Where Axis so the chosen item is visible.

Scrolling

The Where Axis contains traditional horizontal and vertical scrollbars for navigation. In addition, the Where Axis may be scrolled by moving the mouse to the center of the window, holding down the Alt key, and moving the mouse. The mouse pointer will remain fixed, but the Where Axis will scroll around it.

5.5 Call Graph display

A new display added to Paradyn provides a representation of the callgraph of each executable in the application: see Figure 19. Starting from each program's entry-point function (such as main for C/C++ applications), the graph of functions called from it are presented, using the same hierarchical display and search functionality as the WhereAxis.

The resource selection state of functions in the Call Graph and Where Axis are synchronized since they represent the same resources. When a function node in the Call Graph is selected, the shadow nodes in the Call Graph and the corresponding function in the code hierarchy of the Where Axis are also selected. On the other hand, when a function in the code hierarchy is selected, the primary and shadow nodes in the Call Graph representing the same function are also selected.

One view option currently supported allows the module containing each function to be prepended to its name.

The call graph displays statically determined edges, that is function calls that are not made through function pointers. Function calls made through function pointers will be displayed in the

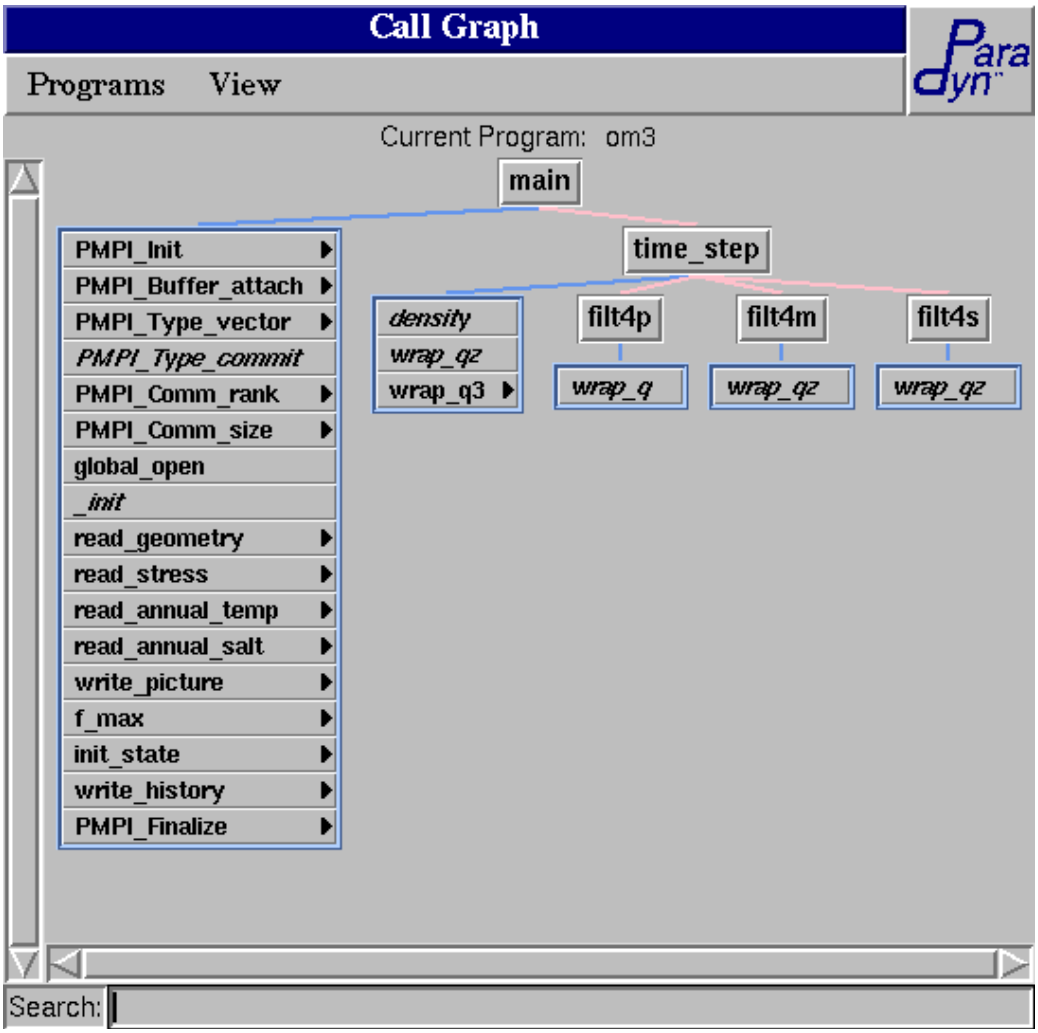


Figure 19: Callgraph display

call graph when the Performance Consultant instruments the function containing the call through the function pointer.

6 SELECTING METRICS

A metric is a time-varying function that quantifies some aspect of program performance. This section illustrates the metrics selection process in Paradyn. Section 6.1 describes how to select metrics and Section 6.2 describes all the metrics currently defined in Paradyn.

6.1 How to select metrics

When you wish to display or modify performance data, you must select a focus (see Section 5) and list of metrics. This section discusses how to select *metrics*—the type of performance data.

The Metrics Dialog Box appears when Paradyn needs the user to specify one or more metrics for some operation. Currently, there is only one place in Paradyn where the Metrics Dialog Box is used: when choosing metric-focus pairs to add to a *visi*. Choosing a set of metric-focus pairs involves making selection(s) from both the Metrics Dialog Box (for the metrics) and from the where axis (for the foci). In this section, we will discuss only metric selection; Section 5 describes in detail how to make focus selections.

A sample metrics dialog box appears in Figure 20. Note that the metrics which appear in the

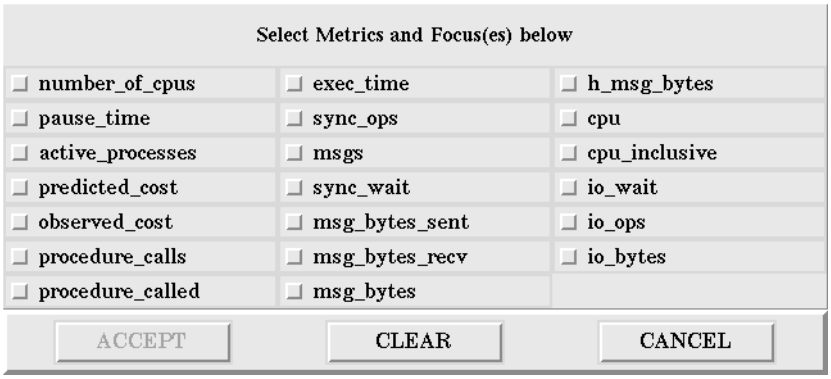


Figure 20: Metrics dialog box

dialog box are specific to the platform being run (such as sequential vs. parallel/PVM). In addition, if the **developerMode** tunable constant is set (see Section 4.3), the "developer mode metrics" are also made available. Complete descriptions of the various metrics are provided in Section 6.2; expert users can use Paradyn Configuration Language's Metric Description Language (Section) to add custom metrics.

When the metrics dialog box appears, select one or more metrics from the given list. To select a metric, simply click the mouse in any checkbox. Selected metrics will have a red square to the left of the metric name in the dialog box. Figure 21 shows how the metrics dialog box of Figure 20 would look after the metrics **cpu**, **msg_bytes_sent**, and **procedure_calls** were selected. Clicking on a previously-selected metric will deselect it. Clicking on the **CLEAR** button at the bottom will deselect

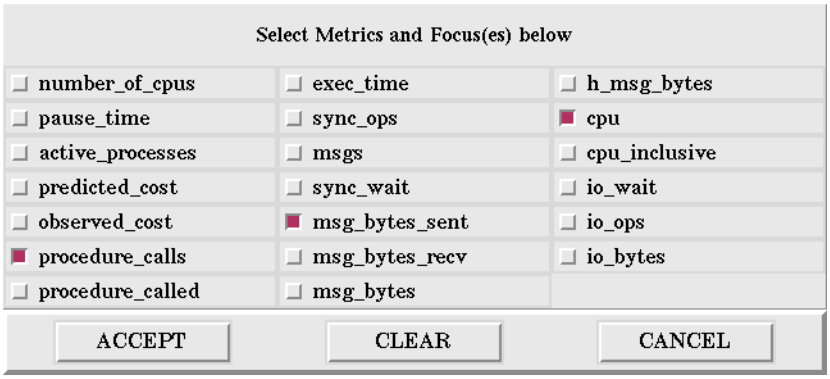


Figure 21: Metrics dialog box with several metrics selected

all selected metrics.

When done with metric selections, press **ACCEPT** or **CANCEL**. The metrics dialog box will disappear at that time; it will reappear the next time a metric selection is required[1].

6.2 Metric Descriptions

A list of all current metrics is presented in Figure 22. As we have described in the previous Section, expert users can create their own custom designed metrics using the Paradyn Configuration Language. Most of the metrics that appear in Figure 20 and Figure 21 were created using this language and are provided within Paradyn. Additionally, an expert user can select *Developer Mode* metrics (Figure 22). Developer mode metrics are mostly *internal metrics*, or metrics that have been hard coded into Paradyn, that can be used to monitor Paradyn's own performance or for debugging purposes. Developer mode can be selected from the **Tunable Constants** option of the **Setup** menu, as it is illustrated in Figure 11 (Section 4). After **developerMode** is selected, a larger list of metrics will appear in the metrics dialog box.

It is important to make a distinction between three types of metrics: *normalized*, *unnormalized* and *sampled*. Normalized metrics are time related metrics that are being computed as a percentage (e.g., *cpu*). Unnormalized metrics are mainly computed using counters (e.g., *procedure_calls*) and they are usually expressed as a rate (e.g., operations per second). Sampled metrics are like unnormalized metrics, but the units are not represented as a rate (e.g., operations).

1. This contrasts with the *Where Axis* window (Section 5), which is kept open because the ability to browse a program's resource hierarchy at any time is desirable.

Metric	Description	Units	Visi Axis Label
active_processes	Each bin represents the number of processes active during the corresponding interval of time. Aggregation is the average number of processes active over an interval of time.	# of processes	operations
cpu	Each bin represents the percentage of CPU time spent during the corresponding time interval. Aggregation is total CPU time over an interval.	CPUs	CPUs
cpu_inclusive	Same as *cpu* but includes called procedures in the process time calculation.	CPUs	CPUs
exec_time	Each bin represents the elapsed wall clock time per unit during the corresponding time interval. Aggregation is the sum over the interval.	exec time	CPUs
io_bytes	This metric represents the number of bytes for Input/Output operations. Currently, only "read" and "write" are supported as input/output operations for UNIX, MPI, and PVM. On Windows, "ReadFile" and "WriteFile" are used.	#bytes read/ written	bytes
io_ops	Number of Input/Output operations. IO operations are the same as for *io_bytes*.	# IO ops	operations
io_wait	Time spent during Input/Output operations. IO operations are the same as for *io_bytes*.	CPUs	CPUs
io_wait_inclusive	Same as *io_wait* but includes called procedures in the process time calculation.	CPUs	CPUs
msgs	The total number of messages sent and received. The unit is operations per unit of time. Aggregation is the sum of all sends and receives over the time interval. Send and receives are defined as follows: **UNIX send**:"write" **UNIX recv**:"read"	#msgs sent/recv	msgs

Figure 22: Metrics defined in Paradyn

Metric	Description	Units	Visi Axis Label
pp_msgs	Similar to *msgs*, but it counts the number of point-to-point messages (only for MPI applications). Point-to-point communications are defined as follows: `PMPI_Send, PMPI_Bsend, PMPI_Ssend, PMPI_Isend, PMPI_Issend, PMPI_Recv, PMPI_Irecv, PMPI_Sendrecv, PMPI_Sendrecv_replace`.	#msgs	msgs
cc_msgs	Similar to *msgs*, but it counts the number of collective communications (only for MPI applications). Collective communications are defined as follows: `PMPI_Bcast, PMPI_Alltoall, PMPI_Alltoallv, PMPI_Gather, PMPI_Gatherv, PMPI_Allgather, PMPI_Allgatherv, PMPI_Reduce, PMPI_Allreduce, PMPI_Reduce_scatter, PMPI_Scatter, PMPI_Scatterv, PMPI_Scan`.	#msgs	msgs
msg_bytes	Number of message bytes sent and received. Aggregation is the total number of bytes sent and received. Send and receive are defined as follows: **UNIX**: “read”, “write”	#bytes sent/recv	bytes
msg_bytes_recv	Number of message bytes received per unit of time. Aggregation is the total number of bytes received. Message receives are defined as for *msgs*.	msg-bytes recv’d	bytes
pp_msgBytesRecv	Similar to *msg_bytes_recv*, but only for receive messages involved in point to point communications (MPI applications only). These point to point communications are defined as follows: `PMPI_Recv, PMPI_Irecv, PMPI_Sendrecv, PMPI_Sendrecv_replace.`	msg bytes recv’d	bytes

Figure 22: Metrics defined in Paradyn

Metric	Description	Units	Visi Axis Label
cc_msgBytesRecv	Similar to *msg_bytes_recv*, but only for receive messages involved in collective communications (MPI applications only). These collective communications are defined as follows: PMPI_Bcast, PMPI_Alltoall, PMPI_Alltoallv, PMPI_Gather, PMPI_Gatherv, PMPI_Allgather, PMPI_Allgatherv, PMPI_Reduce, PMPI_Allreduce, PMPI_Reduce_scatter, PMPI_Scatter, PMPI_Scatterv, PMPI_Scan.	msg bytes recv'd	bytes
msg_bytes_sent	Number of message bytes sent per unit of time. Aggregation is the total number of bytes sent. Message sends are defined as for "msgs".	msg-bytes sent	bytes
pp_msgBytesSent	Similar to *msg_bytes_sent*, but only for send messages involved in point to point communications (MPI applications only). These point to point communications are defined as follows: PMPI_Send, PMPI_Bsend, PMPI_Ssend, PMPI_Isend, PMPI_Issend, PMPI_Sendrecv, MPI_Sendrecv_replace.	msg bytes sent	bytes
cc_msgBytesSent	Similar to *msg_bytes_sent*, but only for send messages involved in collective communications (MPI applications only). These collective communications are defined as follows: PMPI_Bcast, PMPI_Alltoall, PMPI_Alltoallv, PMPI_Gather, PMPI_Gatherv, PMPI_Allgather, PMPI_Allgatherv, PMPI_Reduce, PMPI_Allreduce, PMPI_Reduce_scatter, PMPI_Scatter, PMPI_Scatterv, PMPI_Scan.	msg bytes sent	bytes
number_of_cpus	Number of CPUs in the system.	#CPUs	#CPUs
observed_cost	**Internal metric**: Indicates the effect on the application from collecting data. Its purpose is to check that the overhead of data collection does not exceed pre-defined levels, and should these levels be exceeded, it reports to the higher level consumers of data.	slow-down	slowdown

Figure 22: Metrics defined in Paradyn

Metric	Description	Units	Visi Axis Label
pause_time	Each bin represents the fraction of time in which the application program was paused by Paradyn. Maximum value is 1.0. Aggregation is the total time paused over an interval.	pause time	CPUs
predicted_cost	**Internal metric**: Expected overhead of collecting the data necessary to compute a metric for a particular focus or combination of resources. The predicted cost is expressed as the percentage utilization of CPU.	slowdown	slowdown
func_calls_to	Represents the number of procedure calls to the specified function or module.	#calls	operations
func_calls_by	Represents the number of procedure calls made by the selected function or by all of the functions in the selected metric.	#calls	operations
sync_ops	The number of synchronization operations per unit of time. Aggregation is the sum. The following are defined as synchronization operations: **UNIX**: "write", "read", "recv", "recvfrom", "select", "sendmsg", "send", "sendto"	#sync ops	operations
sync_wait	The elapsed wall time spent waiting for a synchronization operation. Aggregation is the sum of all waiting time. The following will be included in the reported times: **UNIX**: "write", "read"	sync wait time	CPUs
sync_wait_inclusive	Same as sync_wait, but includes called procedures in the process time calculation.	sync wait time	CPUs

Figure 22: Metrics defined in Paradyn

Metric	Description	Units	Visi Axis Label
sampling_rate	**Internal metric**: It is the time interval at which samples are taken of the application by the daemon.	milliseconds	milliseconds
stackwalk_time	**Internal metric**: Amount of time the Paradyn daemon(s) spend walking the stack. This is done when timers or counters need to be manually started.	stackwalk time	CPUs
numOfActCounters	**Internal metric**: The number of active count based metric-focus pairs. Or the number of counters that are actively being sampled.	# of counters	operations
numOfActProcTimers	**Internal metric**: The number of active process time based metric-focus pairs. Or the number of process timers that are actively being sampled.	# of timers	operations
numOfActWallTimers	**Internal metric**: The number of active wall time based metric-focus pairs. Or the number of wall timers that are actively being sampled.	# of timers	operations

Figure 23: Developer Mode Metrics defined in Paradyn

7 CONTROLLING VISIS

This section describes how to start and stop visualizers (known as 'visis') from Paradyn.

7.1 Starting

A new visualization can be requested by pressing the **Visi** button from the Paradyn main window menubar (Figure 24), which opens the **Start A Visualization** dialog.

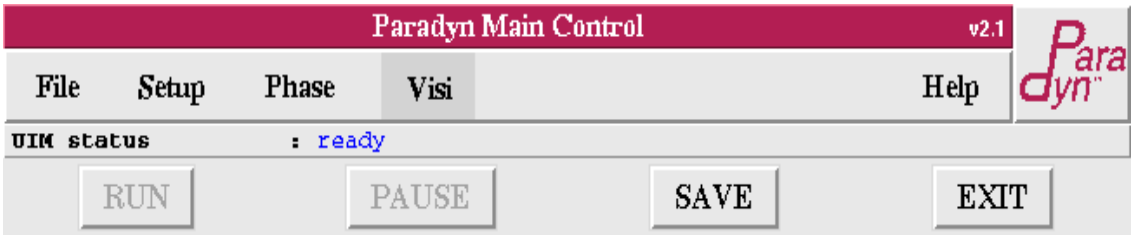


Figure 24: Paradyn Main Control window

This dialog presents a list of visualizations to choose from as shown in Figure 25. Visualizations can be started so that they receive either global phase performance data or current phase performance data. The selection in Figure 25 is for a Histogram visualization that will receive performance data from the global phase of the application's execution.

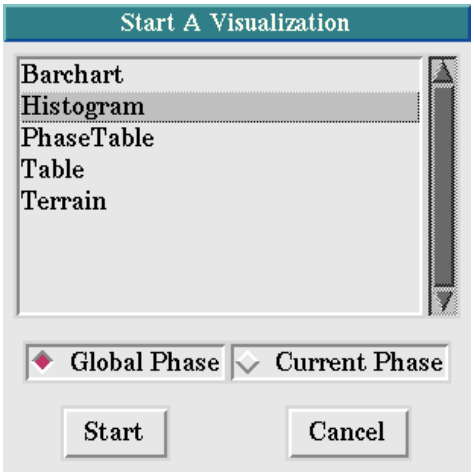


Figure 25: Start A Visualization menu

Once a visualization has been defined, metric and focus menuing is usually initiated before the visualization process is started. Whether or not this menuing is done is determined by the *force* flag setting in the PCL entry for this visualization. If the force option is set then the visualization process is started without metric and focus menuing. This is typically used for starting visualiza-

tions that do not want to enable data flow before starting. The Phase Table visualization is an example of one which should have the force option set. For other visualizations, once menuing is done, and at least one metric/focus combination is successfully enabled, the visualization process is started.

The instrumentation required to collect data to be shown in a visualization can be *deferred* if an application process is executing in the neighborhood of an instrumentation point. If instrumentation is deferred, Paradyn displays a message box like the one shown in ??? listing the metric-focus pairs with deferred instrumentation.

Figure 26: Message box indicating instrumentation has been deferred

7.2 Stopping

Each visualizer has a menu option to quit that invokes the VisiLib **QuitVisi** routine which takes care of disabling data collection and cleans up any associated state, then exits.

8 PHASES

Phases in Paradyn are contiguous time-intervals within an application's execution. There are two kinds of phases: the *global phase* and *local phases*. The global phase starts at the begining of the program execution and extends to the current time. Local phases are non-overlapping subintervals of the global phase. When a new local phase is defined in the system, the current local phase ends and all data collection for the current phase stops. Data collection for the new phase will occur at a finer granularity than collection for data the global phase. At any time in the program's execution, data collection can be enabled for one or both of the the current local phase and the global phase. Similarly, a Performance Consultant search (Section 9) can be started for the global phase of an application's execution, or can be restricted to search over only the current local phase of execution.

8.1 Starting a new phase

A new local phase can be defined by selecting **Start** under the **Phase** menu of the Paradyn main window (Section 3) or **Start Phase** from the Phase Table display menu (Figure 27). When a new phase is defined, any visualizations defined for the current local phase stop receiving performance data. Similarly, if the Performance Consultant is active for the current phase, its search ends when a new phase is defined.

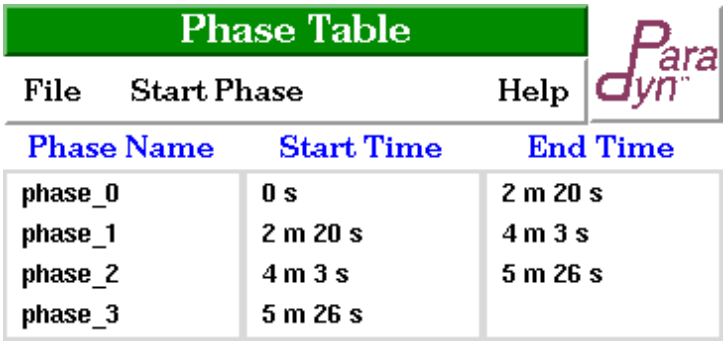


Figure 27: Phase Table Display

8.2 Visualizations and Phases

Visualizations can show data for either the local phase or the global phase. Local phase visualizations receive and display performance data from the phase's start time until the phase ends. Typically, local phase data is collected at a finer granularity than global phase data. Figure 28 shows a real time histogram visualization that has been defined for the global phase, and Figure 29 shows one that has been defined for specific phase.

8.3 The Performance Consultant and phases

The Performance Consultant (Section 9) can simultaneously search both the local phase and the global phase. Complete details are given in the Performance Consultant section; searching on multiple phases in particular is discussed in that section.

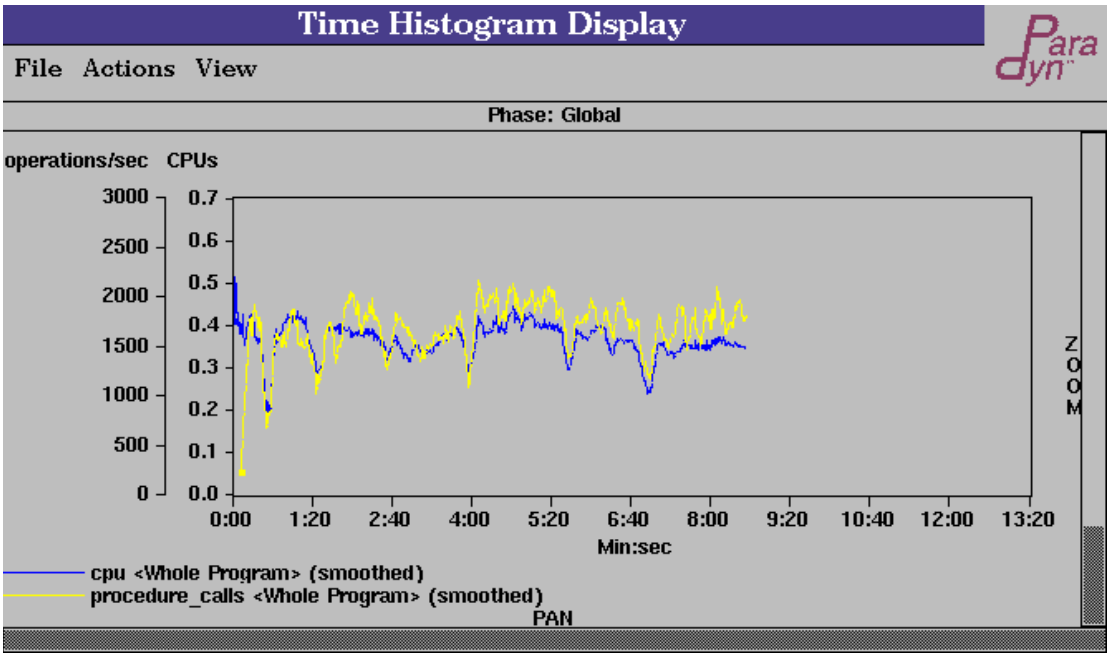


Figure 28: Time Histogram: Global Phase

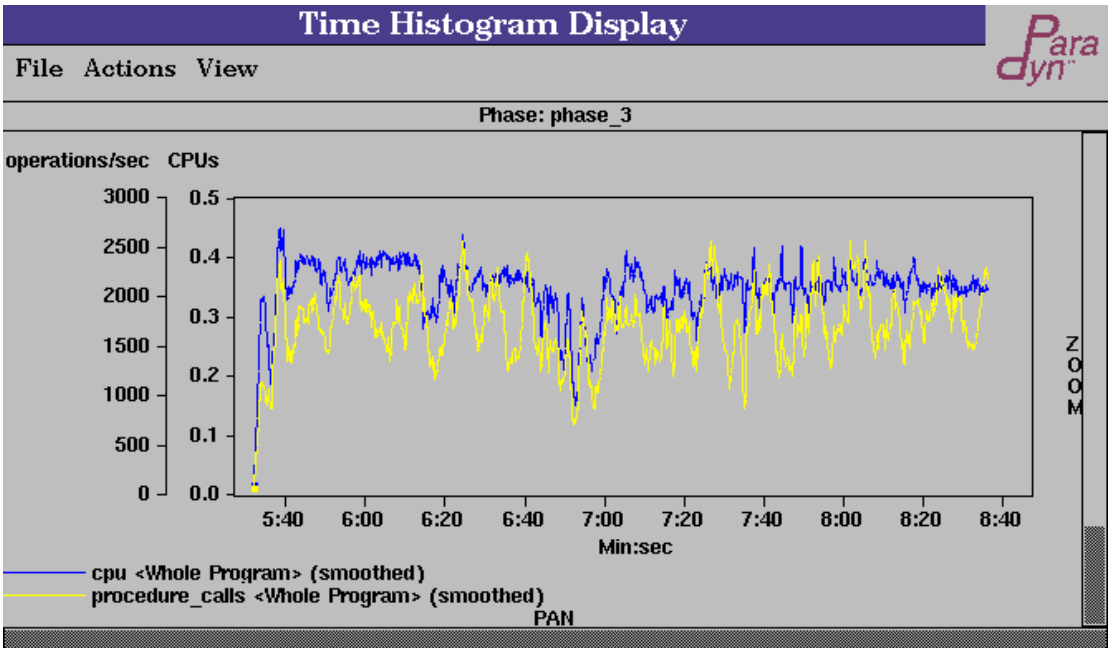


Figure 29: Time Histogram: Local Phase (3)

9 PERFORMANCE CONSULTANT

Paradyn provides many options for selecting and displaying performance information about your application program. Sometimes these options can be overwhelming. In a large, complex program, it can be difficult to know where to start looking for performance problems, and Paradyn's Performance Consultant is designed to help. The Performance Consultant (PC) helps identify the type of performance problems ("why"), where in the program these problems occur ("where") and the time during the execution during which the problem occurred ("when"). This "why-where-when" model of searching for performance problems is called the W^3 (pronounced "W- cubed") Search Model and forms the core of the PC.

The PC is automated so that, in its normal mode of operation, you simply tell it to start searching for performance problems. The PC will continually select and refine which performance metrics are enabled and for which foci they will be enabled.

The Performance Consultant in Release 3.0 of Paradyn introduced a couple of significant changes to previous behavior. Firstly, the former Process/Thread hierarchy is now relocated underneath Machine, so that searches progress from considering machines to processes (to threads where supported), instead of separately considering machines and processes together. Secondly, a callgraph-based search of the Code hierarchy replaces (as default) the former unwieldy modules-then-functions search.[1]

In this section, we describe the W^3 Search Model (Section 9.1), the components of the Performance Consultant's window (Section 9.2), how to interpret what the PC tells you (Section 9.4), and (once you get a bit of experience) how to adjust and fine-tune its operation (Section 9.5).

9.1 The W^3 search model

The Performance Consultant automatically locates potential bottlenecks in your code. The PC describes each bottleneck by stating *why* there is a problem (the *hypothesis*), and *where* in the application the problem was found (the *focus*, see Section 5*)*. You can direct the search to find out when the problem occurred by including either the entire execution or a particular phase of its execution.

The "Why" Axis: The PC includes the definition of a set of generic performance problems. These problems, called "hypotheses", are typically of the form:

```
PerfMetricX > SpecifiedThreshold
```

where `PerfMetricX` is some metric defined in Paradyn (Section 6) and the `SpecifiedThreshold` is a value that you can set by using a Tunable Constant (Section 4). The threshold value is typically expressed as a fraction (between 0 and 1) of the execution time of the application program. Each hypothesis may also contain pruning directives, which cause some portion of the resource hierarchy to be ignored while searching.

---

1. The former search behavior is available by setting (before opening the Performance Consultant window) a developer mode Tunable Constant: PCuseCallGraphSearch.

The "Where" Axis: A focus in Paradyn allow you to constrain a performance metric to a particular subset of program resources. The PC makes step-by-step selections in the Where Axis, as it tries to isolate the cause of performance problems.

The "When" Axis: The PC can look for performance problems whose effect is large enough to stand out over the total execution of the application program, or it can look for problems that stand out during a restricted interval of time. You can associate a PC with each phase (Section 8). The PC associated with the global phases searches for performance problems that affect the entire program execution; the PC associated with a local phase searches for problems that affect (at least) that interval of execution.

Depending on the complexity of the application program, i.e., the number of nodes in the Where Axis for the application, the number of hypothesis/focus pairs that could be explored might be quite large. On the other hand, the goal is to find the handful of most troublesome bottlenecks in the application. Any hypothesis/focus pair that doesn't exceed the threshold does not require further attention; realistically, any that exceeds the threshold for only a short interval of time won't get any attention. For this reason the PC only reports bottlenecks that exist for a significant portion of the overall phase being tuned.

9.1.1 The Why Axis

The space of all possible hypotheses (such as synchronization-bound, CPU-bound, etc.) is

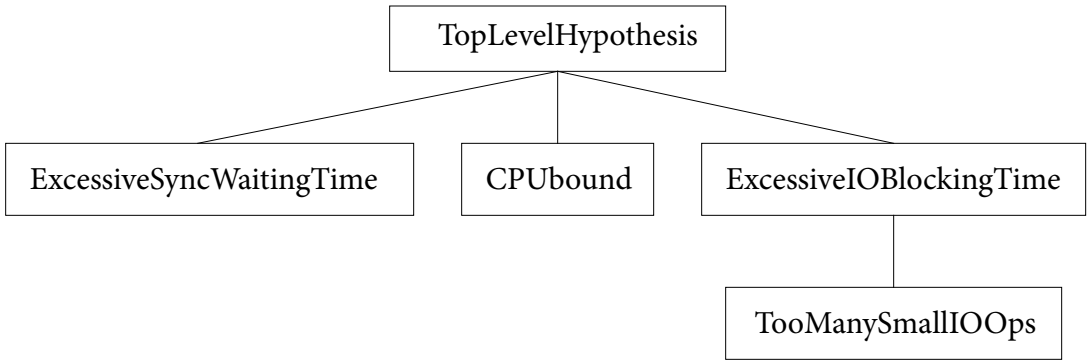


Figure 30: The Why Axis

called the Why Axis. The root hypothesis is the generic TopLevelHypothesis. This hypothesis is considered true if any hypothesis at the next level is true. The remaining hypotheses are:

- **CPUbound**: Compares CPU time to the tunable constant **PC_CPUThreshold**. Searching through /SyncObject and /Process hierarchies is disabled.
- **ExcessiveSyncWaitingTime**: Compares total synchronization waiting time to the tunable constant **PC_SyncThreshold**.
- **ExcessiveIOBlockingTime**: Compares total I/O waiting time to the tunable constant **PC_IOThreshold**. Searching through the /SyncObject hierarchy is disabled.

- **TooManySmallIOOps**: Compares average number of bytes per I/O operation to **PC_IOOpThreshold**. Searching through the /SyncObject hierarchy is disabled.

If a particular hypothesis in the Why Axis tests true, the PC will try to test the children of that hypothesis next. When the Performance Consultant searches along the Why Axis in this way to test more detailed hypotheses for a particular focus, we say a Why Axis refinement has been made.

9.1.2 The search strategy

When a new search is started, the Performance Consultant makes instrumentation requests to evaluate the topmost levels of the why and where axes; that is, it evaluates each top level hypothesis (CPUBound, SyncWaiting, IOBlocking) for WholeProgram. These particular hypothesis/focus pairs will continue to be evaluated for the entire phase.

There are two questions of interest here: when is the search expanded, and how is expansion done? The search is expanded anytime a (hypothesis : focus) pair tests true. The only exception is at start-up, when an initial set of (hypothesis : focus) pairs are enabled. If, at any time, a (hypothesis : focus) pair (h : f) tests true, then the following hypothesis:focus pairs will be added to the search: (h : all child foci of WholeProgram), plus (all child hypotheses of h : f). The why axis, and each of the resource hierarchies, are trees, so refining one step in the search is defined as moving down along a single edge in either the why axis or one of the resource hierarchies. For example, from (ExcessiveIOBlockingTime : WholeProgram), using the resource hierarchy in Figure 17, the following set of (hypothesis : focus) pairs would be added:

1. One step along the Why Axis:
   (TooManySmallIOOps : WholeProgram)
2. One step along the Code hierarchy:
   (ExcessiveIOBlockingTime : bubba.c)
   (ExcessiveIOBlockingTime : channel.c)
   (ExcessiveIOBlockingTime : graph.c)
   (ExcessiveIOBlockingTime : outchan.c)
   (ExcessiveIOBlockingTime : partition.c)
   (ExcessiveIOBlockingTime : anneal.c)
3. One step along the Machine hierarchy:
   (ExcessiveIOBlockingTime : goat)

All of the new (hypothesis : focus) pairs resulting from this expansion generate instrumentation requests, and, if possible, data collection begins immediately. However, the total amount of instrumentation active at any given time during the tuning session is limited by an internal cost-tracking system. If the total cost of currently enabled instrumentation for all visualizations and searches exceeds the cost threshold, new (hypothesis : focus) pairs are queued and activated after some other instrumentation is disabled.

Each (hypothesis : focus) pair is represented as a node of a directed acyclic graph (DAG), called the Search History Graph (SHG). The root node of the SHG represents the pair (TopLevelHypothesis : WholeProgram), and its child nodes represent the list of possible refinements chosen as described above. If a SHG node tests false, it is not expanded. After a certain minimum observation interval, testing on all but the topmost level false nodes is halted. If an already-

expanded node changes from true to false, then testing is halted for all of its children.

9.2 Running the Performance Consultant

In this section, we describe how to run the Performance Consultant on an application. Section 9.2.1 describes the Performance Consultant Window, Section 9.2.2 describes starting and stopping a search, and Section 9.2.3 provides a detailed description of the Search History Graph.

9.2.1 The Performance Consultant window

The Performance Consultant window may be opened any time after an application has been defined (see Section 2.4) by choosing **Performance Consultant** from the **Setup** menu of Paradyn's main window. Figure 31 shows a sample Performance Consultant window.

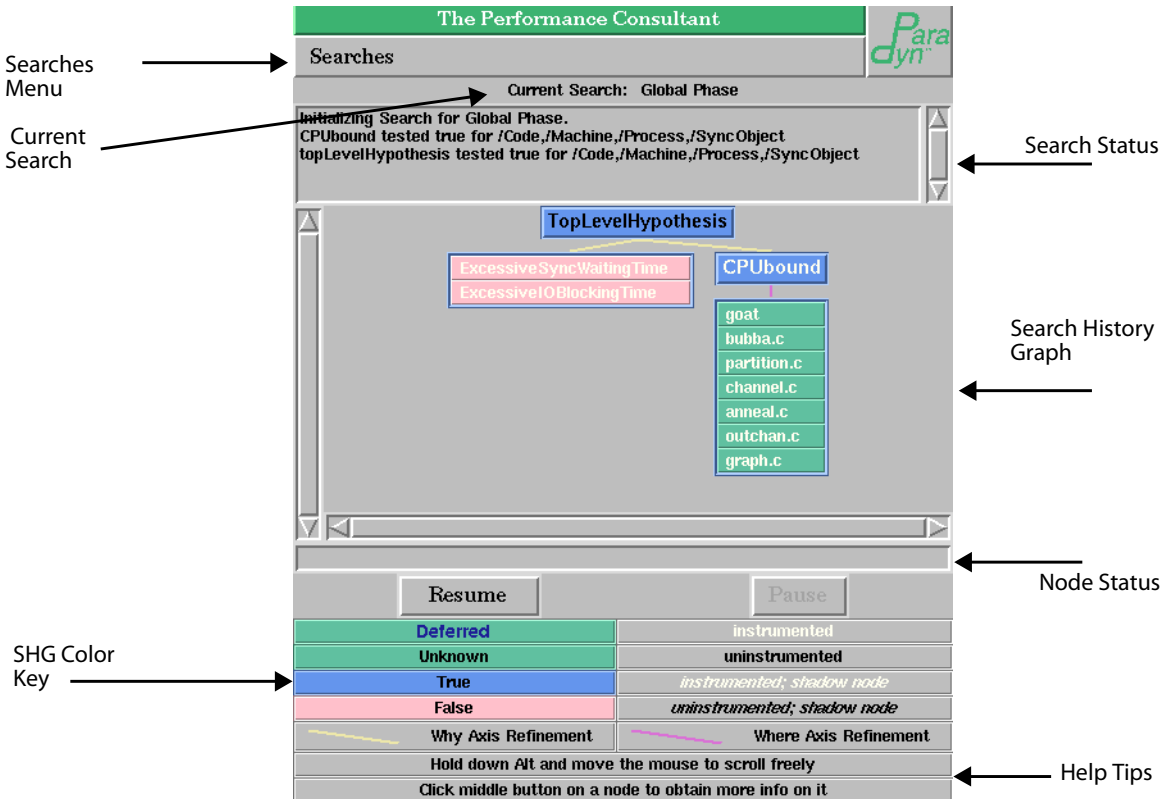


Figure 31: A sample Performance Consultant window

The **Searches** menu contains a list of all possible phases on which a search has been requested or may be started, including the default, "Global", for whole program searches. If you have not defined any phases for the application, then you will see only two choices, "Global" and "Current." The currently displayed phase has a blue diamond next to its name; the others have gray diamonds. Choosing items under this menu allows you to page through the search displays for all active, paused, and completed searches. When a new phase is defined, the Performance Consultant detects it, and adds the new phase's name to its **Searches** menu.

The **Current Search** line gives the name of the currently displayed search phase. At any given time, it is always the same as the **Searches** menu item having a blue diamond before its name.

The **Search Status** box is a scrolling text display just beneath the **Current Search** line. From time to time, the Performance Consultant adds some information about the currently displayed search in this box; examples include when refinements are made, phases end, and unexpected error conditions occur. As with most items in the Performance Consultant window, the information displayed is specific to the currently displayed search; changing searches (by choosing a different item under the **Searches** menu) will show different information.

The **Search History Graph Display** is the resizable window below the status box. The Search History Graph is a compact graphical display of the history of refinements made by the Performance Consultant. Each search has a distinct Search History Graph; changing searches will show a different Search History Graph. Section 9.2.3 discusses the Search History Graph in detail.

The **Node Status** displays extra information for a given Search History Graph node. To see the full description for any node in the SHG display, click the middle mouse button on the node. For example, the node status line in Figure 35 shows the full hypothesis and focus for the blue node labeled goat. For convenience, this line of information will remain present until the next time a node is clicked on with the middle mouse button. The **Search** and **Pause** buttons are described in Section 9.2.2, where we discuss running the Performance Consultant.

The **SHG Color Key** explains the colors and display styles used in the SHG display. We discuss each key item in Section 9.2.3. To conserve screen space, the SHG Color Key may be removed by setting the tunable constant **showShgKey** to false.

Help Tips describe mouse and key presses in the Performance Consultant window. To conserve screen space, these may be removed by setting the tunable constant **showShgTips** to false.

9.2.2 Starting and stopping a search

To start the currently displayed search, click on the **Search** button in the Performance Consultant window. The PC directs instrumentation insertion to begin locating application bottlenecks. Note that the application program must be running for data to be collected for the PC; if the application has not been started or has been paused, (re-)start it by pressing the **RUN** button of the Paradyn Main Window (see Section 2). Current Phase Searches may also be started at the same time a new phase is defined. To do so, choose **Start with Performance Consultant** from the **Phase** menu of Paradyn's main window.

The **Pause** button stops the Performance Consultant temporarily, and removes all instrumentation for the particular search displayed. Note that it does not stop the application itself; its only effect is on the PC's currently displayed search. To resume a search after pausing, press **Resume**.

A search ends when its phase ends; for a current phase, this is when you define the start of a new phase; for the global phase, it is when the application terminates.

9.2.3 The Search History Graph display

As the Performance Consultant searches for bottleneck(s), it leaves a record of its progress in the *Search History Graph*. Initially, the Search History Graph contains only a single item: TopLevelHypothesis. A few moments after the search begins, the Search History Graph will look like that in Figure 32. The three items within the listbox below TopLevelHypothesis are what the Perfor-

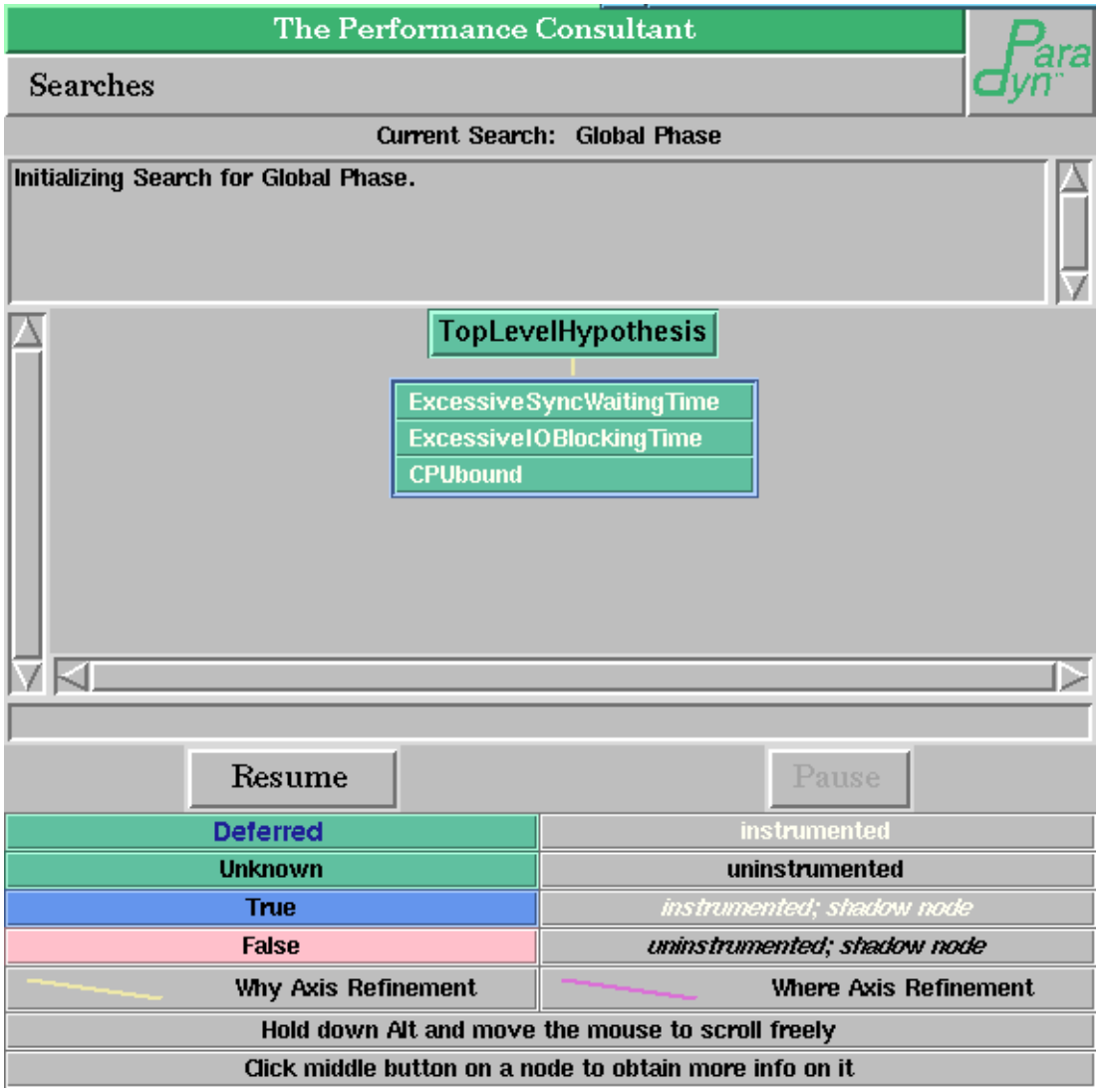


Figure 32: The Performance Consultant's search begins

mance Consultant first tests the program for—excessive synchronization waiting time, excessive I/O blocking time, or CPU bound. To use the terminology given above, the Performance Consultant is presently trying to find out *why* the program is running slowly, as opposed to *where* (what program resource(s)) it is running slowly. Whenever items are added to the Search History Graph, we say that a *refinement* has been performed. In this example, a *Why Axis refinement* has been performed, indicated by the yellow line connecting TopLevelHypothesis to its descendant listbox. As

shown in the window's key area at the bottom of the Performance Consultant window, a yellow line is a Why Axis refinement; a purple line is a Where Axis refinement.

Each item in the Search History Graph of Figure 32 has a green background. As shown in the window's key area, a green background indicates **Unknown** status; that is, we do not yet know whether any of ExcessiveSyncWaitingTime, ExcessiveIOBlockingTime, or CPUBound are true or false, since we have just begun the Performance Consultant's search. Also, the text of each item in the listbox has a white foreground. As shown in the window's key area, white text indicates an *active* test; that is, the Performance Consultant has instrumented the program to perform the test and is collecting data for it. If an experiment's instrumentation has been deferred because an application process was executing in the neighborhood of an instrumentation point, the text of the listbox item will be drawn in blue. When the instrumentation is eventually inserted, the item's text will be redrawn in white.

We continue the search until the Performance Consultant has made a further refinement (Figure 33). First, note that the Performance Consultant has decided that the program is CPU bound (because CPUbound is drawn with a blue background). The nodes (ExcessiveSyncWaitingTime : WholeProgram) and (ExcessiveIOBlockingTime : WholeProgram) have both tested false, so their background color is now pink. Although all three of these nodes will continue to be tested (which we see by the white text), only the true node, CPUBound, has been expanded to try to further refine the bottleneck. As a result of the search, a listbox below CPUbound has appeared. The line connecting CPUbound to its children is drawn in purple, since is a Where Axis refinement. Each item in the listbox contains program resources that are being examined with the CPUbound hypothesis. The Performance Consultant has decided that the program is CPU bound; now it's trying to refine the bottleneck to (in this case) a certain machine (goat) or in the Code hierarchy.

Double-clicking on a true node (such as CPUbound in Figure 33) collapses the display so its children are no longer shown. Because it saves screen space, this is useful for traversing large complex search graphs. In the example of Figure 33, double-clicking on CPUbound would put CPUbound into the listbox with ExcessiveSyncWaitingTime and ExcessiveIOBlockingTime. A triangle will appear next to CPUbound in the listbox to indicate that it has children which are presently being hidden to save screen space. To expand the node's children, double-click on the name in the listbox.

Screen space can be saved in the Search History Graph by hiding certain combinations of node types. For example, you may wish to view only nodes which the Performance Consultant has determined to be true bottlenecks (blue nodes). Or, you may wish to show all but those nodes which have been determined *not* to be bottlenecks (pink nodes). There are seven such node characteristics; boolean tunable constants (Section 4) can be set to show or not to show nodes of any given characteristic. We now briefly describe each node characteristic; they are discussed in more detail in Section 9.4.

9.3 Interpreting the results of callgraph-based search

The callgraph-based search is similar in principle to the module-then-function search, differing in the way searches and refinements proceed through the Code hierarchy, and in the initial use of inclusive metrics instead of exclusive metrics. Since inclusive metrics are less costly/intrusive and

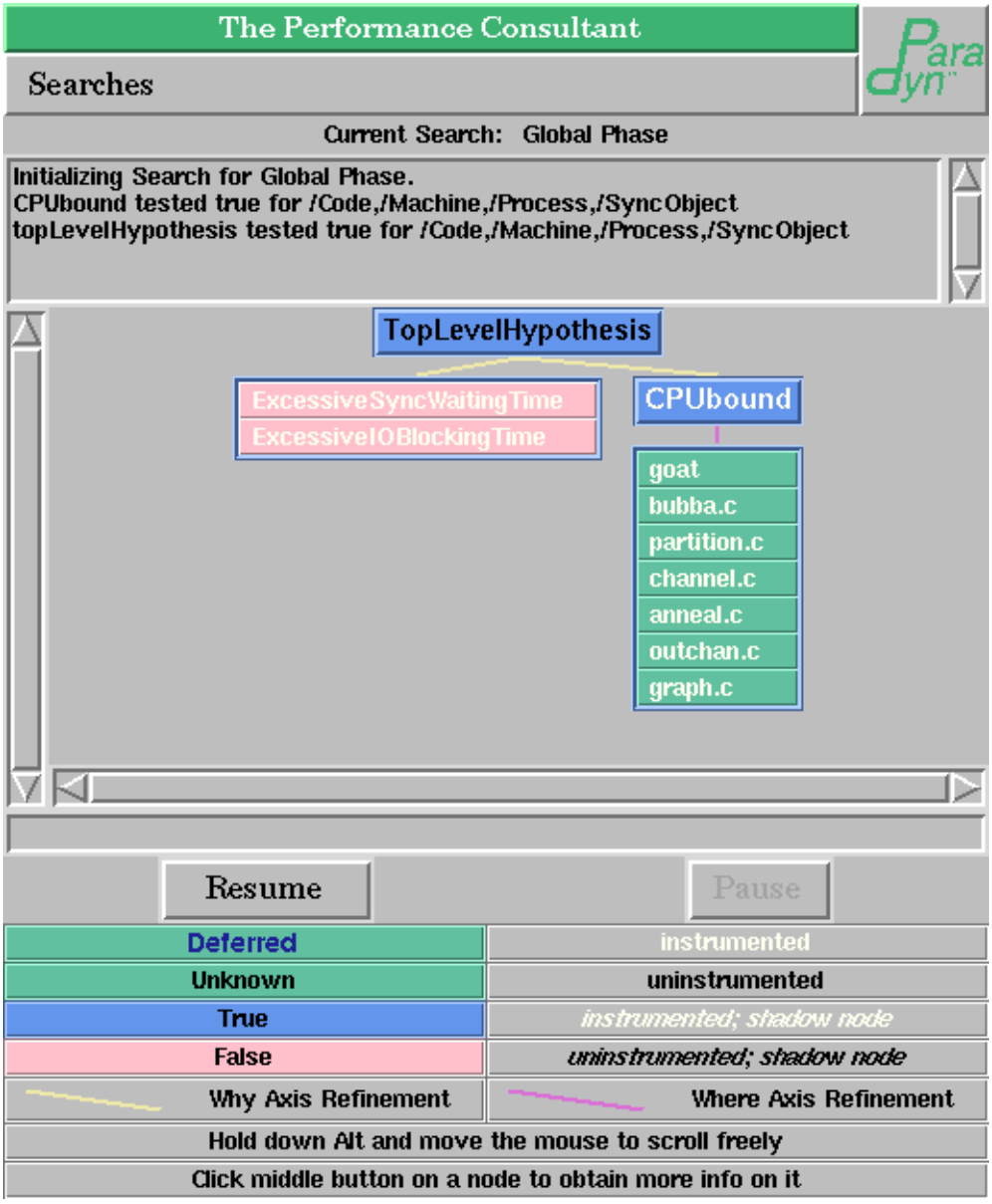


Figure 33: The Performance Consultant refines bottleneck to CPUbound

the search is directed by executing functions, the callgraph-based search is typically faster and more accurate than the module-then-function search.

9.4 Interpreting the results of module-then-function search

Results may change over time because the Performance Consultant continues running for the duration of the phase being tuned. Figure 35 shows a search in progress and explains how to interpret the PC display. The Performance Consultant has decided that the program is CPU bound (and it represents this by presenting CPUbound drawn with a blue background). The nodes (ExcessiveSyncWaitingTime : WholeProgram) and (ExcessiveIOBlockingTime : WholeProgram) have both

Visual representation	Tunable Constant to control display look	Description
1. Gray node background	**shgHideNeverSeenNodes**	Nodes that the Performance Consultant has not yet examined.
2. Green node background	**shgHideUnknownNodes**	Nodes that the Performance Consultant has not yet determined to be true or false.
3. Blue node background	**shgHideTrueNodes**	Nodes that the Performance Consultant has determined to be true.
4. Pink node background	**shgHideFalseNodes**	Nodes that the Performance Consultant has determined to be false.
5. White node text	**shgHideActiveNodes**	Nodes with white text are those that are active—the Performance Consultant has instrumented the program and is collecting performance data for it.
6. Black node text	**shgHideInactiveNodes**	Nodes with black text are inactive—the Performance Consultant has not instrumented the program to collect performance data for it.
7. Red node text		Nodes with red text are deferred—the daemon has communicated to the performance consultant that the metric-focus request is delayed. The delay can vary depending on the function and the program.
8. Italicized node text	**shgHideShadowNodes**	Nodes with italicized text are shadow nodes; they are discussed in Section 9.4.

Figure 34: Search History Graph tunable constants for saving screen space

tested false, so their background color is now pink. Although all three of these nodes remain active, only the true node, CPUbound, has been expanded to try to further refine the bottleneck. Each item in the listbox under CPUbound contains program resources that have been tested as possible refinements of the CPUbound hypothesis. Refinements to two different true nodes (machine name goat and source code module partition.c) have been made. The Performance Consultant is capable of making an arbitrary number of simultaneous refinements, because multiple hypothesis/focus pairs may be tested concurrently. For example, in the Search History Graph of Figure 35, the Performance Consultant will try to make refinements of the two true nodes below CPUbound: goat and partition.c.

Two separate search paths may converge through expansion to the same child node. For example, the next refinement of goat might be partition.c, and the next refinement of partition.c might be goat. If so, they would share the same child node: (CPUbound : /Code/partition.c,/Machine/goat,/Process,/SyncObject). The search display does not connect the two different parent nodes to the same child; instead, it adds a child node for each, where one is a regular node and the other(s) is a copy. These copies are called *shadow nodes*. In Figure 35, the regular node goat has been clicked with the middle mouse button to provide its details in the information line below the shg, while the listbox item goat under partition.c is drawn in italics to indicate that it is a shadow node. The color of a shadow node will be updated to reflect the status of its regular node. Shadow nodes are always leaf nodes; although the regular node may be expanded in the usual way, the resulting listbox is not be copied

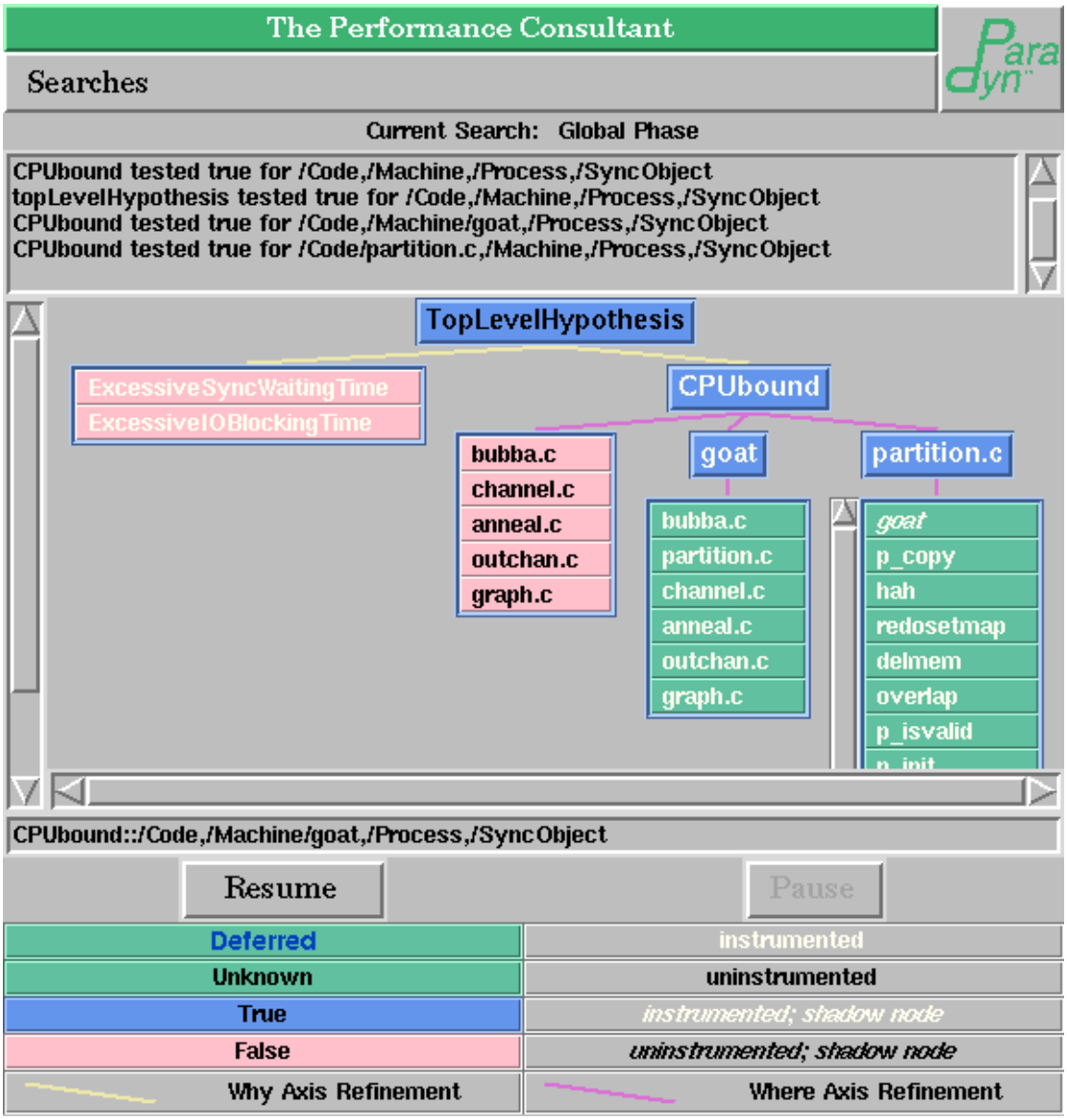


Figure 35: The Performance Consultant refines bottleneck beyond CPUbound

to the shadow nodes. In this example, the node goat under partition.c is a shadow node because it has the same hypothesis/focus pair (CPUbound : /Code/partition.c,/Machine/goat,/Process,/SyncObject) as the listbox item partition.c under goat.

Figure 36 shows the contents of the Search History Graph after the next set of refinements are made. First, the node partition.c under goat has been found true; this is the hypothesis/focus pair CPUbound : /Code/partition.c,/Machine/goat,/Process,/SyncObject discussed above. This focus can be read as, "code in module partition.c when executing on machine goat". The shadow node goat under partition.c is also true; no attempt is made to refine anything beyond it, however, because it's just a shadow node of partition.c under goat. Additionally, the node p_makeMG under partition.c is now true. Its hypothesis/focus pair is (CPUbound : /Code/partition.c/p_makeMG,/Machine,/Process,/SyncOb-

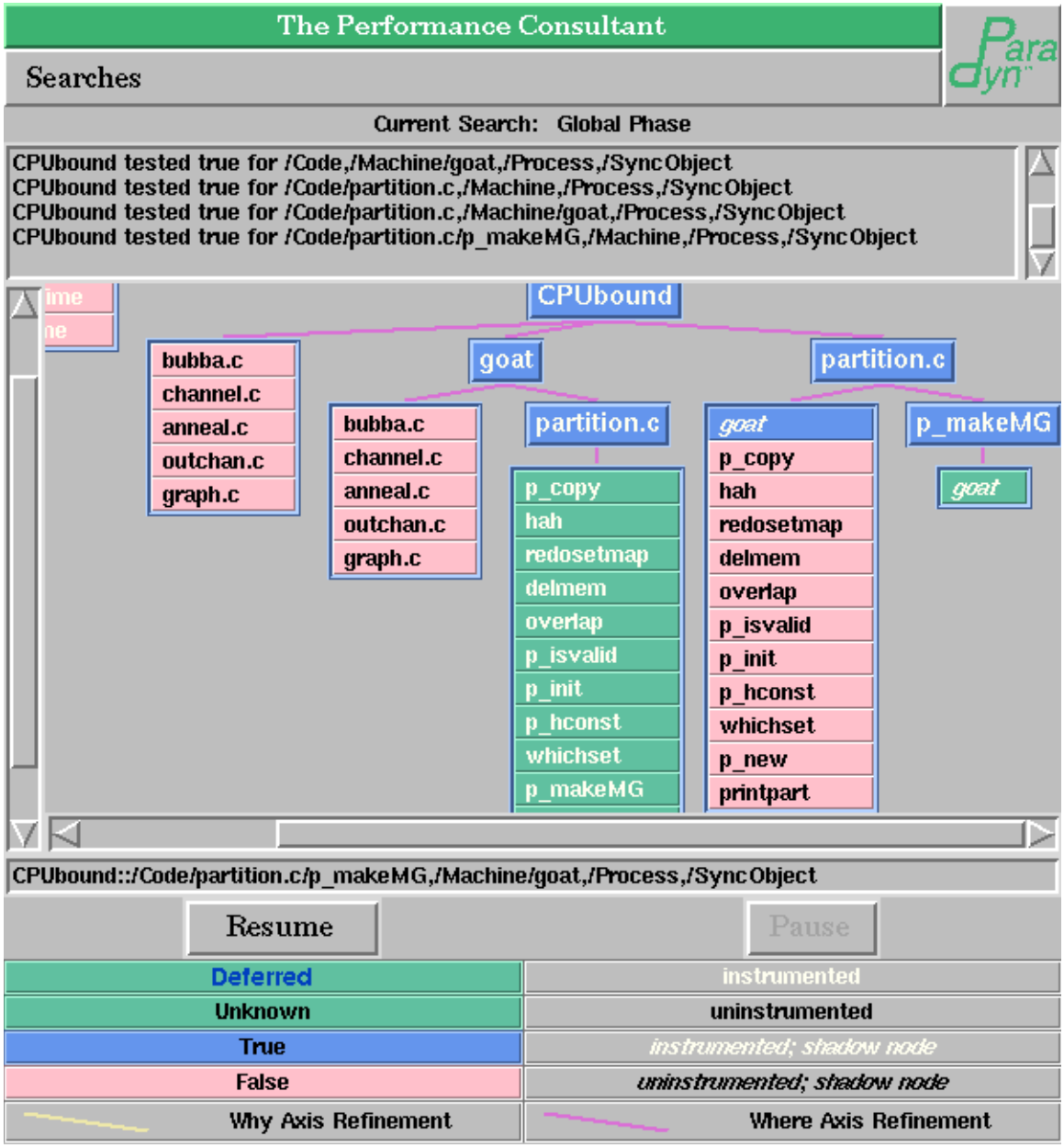


Figure 36: The second set of Search History Graph refinements

ject). Note that p_makeMG has just a single element in the listbox below it (goat), and it's a shadow node. The hypothesis/focus pair for this node is shown in Figure 36 (i.e., we have clicked the middle button on that node). It can be read as "function p_makeMG of module partition.c; machine goat". This item is a shadow node of p_makeMG, located in the listbox below partition.c which is in turn under goat. Hence, in Figure 36, two searches are in progress. The first has tentatively concluded that a bottleneck exists for module partition.c on machine goat. The other has tentatively concluded that a bottleneck exists for the function p_makeMG (of module partition.c), and is trying to refine further.

The state of the Performance Consultant after the next (and last) refinement is shown in Figure 37. In the middle of the figure, we see that p_makeMG (under partition.c, in turn under goat) is

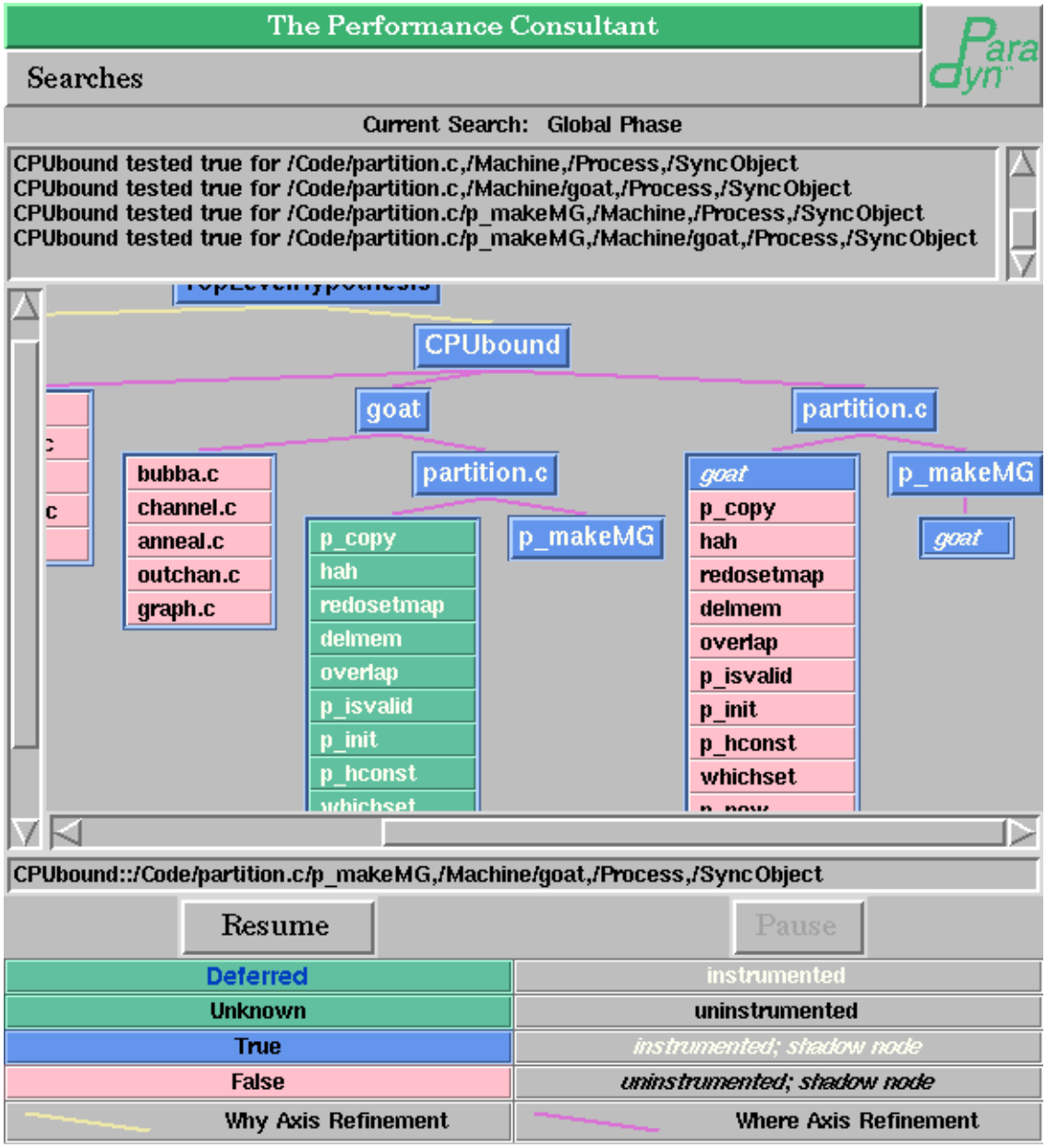


Figure 37: Final Search History Graph bottleneck refinement

true. Its hypothesis/focus pair is shown below the Search History Graph (i.e., we have clicked the middle button on the node). It can be read "function p_makeMG of module partition.c; machine goat". In addition, the shadow node goat under p_makeMG (in turn under partition.c) has been set to true, to reflect the change in truth value of the actual node for which it is a marker.

In this example, we are done. The Performance Consultant has found the bottleneck, and will not refine any further nodes. After a few more moments, the green items (unknown) in the listbox

below partition.c will turn pink (false); though we do not show a picture of it here. The Performance Consultant will continue to re-evaluate all true nodes and top level hypotheses so that a change in application behavior will update the search.

All nodes which are true (blue) at the end of the search indicate hypothesis/focus pairs that have remained true for a significant portion of the phase searched. The PC refines all true nodes to as specific a focus as possible; in some cases the focus will be refined down to the leaf level of the resource hierarchies, but in others the bottleneck is spread across some number of foci and so refinement stops earlier. For example, total CPU time for a module may exceed the current **PC_CPUThreshold**, but the module may contain a number of functions with roughly equal CPU times. If no single function exceeds the threshold, refinement will terminate at the module level.

Whenever a node tests true, a note is added to the **Search Status Box** near the top of the window. During your tuning session you may scroll through this list to see a history of test results.

Note that it is possible for the Performance Consultant to report false negatives: that is, it may fail to detect bottlenecks in the code for any of the following reasons: you start a search after the behavior has started and ended; you perform a search on a phase that contains several distinct behavioral phases, so no individual bottleneck occurs throughout the entire phase; or the bottleneck is of relatively short duration, relative to the length of the phase being tested. The PC may fail to completely refine a given bottleneck, if the individual refinement changes from false to true after the PC has tested and found it false.

9.5 Customizing the search parameters

The Performance Consultant has several kinds of controls that you can set to customize its search operation. These controls are tunable constants that set the threshold for deciding when a performance problems exists. Setting the tunable constants is easily done following the instructions in Section 4.

Several user-level tunable constants are currently defined to control the search: **PC_CPUThreshold**, **PC_SyncThreshold**, **PC_IOThreshold**, and **PC_IOOpThreshold**. For example, if **PC_CPUThreshold** is set to 0.3 (30% of the phase), then any focus with CPU time greater than 30% of the phase's elapsed time will be reported as a bottleneck. Other tunable constants control the sensitivity of the hypothesis testing.

The tunable constants determine the thresholds used for testing hypotheses:

PC_CPUThreshold: used for hypothesis *CPUbound*.

PC_SyncThreshold: used for hypotheses *ExcessiveSyncWaitingTime*.

PC_IOThreshold: used for hypothesis *ExcessiveIOBlockingTime*.

PC_IOOpThreshold: used for hypothesis *TooSmallIOOps*.

These tunable constants determine search parameters:

minObservationTime: all tests will be continued for at least this interval of time before any conclu-

sions are drawn. This protects against transitory effects at the start of a phase.

costLimit: determines an upper bound on the total amount of instrumentation that can be active at a given time while the application runs. A low value permits less concurrent instrumentation, so the search may proceed more slowly but perturbation of the application will also be lower. A high value increases perturbation, which may result in less accurate values for all visualizations as well as the Performance Consultant.

10 STANDARD VISI MODULES

Paradyn provides an open interface to its performance data. All visualization modules (*visis*) in Paradyn are external processes that use the Paradyn-provided library and remote procedure call interface (*VisiLib*) to access performance data in real time. Existing visualizations can be easily added to Paradyn by modifying them to use VisiLib routines to access Paradyn performance data. Paradyn currently has visis for a time-histogram, bar chart, table and 3D terrain visualization. These visualizations are described in the following sections, and the VisiLib library is described in a separate document, the ***VisiLib Programmer's Guide***.

10.1 Time Histogram visi

The time-histogram visualization plots performance data for metric/focus pairs over time. Figure 38 displays a time-histogram showing three curves corresponding to three enabled metric/focus pairs. The time axis begins at the start time for the phase over which the data is being displayed (in this case the data is displayed for the global phase which begins at time 0). The time-histogram can display multiple *y*-axes. In Figure 38 there are two *y*-axes displayed; the rightmost one corresponding to the metric "CPU utilization", and the leftmost corresponding to the metric "Procedure Calls". Each *y*-axis is labeled with the units in which its corresponding metric is measured. The *y*-axis labels can be seen in Figure 38.

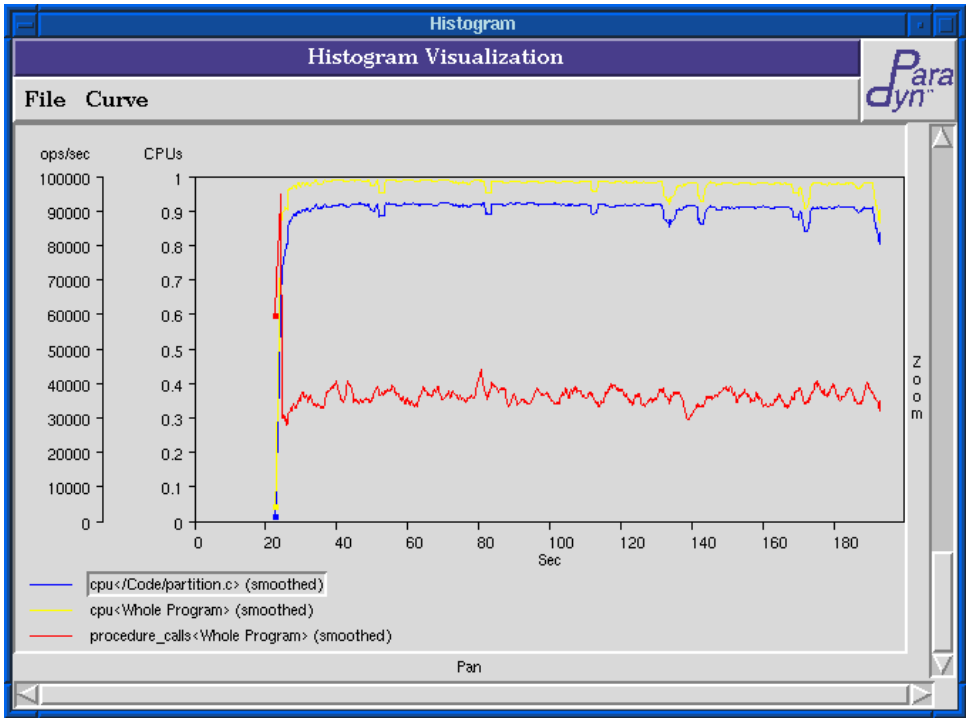


Figure 38: Time Histogram with selected curve

Time-histogram is launched by choosing Histogram from the **Start A Visualization** dialog produced by pressing the **Visi** button in the Paradyn Main Control window. A dialog box with a list of all visis known to Paradyn is brought up; choose Histogram and click **Accept**.

10.1.1 File menu

The Histogram visi's File menu contains two items for operations that apply to the Histogram visualization as a whole. The **Keep on Paradyn Exit** item is a check box that specifies that the Histogram display should not terminate when Paradyn is closed. This behavior is useful in case the user wishes to make screen dumps of the histogram display after the application under study has completed and Paradyn has been closed. By default, this behavior is disabled (i.e., the Histogram window closes when Paradyn is closed).

The **Close** item closes the Histogram visi window, disabling instrumentation and removing resources no longer required.

10.1.2 Curve menu

The Curve Menu contains items that manipulate curves in the Histogram visi window. Most of the items operate on the selected curve(s). To select a curve, click on its name in the legend of the Histogram visualization window. See Figure 38 for an example of a selected curve.

The **Add...** menu item adds curves to the Histogram visualization. When selected, the "Paradyn Metrics Menu" dialog will be displayed to allow the user to choose which metrics are to be shown in the Histogram. The Where Axis is also used in adding a new curve, in that the current selection in the Where Axis is the focus for the new curve. The **Add...** menu item is the only item in the Curve menu that does not require a current selection.

The **Remove** menu item removes one or more curves from the Histogram visualization. When this item is chosen, the curves selected in the Histogram visualization window's legend are removed from the Histogram visualization. If possible, the instrumentation that was used to collect the data for that curve is removed from the application under study.

The **Smooth** menu item smoothes the data for the selected curve(s). A smoothed curve is one that shows the effects of passing a filter over the data to remove spikes from the curve's data. The curves shown in Figure 38 are smoothed. By default, curves are smoothed when added to the Histogram.

The **Unsmooth** menu item removed the smoothing effect from the selected curve(s). For comparison with the smoothed curves of Figure 38, see the same curves unsmoothed in Figure 39.

The **Show** menu item reshows a hidden curve in the Histogram visualization window. Conversely, the **Hide** menu item hides a visible curve in the display. Both operations work on the selected curve(s) only. In Figure 39, the "cpu<Whole Program>" curve has been hidden in the display..

10.1.3 Panning and zooming

The scroll bars at the bottom and right of the time-histogram allow the interval of time displayed in the histogram window to be adjusted. The zoom bar can be adjusted to get a more detailed view of a particular time interval along the *x*-axis. As the zoom bar is moved upwards, the percent of the total *x*-axis displayed decreases. Also, once the zoom bar has been moved, the pan bar can be used to change the time interval that is currently being displayed in the window. Figure 40 shows the time-histogram with a zoomed and panned view.

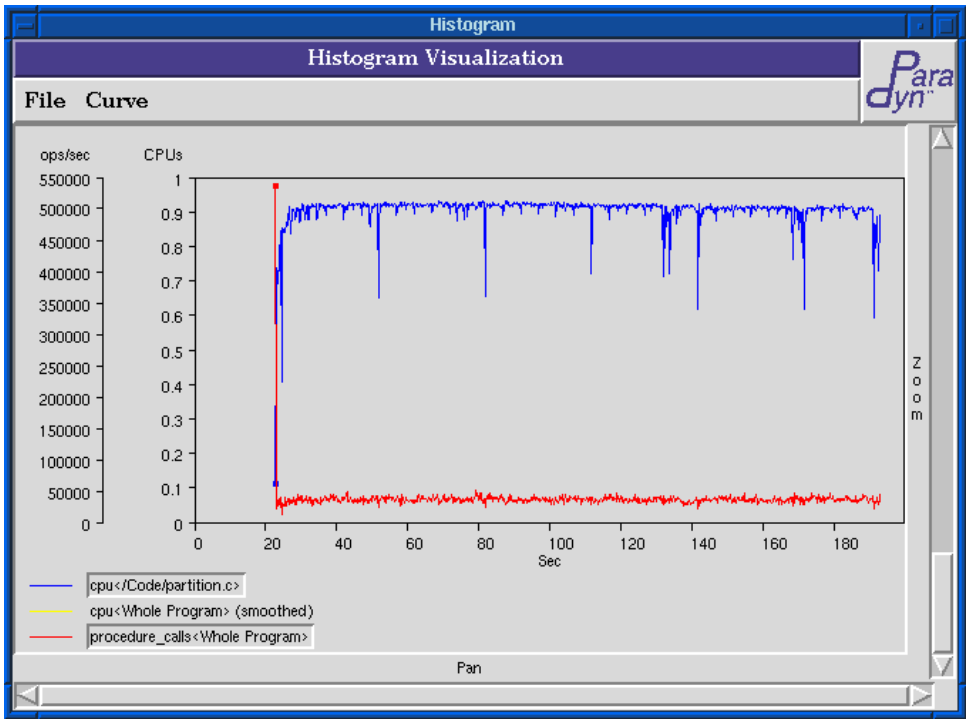


Figure 39: Time Histogram with unsmoothed and hidden curves

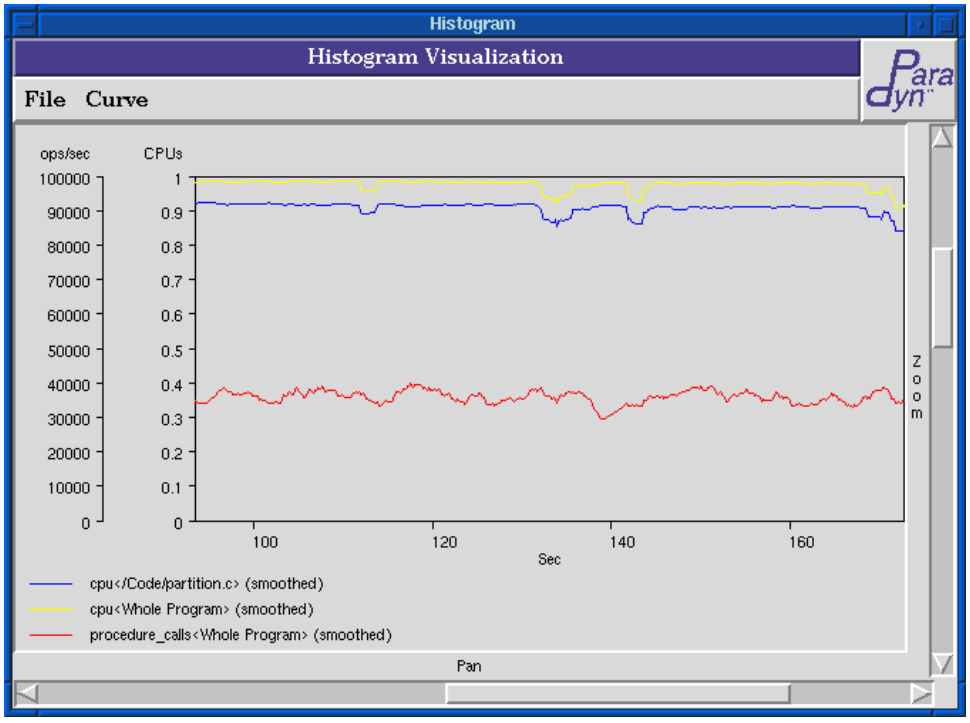


Figure 40: Zoomed Time Histogram

10.2 Barchart visi

Barchart is an external visualization module that enables many metric-focus pairs to be viewed in real time. Barchart receives its data through the *visi lib*. The visi lib is described in the ***VisiLib Programmer's Guide***; we do not discuss it further here.

Figure 41 shows the Barchart window. The vertical axis contains the names of all foci selected for viewing. There are also a certain number of metrics currently selected for viewing; they (along with a range of values) are displayed in the horizontal axis. Note that each metric has its own color; this helps identify the bars emanating horizontally next to each focus.

Barchart is designed to view many metric/focus pairs. In Figure 41, there are seven foci and two metrics, leading to $7 \times 2 = 14$ metric/focus pairs. Barchart can easily handle far more; it is not unusual to display 30 or more foci, and five or more metrics. This contrasts with the Histogram visi (see Section 10.1), which is restricted to eleven metric/focus pairs at a time. On the other hand, Barchart has no way to show how metric/focus pairs change over time.

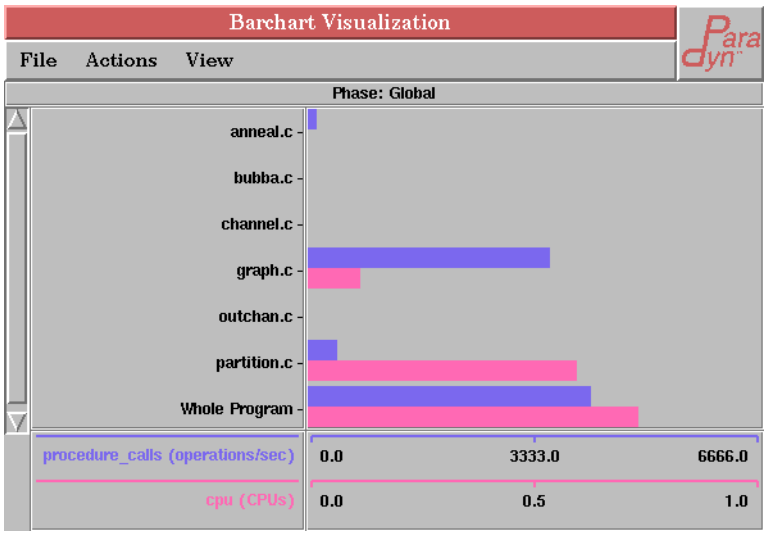


Figure 41: Barchart visualization window

Barchart is launched by choosing it from the **Start A Visualization** dialog produced by pressing the **Visi** button in the Paradyn Main Control window. A dialog box with a list of all visis known to Paradyn is brought up; choose Barchart and click **Accept**.

A dialog box containing all metrics known to Paradyn will appear. Paradyn is asking you to select some initial metric-focus pair(s) for the Barchart. Choose metric(s) by selecting desired checkboxes in the metrics dialog box[1]. Choose foci by selecting desired resources in the Where Axis window.[2] The metric-focus pairs generated will be the cross-product of the foci and metrics.

1. For details on selecting metrics, refer to Section 6.
2. For details of focus selection, and the Where Axis in general, refer to Section 5.

At this point, the Barchart window (as in Figure 41) should appear, with the metrics and foci you selected. If Paradyn is running an application, data should begin appearing immediately.

10.2.1 Changing metrics and foci being viewed

You must specify an initial metric/focus set when launching a barchart. You may later add as many more metric/focus pairs as desired (duplicates will be correctly filtered). To do this, choose **Add Bars** from the Barchart's **Actions** menu. The interface for adding metrics and foci at this point is the same as upon startup; you will be shown the metrics dialog box for choosing metrics, and the where axis for choosing foci.

You may delete foci by clicking on their names and choosing **Delete Selected Foci** from the **Actions** menu.

10.2.2 Viewing data

Values being viewed in a Barchart are, by default, current. Each time a screenful of new data arrives (from Paradyn), Barchart immediately displays the most recent values, thereby overwriting the previous screenful of data, which cannot be re-displayed later[3].

There are two other ways of viewing data. Under the **View** menu, we could choose to view **Average** values. In this case, what we see on the screen will be the average (over time) of all values collected by this instantiation of Barchart. After a short time, the bars will probably setting down to a steady state; this is to be expected when viewing average values. The third way of viewing data is to view **Total values**. This causes the bar values to monotonically increase over time. Figure 42 shows a Bar-

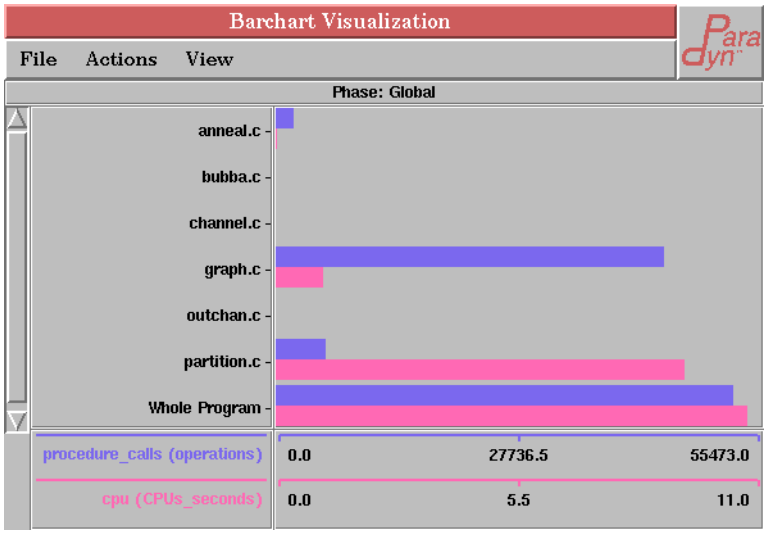


Figure 42: Barchart showing total values

chart, otherwise similar to that of Figure 41, with **Total** instead of **Current** values displayed. Note

3. To get a feeling for metric/focus pair changes over time, try the run-time Histogram visi (Section 10.1).

that the metric units (the lower left corner of Figure 42) changes accordingly, and that the metric bounds (the lower right part of Figure 42) adjust accordingly.

10.3 Table visi

Like the Time Histogram (Section 10.1) and Barchart (Section 10.2), Table is a Paradyn visualization module (visi) that receives its data through the *visi lib* (described in the document ***VisiLib Programmer's Guide***) interface.

Figure 43 shows the Table window. The columns are metrics; the rows are foci. Note that there

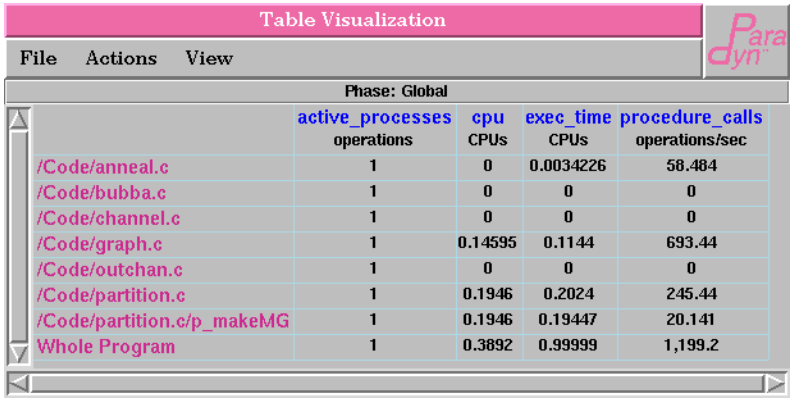


Figure 43: Table visualization window

are two lines describing each metric: the first name (in blue) is the metric name; the line below it (in black) gives the metric's units.

Like Barchart, Table uses screen real estate efficiently—it can show many metric-focus pairs at a time. For example, Figure 43 has four metrics and eight foci for a total of $4 \times 8 = 32$ metric-focus pairs. It is reasonable for a Table to show hundreds of metric/focus pairs at a time. However, like Barchart, Table cannot show how metric/focus pair values are changing over time.

Table is launched from the **Start A Visualization** dialog resulting from pressing the **Visi** button in the Paradyn Main Control window menubar. A dialog with a list of all visis known to Paradyn is brought up; choose **Table** and click **Accept**.

A dialog box containing all metrics known to Paradyn will appear (see Section 6). Paradyn is asking you to select some initial metric-focus pair(s) for the Table. Choose metric(s) by selecting desired checkboxes in the metrics dialog box. Choose foci by selecting desired resources in the **Where Axis** window (see Section 5). The metric-focus pairs generated will be the cross-product of the foci and metrics.

At this point, the Table window (as in Figure 43) should appear, with the metrics and foci you specified. If the application is running, data should begin appearing immediately.

10.3.1 Actions menu

Launching Table requires an initial metric/focus set to be specified. However, you may later add or delete metric-focus pairs as desired (when adding, duplicate pairs will be correctly filtered). To add metric-focus pairs, choose **Add Entries** from Table's **Actions** menu. The interface for adding metric/focus pairs is the same as when starting Table (Section 10.3); choose entries from the metrics dialog box and the Where Axis window.

Deletion in Table can take 3 forms; you can delete a focus (an entire row of the table), a metric (an entire column of the table), or a single metric-focus pair (a single cell of the table) with one delete operation. First you select what to delete by clicking once with the left mouse button on the appropriate item. To delete a focus, click on the focus name itself on the left side of the table; the entire row will become highlighted. To delete a metric, click on the metric name itself at the top of the table; the entire column will become highlighted. To delete an individual metric/focus pair, click on the cell value; it will become highlighted. Once you have selected an item, the second entry of the **Actions** menu (named **Delete Selected Focus (entire row)**, **Delete Selected Metric (entire column)**, or **Delete Selected Cell**, as appropriate) will become active. Choose that menu item to perform the deletion.

10.3.2 View menu

Long vs. short names

Focus names can be displayed in long form (e.g., `/Code/anneal.c`) or in short form (e.g., `anneal.c`). To toggle between the long and short forms, choose **Long Names** from Table's **View** menu. The default is to show long names. Figure 44 shows the equivalent of Figure 43 with short instead of long names..

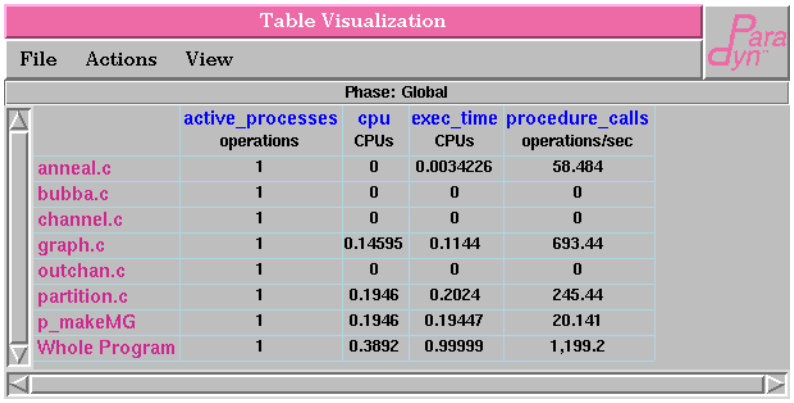


Figure 44: Table visualization showing short focus names

Current vs. average vs. total values

By default, Table cells are "current": As soon as a screenful of new data arrives from Paradyn,

Table redraws the cells with the new values. As with Barchart, there are two other ways to view data. Under the **View** menu, we could choose to view **Average** values. In that case, metric/focus pair values shown will be the average (over time) of all values collected by this instantiation of **Table**. After a short time, the values shown will probably settle down to a steady state; this is to be expected when viewing averages. The third way of viewing data is **Total values**. This causes the bar values to monotonically increase over time.

Sorting metrics

By default, Table displays the columns (metrics) in the order in which they were added. To sort them by name, choose **Sort Metrics (ascending)** from Table's **View** menu. To change back to the default, choose **Don't Sort Metrics** from Table's **View** menu.

Sorting foci

By default, Table displays the rows (foci) in the order in which they were added. To sort by name, choose **Sort Foci (ascending)** from Table's View menu. Note that sorting foci is sensitive to the current setting of **Long Names** in the **View** menu: if long focus names are displayed, sorting is according to these long names; if short focus names are displayed, sorting is according to the short names.

There is another way to sort foci: by value. Choosing **Sort Foci By Values (of Selected Metric)** effectively turns Table into a profiler; whenever a screenful of new data arrives from Paradyn, the foci (rows) are reordered to match the new values. When viewing **Current Values**, rows can seem to jump around so quickly that they are difficult to read. Sorting foci by value clearly works best when viewing **Average Values** or **Total Values**, which reach a steady state quickly.

In order to sort foci by value, Table needs to know which metric to sort by. To give an example, sorting the foci of the table in Figure 44 would yield very different orderings between the foci **procedure_calls** and **cpu**. In the former, **graph.c** has a higher value than **partition.c**; not so the latter.

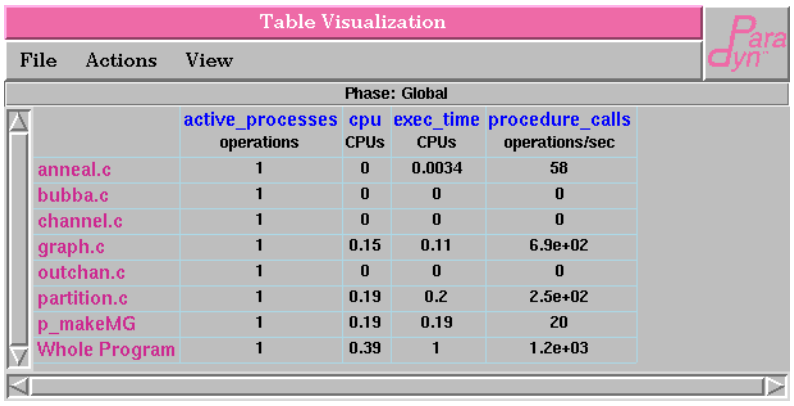
Significant digits

Individual metric/focus pairs are floating point values. You can change the number of significant digits in which these values are viewed by choosing the desired item under the **View** menu. Figure 44 is shown to five significant digits. Figure 45 shows the same table with two significant digits. Scientific notation is used when necessary.

10.4 3D Terrain visi

Like all the previous visis, Terrain is a Paradyn visualization module (visi) that receives its data through the visi lib (described in the document ***VisiLib Programmer's Guide***) interface.[4] The Terrain visualization displays data in 3D, allowing the performance data to be analyzed using a *surface* rather than curves or bars. This visualization can be particularly useful when we want to compare

4. Note that the Terrain visi is not supported on Microsoft Windows.



	active_processes operations	cpu CPUs	exec_time CPUs	procedure_calls operations/sec
anneal.c	1	0	0.0034	58
bubba.c	1	0	0	0
channel.c	1	0	0	0
graph.c	1	0.15	0.11	6.9e+02
outchan.c	1	0	0	0
partition.c	1	0.19	0.2	2.5e+02
p_makeMG	1	0.19	0.19	20
Whole Program	1	0.39	1	1.2e+03

Figure 45: Table visualization with values shown to two significant digits

a particular metric for different foci (like the example shown below in Figure 46, which displays CPU time for machines “beaufort”, “cham” and “poona”).

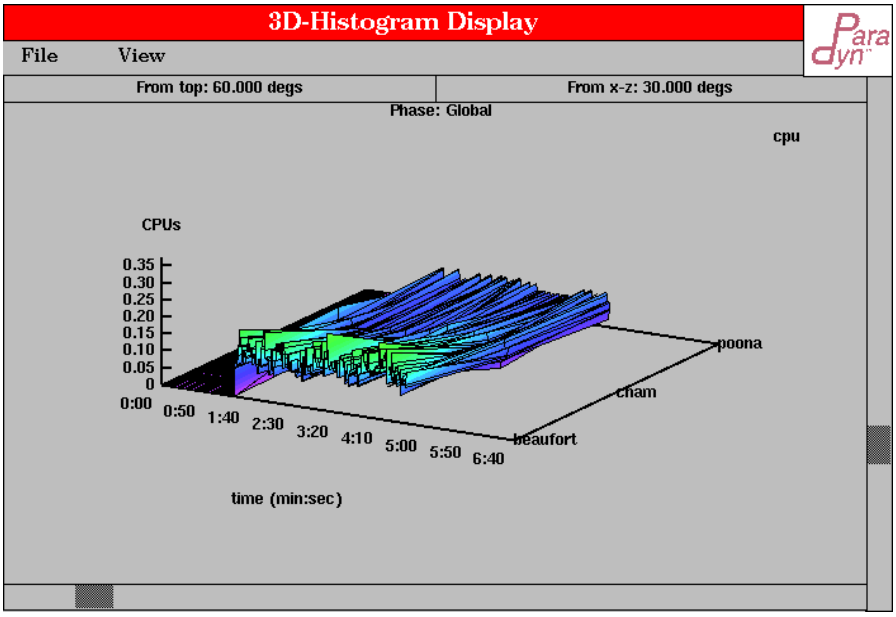


Figure 46: 3D Terrain visualization

10.5 Viewing the application output display

A special graphical window, called the termWin, is created whenever Paradyn starts an application. This window displays the output of the application. The output information displayed in this window can be edited and saved to a file. In developer mode, debug information from Paradyn

daemon is also displayed in this window; a different color is used to identify the debugging output. An example is shown in Figure 46.

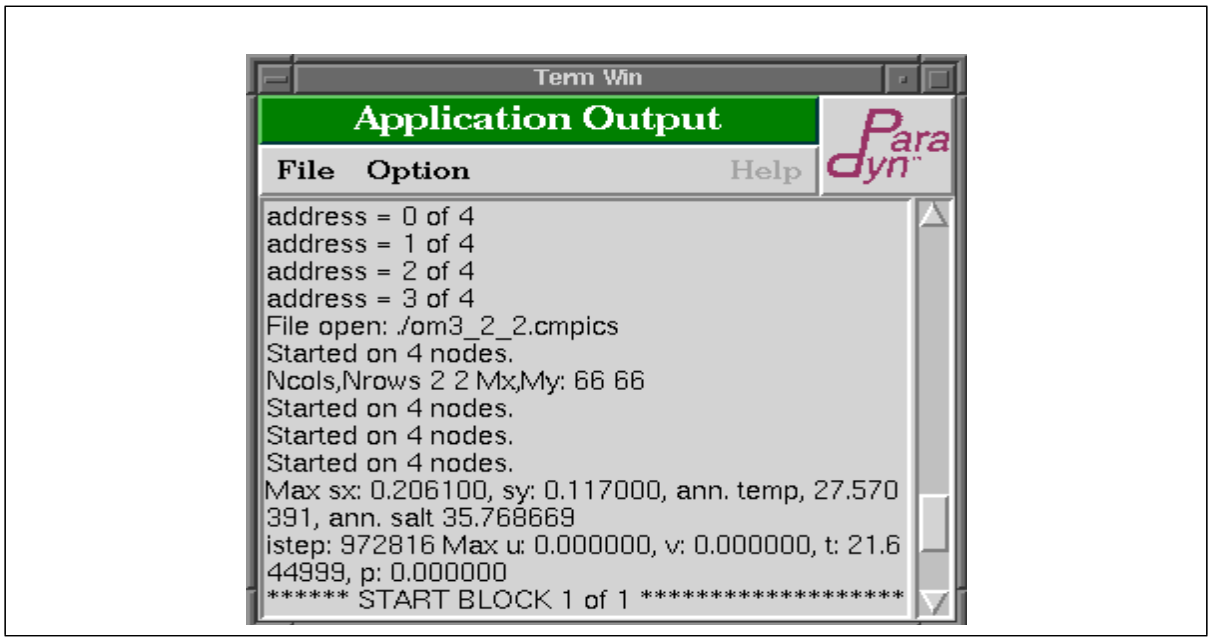


Figure 47: Application output window

There are two commands available: the File|Save menu option, which creates a standard save dialog for the application output; the Option|Close menu option, which controls when the window can be closed. By default it is closed when Paradyn exits. Alternatively, it will remain even after Paradyn exits, so that the user can continue working on it.

There currently is no termWin visi for Paradyn on the Windows platform. On this platform, application output is directed to a terminal window.

11 EXPORTING PARADYN DATA

Paradyn profiling, resource, and search data may be saved for external analysis or examination by pressing the **EXPORT** button from the Paradyn main control window. This opens the **EXPORT** dialog, presenting a variety of options for writing Paradyn data to files as shown in Figure 48. You

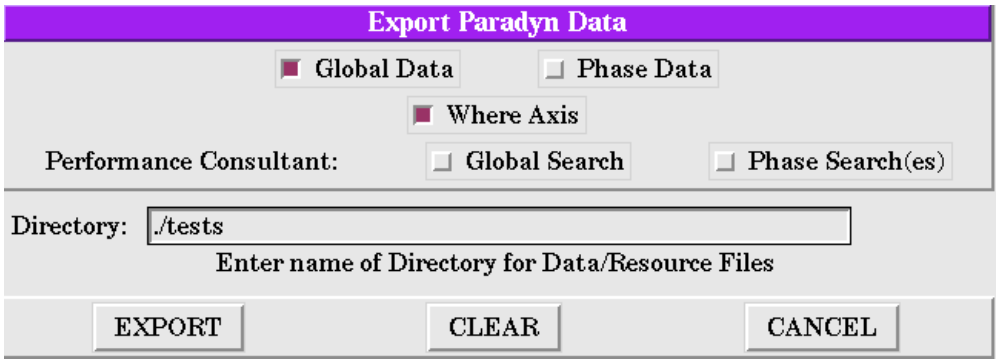


Figure 48: Export dialog window

may select one or more types of data to be saved: raw Paradyn performance data, the Paradyn Where Axis, and/or Performance Consultant Search Data. Depending on your selections, one or more files will be written into the directory you specify in the dialog window. Next we describe each specific option in detail.

11.1 Saving Performance Data

Paradyn allows one to save the performance data that is collected during monitoring of an application.

11.1.1 Saving Performance Data From Front-End

To save raw performance data select Global Data or Phase Data in the Export dialog. Global or Phase correspond to the phase Paradyn was in when the data was collected, which is user specified, so if you didn't start new phases during your Paradyn session there will be only data for the global phase. Type in the name of the directory to which you want the data written. Then select "EXPORT."

Two types of files are written into the specified directory. An index file ("index") is generated, with one line per data file written, of the form: "*filename metric focus*". Then, each existing internal Paradyn histogram is written to a separate data file into the same directory. If the tunable constant "persistentData" was set during the Paradyn session, there is one Paradyn histogram for each metric/focus pair instrumented at any time during the current execution. Otherwise, there is one Paradyn histogram for each metric/focus pair instrumented at the time the Export button was pressed. The files are named hist_0, hist_1, ... hist_n where n is the total number of Paradyn histograms.

Each data file starts with a 3-line header containing the following:

- Line 1: *dataType metric*
- Line 2: *focus*
- Line 3: *maxNumBins binSize.*

At present, *dataType* is always "histogram" for exported Paradyn data. The *metric* is the particular measurement instrumented, such as CPU time (metric "cpu") or I/O waiting time (metric "io_wait"). See Section 6.2 for a description of metrics available in Paradyn. The *focus* is the particular part of the program run measured, which might be the whole program or perhaps one function or one process out of many in the whole program.

The header is followed by a list of up to *maxNumBins* values, one per line. The header and first few data values of a data file is shown in Figure 49. To calculate the start and end time for each measured value, use the *binSize*: each data value represents one *binSize* unit of time in seconds, starting at time 0. In the example shown, the bin size is 0.2 seconds, so the first value was measured for

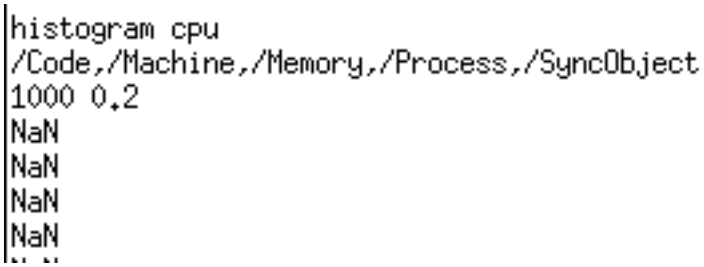


Figure 49: Performance Data File Header and Data Format

time 0 to time 0.2, second value was measured for time interval 0.2 to 0.4, etc. Note that "NaN" indicates that no value was measured for a particular time interval. Since Paradyn instrumentation may be inserted and deleted freely during execution, the common case is to have measured values for only a subset of the total number of data bins. Missing values at the start and middle of the time period are indicated with a "NaN" entry. The total number of bins that can be stored for the Paradyn histogram, *maxNumBins*, is indicated in the third row of the Data File header. There may be fewer than *maxNumBins* total data values written in the file, since trailing NaN's are not written. However, the maximum number of lines in the data file is 3 (for the header) plus *maxNumBins* (one line per data value).

After all data files and the index file are successfully written, the UIM status bar is updated as shown in Figure 50.

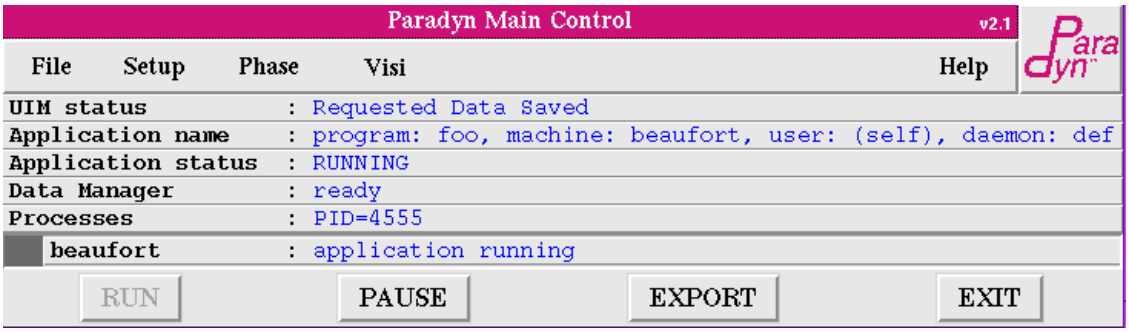


Figure 50: The Paradyn Main Window after saving performance data.

11.1.2 Saving Performance Data From the Visis

The previous sub-section describes how to do a "bulk" save of all the performance data being collected at the Paradyn front-end. This section discusses how a user may selectively choose the histogram data he wishes to save.

For each of the following visis, Time Histogram, Barchart and Table, a **Save** function is available from the **FILE** menu. When selected, a dialog window like the one shown in Figure 51 pops up. This dialogue presents a list of all the valid Metric-Focus pairs that the current visi is subscribed to in the format <Metric:Focus>. The user chooses the Metric-Focus pairs he is interested in and clicks the *Export* button. A standard TCL dialogue box now pops up allowing the user to select a directory in which to save the performance data files. Clicking the **OK** button on this dialogue instantiates an exportation of the selected performance data to the chosen directory. The format of the files is similar to those obtained when doing the bulk data exportation described previously. Note that saving to an existing directory overwrites any previously stored performance data.

Figure 51: Visi Menu to Export Paradyn Data

11.2 Saving the Where Axis

Generate a list of all valid resource nodes for the application session by selecting “Where Axis” from the Export Menu. The resulting file (named “resources”) is a simple text file with one resource name per line. The complete Where Axis may be reconstructed from this file. After the resources are successfully written, the UIM status bar is updated as shown in Figure 52.

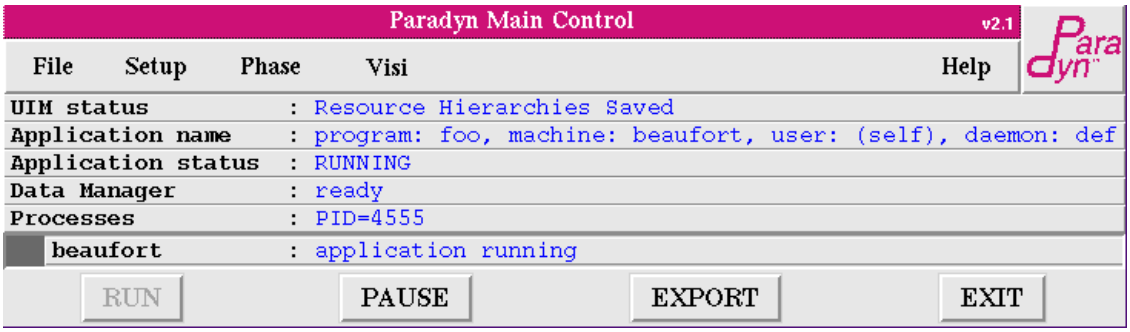


Figure 52: The Paradyn Main Window after saving resource names.

11.3 Saving Performance Consultant Search Data

Save the contents of one or more Search History Graphs by selecting Global Search or Phase Search(es) from the Export Menu. One file (“shg.txt”) is generated that lists the contents of all Search History Graphs in the format shown in Figure 53. Note that “#” is used as a comment character in the file. There is one Search History Graph for each phase that was diagnosed in the Paradyn session. If Phase Search(es) was selected, this file will contain data for one Search History Graph for each phase-level search either in progress or completed. A banner of the form “# *searchNum* Performance Consultant Search History Graph” starts each set of search data; the example shown lists search 0, which is the global search. A banner of the form “*** search history node ***” starts each search history node listing. Each node listing has the following format:

- Row 1: *nodeID*
- Row 2: *hypothesisName*::*Focus*
- Row 3: *experiment defined?*
- Row 4: *experimentStartTime experimentEndTime currentValue currentThreshold.*

NodeID is a unique integer id assigned to each node of a single Search History Graph. *HypothesisName* is the Performance Consultant test name as described in Chapter 9, and *focus* is the particular part of the program run that was measured, e.g. function foo or process 1. *ExperimentDefined?* is 0 if this search node’s experiment was never activated, for example, if the search was halted before the instrumentation was inserted; otherwise its 1. *ExperimentStartTime* and *experimentEndTime* are the timestamps on the first and last piece of data received for the

experiment. *currentValue* is the calculated value, averaged over the interval, of the hypothesis, and *currentThreshold* is the value *currentValue* was compared against to determine a result of true or false.

After all Performance Consultant data is successfully saved, the UIM status bar is updated as shown in Figure 54.

```
# 0 Performance Consultant Search History Graph
*** search history node ***
0  #nodeID
topLevelHypothesis:/Code,/Machine,/Memory,/Process,/SyncObject
0  # experiment defined?
*** search history node ***
1  #nodeID
ExcessiveSyncWaitingTime:/Code,/Machine,/Memory,/Process,/SyncObject
1  # experiment defined?
-1 0 0 0 # experiment start end value threshold
```

Figure 53: Format for exported search data in shg.txt.

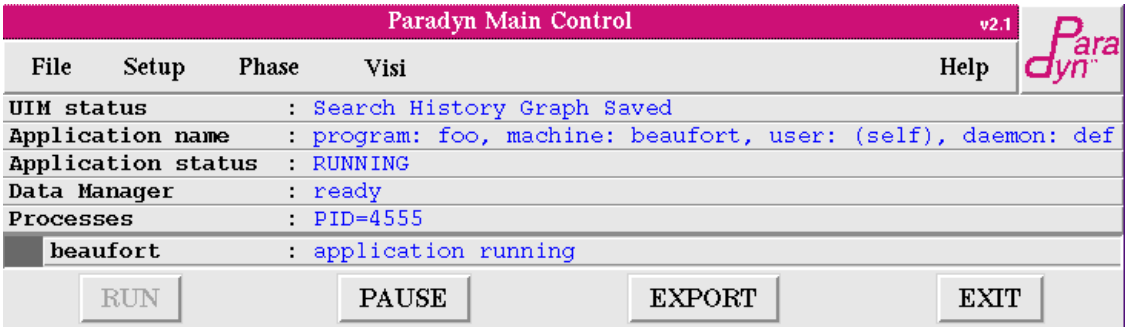


Figure 54: Paradyn Main Window after saving Performance Consultant data.

12 PARADYN CONFIGURATION LANGUAGE

The Paradyn configuration language (PCL) is used for defining daemons, processes, and visis, setting value of tunable constants, and defining new metrics. Paradyn reads commands from one or more of the following files (in this order):

1. a file named `$PARADYN_ROOT/paradyn.rc`, where `PARADYN_ROOT` is a shell environment variable defining a path, or if this file is not found, a file named `paradyn.rc` in the current working directory (see Section 2.2, and also the ***Paradyn Installation Guide***);
2. a file named `$HOME/.paradynrc` in the user's home directory;
3. a configuration file given as a command line argument to Paradyn with the '`-f`' option (e.g. "`paradyn -f foo`").

The remainder of this chapter describes the syntax and semantics of the Paradyn configuration language.

12.1 Notation

We use an extended-BNF (EBNF) notation to describe the syntax of the language. Nonterminal symbols in the grammar are written in *italics*, terminal symbols (tokens) in `courier`, and reserved keywords and symbols are written in **boldface**.

In the description of the grammar the symbol ::= is used to introduce the definition for a nonterminal symbol, a vertical bar | represents a choice, braces {} represent zero or more repetitions, and brackets [ ] are used to represent an optional item. Parentheses are used for grouping.

12.2 Lexical conventions

The tokens of the language are identifiers (`Ident`), integer (`Integer`), floating-point (`Float`), and string (`string`) constants, and the reserved keywords and symbols enumerated below. White spaces, tabs, newlines, and comments are ignored, except to separate tokens. A comment is started by the characters **`//`**. All characters from the **`//`** until the first newline are considered as part of the comment and are ignored.

Identifiers are a sequence of letters, digits, and underscore, starting with a letter or underscore. Identifiers are case sensitive and may be of arbitrary length. Predefined identifiers start with a $ (dollar) sign.

Some identifiers are reserved for use as keywords and cannot be used in any other way. Figure 55 is a list of all keywords in the language (all keywords are case sensitive, except for "true" and "false"). The four words "setCounter", "addCounter", "subCounter", "functionCall" are obsolete, but they are reserved so that MDL can detect an old configuration file.

There are six words: "readAddress", "readSymbol", "startProcessTimer", "stopProcessTimer", "startWallTimer", "stopWallTimer", which are not keywords, but are considered as Paradyn standard function calls. See Section 12.9.10 for an explanation of their meanings.

```
$arg                    $return                 addCounter
aggregateOperator       append                  avg
base                    Call                    command
constraint              constrained             counter
daemon                  default                 derived
developer               dir                     EventCounter
exclude                 false                   flavor
float                   force                   foreach
functionCall            host                    if
is                      in                      int
items                   library                 limit
list                    max                     metric
min                     mode                    module
name                    normal                  normalized
postInsn                preInsn                 prepend
procedure               process                 processTimer
replace                 resourceList            SampledFunction
setCounter              string                  style
subCounter              sum                     true
tunable_constant        units                   unitsType
unnormalized            user                    visi
void                    wallTimer
```

Figure 55: List of MDL keywords

There are three types of constants: strings, integers, and floating-point. A string is a sequence of zero or more characters (not containing a newline or a double quote) surrounded by double quotes. (Note that the usual expansion of control characters does not apply, e.g. "\n" is a string containing two characters, a '\' and a 'n', not a string containing the newline character.)

Integer and floating-point constants are unsigned and defined as:

```
Integer ::= digit { digit }
Float ::= Integer . Integer
```

The following operators are currently supported. More operators may be added in the future.

& = + += - -= / * < > <= >= == != && || ( ) [ ] , . ++

A statement is teminated with a semicolon, and statements are grouped together with curly braces. Instrumentation code are inside brackets (* and *), see Section 12.9.12.

12.3 Language structure

A Paradyn configuration file consists of a sequence of zero or more definitions of daemons, processes, visis, metrics, values for tunable constants, and functions excluded from shared objects.

DefinitionList ::= { *Definition* }

Definition ::=
DaemonDef |
ProcessDef |
TunableDef |
VisiDef |
ExcludeDef |
mdlDef

Each definition introduces a name to an object. The scope of names is global. A name may be redefined, in which case the new definition replaces the old one. Thus, all references to a redefined name become a reference to the newest object that is bound to the name, even if the use is made before the redefinition. However, different types of objects have different name spaces, so a daemon and a process may have the same name, for example. The name and scope rules for metric definitions differ from the rules for other definitions (see Section for a complete description).

One attribute that can appear in many object definitions is a flavor. Paradyn may have different versions that are used on different systems. Currently, there are five versions, one for standard Unix systems, one for Unix systems running PVM, one for MPI, one for Microsoft Windows and one for COW (cluster of workstations). Each of these versions is called a Paradyn flavor. A PCL file can have many object definitions, some of which may make sense only for some flavors of Paradyn. The flavor of an object tells Paradyn that this object is meaningful only for some subset of the flavors of Paradyn, and that it should be ignored for all of the other flavors. When we run Paradyn only those objects that are of the same flavor of the Paradyn that is being used are considered. The others are ignored.

12.4 Daemon definition

DaemonDef ::= **daemon** `Ident` **{** { *DaemonField* } **}**

DaemonField ::=
remoteShell `string;` |
command `string;` |
MPItype `string;` |
flavor `Ident;`

A daemon definition defines a new daemon with a given name. The name is used to identify the daemon in other PCL definitions, such as the process definition. A daemon definition does not cause the daemon to be started immediately; the daemon only starts when an application process that uses that daemon is run.

A daemon definition must include at least one field (the **command** field). The other fields are optional. The **command** field gives the command (that is, the executable file name and command arguments) that Paradyn uses to start the daemon. The executable path may be a relative pathname, in which case Paradyn searches for the file like the shell does, using the user's PATH environment variable on the machine where the daemon will run.

The field **MPItype** should be either "MPICH" or "LAM." Paradyn defaults to MPICH. Using

the wrong MPITtype can have unpredictable results.

The field **flavor** should be one of `pvm, unix, winnt, mpi`. Defining a daemon to be of a wrong flavor can have unpredictable results.

The optional **remoteShell** field allows a substitute start-up shell/mechanism to be specified, to be used instead of the default "rsh" for starting the daemon process.

Example:

```
daemon pd_daemon {
      remoteShell "/bin/ssh";
      command "/u/mjrg/bin/sparc-sun-solaris2.4/paradynd";
      flavor unix;
}
```

This PCL command defines a unix daemon named `pd_daemon` that is started via `/bin/ssh` with the command "`/u/mjrg/bin/sparc-sun-solaris2.4/paradynd`".

Paradyn provides predefined daemons, `defd` and `pvmd`, that are defined as follows.

```
daemon defd {
      command "paradynd";
      flavor unix;
}

daemon pvmd {
      command "paradynd";
      flavor pvm;
}
```

12.5 Process definition

ProcessDef ::=
process `Ident { {` *ProcessField* `} }`

ProcessField ::=
command `string;` |
daemon `Ident;` |
host `string;` |
user `Ident;` |
dir `string;`

A process definition defines an application program to be run by Paradyn. When the user defines a process to Paradyn (either through a configuration file or with the graphic user interface), Paradyn starts the necessary daemons, which read symbol table information from the executable file, inserts the initial instrumentation, and leave the program in a ready to run state. The application processes can then be run by using the appropriate commands from the Paradyn main menu. A process definition is equivalent to the **Define a Process** command in the Paradyn main menu (see Section 2.4).

A process definition has five fields. The required **command** field specifies the command that Paradyn uses to start the process, including the command arguments, if any. The required **daemon** field specifies the daemon that will run that process. The optional **host** field specifies the name of the machine where the process will run. If no **host** field is present, it will default to the default host specified with the `-default_host` command line option (or the local machine, that is, the machine on which Paradyn is running, if the `-default_host` option is not used). The optional **user** field specifies the user name (login) under which the process will run. The local user, that is the user that runs Paradyn, must be authorized to login as the designated user in the designated host. If no **user** field is present, it will default to the same user name under which Paradyn was started. The optional **dir** field specifies the working directory for the process. If no **dir** field is present, it will default to user's home directory on the remote machine.

Example:

```
process foo {
      command "/u/mjrg/bin/mp3d arg1 arg2";
      daemon defd;
}
```

This example defines a process named foo that is started by the command `mp3d` with arguments `arg1 arg2`, and is monitored by the daemon `defd`.

Paradyn only searches for the executable file in the directory specified by the **dir** field. If this field is not given, then the path to the executable file must be absolute.

12.6 Tunable constant definition

TunableConstant ::=
tunable_constant *TunableItem* |
tunable_constant **{** { *TunableItem* } **}**

TunableItem ::=
`string Integer;` |
`string Float;` |
`string` **true;** |
`string` **false;**

A tunable constant definition gives a value for a tunable constant. For a list of all available tunable constants and their values, see Section 4.

Example:

```
tunable_constant "minObservationTime" 10.0;
tunable_constant "suppressSHG" false;
```

In this example, the value of the tunable constant `minObservationTime` is set to 10.0 and the

value of `suppressSHG` is set to false. Alternatively, these two commands could be rewritten as:

```
tunable_constant {
      "minObservationTime" 10.0;
      "suppressSHG" false;
}
```

12.7 Visi definition

VisiDef ::= **visi** `Ident` **{** { *VisiItem* } **}**

VisiItem ::=
command `string;` |
dir `string;` |
user `Ident;` |
host `string;` |
force `Integer;` |
limit `Integer;`

A visi definition gives the command that Paradyn uses to start a new visualization module. The only required field is **command**, which gives the file path (and optional arguments) to the visi program. Paradyn searches for commands according to the shell rules, using the PATH environment variable. **force** is interpreted as a boolean value, and any non-zero value will cause the visi to start without asking the user for metric selections. **limit** is an upper bound on the number of metric/focus pairs that the visi can have enabled at one time. If this field is not specified, or if it has a non-positive value, then there is no upper bound. **dir**, **user** and **host** have the same previously discussed meaning.

Example:

```
visi Histogram {
      command "rthist";
}
```

12.8 Exclude definition

ExcludeDef ::= **exclude** `string;`

The exclude definition specifies a shared object (or dynamically-loaded library) or a function therein that cannot be included in any focus. Performance data cannot be collected from excluded functions or modules. Also, the Performance Consultant will not search in excluded functions or modules. The string that specifies the shared object function or shared object to exclude should be of the form "/Code/shared_library_name/function_name" or "/Code/shared_library_name" (for Paradyn versions 2.1 and above, or of the form "shared_library_name/function_name" or "shared_library_name" for versions below 2.1). The "*" character can be used within exclude definitions as a wildcard which matches zero or more characters in a library name or function name. Modules and functions from *a.out* files cannot be excluded.

Paradyn versions 2.1 and above allow exclusion of both statically and dynamically linked mod-

ules and functions. Static and dynamic code is excluded identically, using the mechanism described above. All modules and functions are included (not excluded), unless otherwise specified, and accordingly all functions, including those in dynamically linked libraries, appear in the $procedures variable described in "Metric Definition Language" section below. This behavior is different from that encountered in older versions of Paradyn in which shared object code was treated differently than statically linked code with respect to exclusion.

The Paradyn Control Language files distributed with releases 2.1 and above have been modified to take these changes into account. Existing unmodified PCL files should be updated as follows:

1. Names of excluded modules and functions should be proceeded by "`/Code`".

2. All dynamic libraries or functions therein whose members should not be included in $procedures should be explicitly excluded.

Example (version 2.1 and above):

```
exclude "/Code/libc.so.1";         #exclude all functions from libc.so.1
exclude "/Code/libthread.so/read"; #exclude function read from libthread.so
```

Example (version 2.0 and below):

```
exclude "libc.so.1";         #exclude all functions from libc.so.1
exclude "libthread.so/read"; #exclude function read from libthread.so
```

12.9 Metric Description Language

The metric description language (MDL) is a sub-language of the PCL that is used for defining new metrics. A metric is a time-varying function that characterizes some aspect of a parallel program performance, such as CPU utilization or number of synchronization operations. Metrics can be computed for the entire program or they can be restricted to program components (called resources) such as a particular procedure, or a particular processor. Metrics can be computed for the global phase (from the start of application execution until the present time) or the currently defined phase. (Phases are described in Section 8.)

A list of resources of interest to the user is called a focus. A metric definition provides a template (the base metric) that is used to compute the metric, and a list of constraints that are combined with the base metric to restrict it to a particular focus. A constraint defines a flag that is set whenever a particular resource is active.

For example, consider a metric that counts how many functions are called. The metric declaration must provide code to increment a counter every time a function is called. The constraints for this metric can provide ways of restricting the computation to a single function, to a single module, or to a single process. When combined with a focus, such as function *f* and process *p*, the metric will count how many times the function *f* is called in process *p*.

If there's no constraint declaration or replace constraint inside a metric definition, the metric can only be applied to the whole program.

Metric and constraint definitions are not evaluated until there is a request to compute a particular set of metrics for particular focus. At this time the requested metrics are evaluated, taking as input the focus. The result of the metric evaluation is a collection of code blocks that are inserted into the application code to compute the metric. In the remainder of this section, the phrase "metric evaluation time" or "metric insertion time" refer to time a metric is evaluated to generate the code to be inserted into the application, and "metric execution time" or simply "metric execution" refer to the time when the generated code is executed.

12.9.1 Metric definition

An MDL definition consists of declarations of one or more MDL objects: resource lists, constraints, and metrics:

mdlDef ::=
resourceListDef |
constraintDef |
metricDef

Each definition introduces a new object with a given name, which is used for references to that object. A name may be redefined, in which case the definition of the old object is replaced with the definition of the new object. The new object is used in all occurrences of the name, including those that precede the redefinition. Therefore, redefinitions of objects must be done with care, or unexpected results may occur. For example, the value of `foo` at the `foreach` statement in the example below is "bar", because the name `msgFilt` has been redefined after the constraint definition.

Example:

```
resourceList msgFilt is procedure {
      items { "foo" };
      library false;
      flavor {unix};
}

constraint msgTagConstraint /SyncObject/Message is counter {
      foreach func in msgFilt {
            prepend preInsn func.entry (*
                  if ($arg[1] == $constraint[0])
                        msgTagConstraint = 1;
            *)
            append preInsn func.return (*
                  msgTagConstraint = 0;
            *)
      }
}

resourceList msgFilt is procedure {
      items { "bar" };
      library false;
      flavor {unix};
}
```

12.9.2 Variables

There are two classes of variables that can be used in metric descriptions: metric insertion variables and instrumentation variables. A metric insertion variable is simply a name that is bound to an object (a list, constraint, or metric). As explained above, the value of these variables can only be modified by binding the name to a new object. Instrumentation variables are like variables in an imperative language (that is, they denote a memory location) and can only be used in instrumentation blocks, that is, the code to be inserted at the application. Metric insertion variables can be used at any place in a metric definition, including an instrumentation block. Instrumentation blocks are delineated in PCL by the `(*` and `*)` tokens.

12.9.3 Types

Instrumentation variables can have one of three types: counter, wallTimer, or processTimer. A counter is equivalent to an integer variable in imperative languages like C or C++. WallTimer and processTimer are abstract types used to record time and can only be manipulated with timer specific functions. A set of predefined functions, that can only be used in instrumentation requests, is provided for operations with timers: startProcessTimer, stopProcessTimer, startWallTimer, stopWallTimer.

Metric insertion variables can have several different types: integer, floating-point, string, point, procedure, module, memory, and list. The types integer, floating-point, and string have the usual meaning.

Type: Point

A `point` is an abstract type that represents a well-defined location in an application code where instrumentation can be inserted (currently available points are function entry, exit, and individual call sites).

Type: Procedure

`Procedure` is a structured type that describes a procedure (function) in the application code:

```
procedure {
      string name;
      point list calls;
      point entry;
      point return;
}
```

The value of each member is implicitly initialized by Paradyn, and cannot be modified. `Name` is the name of the procedure, as defined in the symbol table in the application program's executable file. `Calls` is the list of calls made in the procedure code. `Entry` and `return` are the entry and return points of the procedure.

The dot operator "." is used to access the value of each member of a structured objects like procedures and modules. If `proc` is a procedure object, then `proc.name` gives the name of the proce-

dure, and `proc.entry` gives the entry point of the procedure.

Type: Module

`Module` is a structured type with two fields that describe the module name and its functions. The value of these fields is implicitly initialized by Paradyn.

```
module {
      string name;
      procedure list funcs;
}
```

Type: List

The type `list` consists of an ordered collection of elements of the same type. The elements of a list can be accessed sequentially with the **foreach** statement, or one particular element may be obtained with the subscript operator `[]`.

The **foreach** statement applies a metric statement to each element in a list. For example,

```
foreach callsite in proc.calls
      << metric statement >>
```

applies `metric statement` (metric statements are defined in Section 12.9.8) to each element of the list `proc.calls`. The expression `proc.calls[1]` returns the first element in list `proc.calls`.

12.9.4 Predefined variables

MDL provides a number of predefined variables, described in Figure 56.

Variable Name	Type	Explanation
`$constraint`	procedure, module, or int	The list of components in the resource path. Each component can be accessed through an incremental index, starting from the last element; for example $constraint[0] is the last component, $constraint[1] is the second to last. Each component can be of a different type. (see Section 12.9.3).
`$arg`	int	The list of arguments to a procedure call. A specific argument can be selected with indexing. for example: `$arg[2]`.
`$return`	int	The return value of a function.
`$start`	point	The entry point of the program (usually main).

Figure 56: Predefined variables

Variable Name	Type	Explanation
`$exit`	point	The exit point of the program (e.g. _exithandle for Solaris).
`$procedures`	procedure list	The list of functions in a module.
`$modules`	module list	The list of modules in a program.
`$machine`	string	The machine where a program is running.
`$globalId`	int	An unique identifier to a particular metric/focus/phase combination. It can be used by metrics that need to maintain extra information on a per metric instance basis.

Figure 56: Predefined variables

12.9.5 Resource lists

A resource list statement defines a new MDL variable of type list:

resourceListDef ::=
```
        resourceList Ident is ListType {
              items { StringList };
              flavor { IdentList } ;
              library OptLibrary:
        }
```

ListType ::=
```
        string |
        procedure |
        module |
        float |
        int
```

StringList ::= `string` { `, string` }

IdentList ::= `IDENT` { `, IDENT` }

OptLibrary ::=
true | **false**

The identifier after the keyword **`resourceList`** gives the MDL variable that will be bound to the list. ListType specifies the type of the elements of the list. Items give the list of elements. Library is used when elements are of type procedure, and tells whether the functions in the list are library functions or not. Flavor gives the Paradyn flavors of this list (e.g., unix or pvm).

The elements of a list can be accessed with the foreach statement, described in Section 12.9.8, or via indexing (e.g. `foo[1]`).

Example:

```
resourceList generic_lib_pvm is procedure {
      items {"write", "read"};
      library true;
      flavor {pvm};
}
```

declares a variable generic_lib_pvm, of type procedure list, with two elements (`write` and `read`), which are library functions. This definition of the variable is valid only for PVM.

12.9.6 Constraints

Constraints provide a mechanism to restrict a metric to a subset of the resources in the resource hierarchy. A constraint definition declares a new counter instrumentation variable that is conceptually a boolean flag. This flag is set whenever a certain resource, such as a function or a module, is active.

constraintDef ::=
constraint `Ident` *matchPath* **is default ;**
constraint `Ident` *matchPath* **is counter** { *metricStmt* }

matchPath ::= { **/** `Ident` }

A constraint definition creates a new constraint with a given name that can be used in several metrics. The *matchPath* specifies the resources to be constrained. A *matchPath* is a sequence of resource names, with a "/" used as a delimiter, and it defines a path in the resource hierarchy. The resource that is constrained is determined by the last element in the path. For example, a path `/SyncObject/Message` specifies that the constraint is to children of this path, in this case a particular message class instance; `/Code` specifies that the constraint is applied to modules in a program. A wildcard, "*", is used as a *matchPath* resource name; for example, `/Code/*` specifies that the constraint is applied to functions in a specific module, which is unknown at the time the constraint was created.

At metric insertion time, the selected focus is compared to the matchPath of each constraint in a metric to determine which constraints to apply. For example, the *matchPath* `/Code` matches the foci `/Code/Mod1` and `/Code/Mod2` where `Mod1` and `Mod2` are modules in the application; and, the *matchPath* `/Code/*` matches the foci `/Code/Mod1/F1` and `/Code/Mod2/F2` modules where `F1` and `F2` are functions in the application. The matchPath `/Code/F1/*` matches the foci `/Code/Mod1/F1/loop_1` and `/Code/Mod2/F1/loop_1.1` modules where `loop_1` and `loop_1.1` are loops in `F1`.

The predefined variable `$constraint` is initialized at metric insertion time to the list of components in focus. If path is `/Code/*`, and the foci is `/Code/Mod1/F1` and `/Code/Mod2/F2`, then the value of `$constraint[0]` is a procedure list with `F1` and `F2`, and `$constraint[1]` is a module list with `Mod1` and `Mod2`.

A default constraint defines a constraint that matches some focus. It does not generate any

instrumentation code. Usually, a metric must provide a constraint for each resource that may be specified in a focus. If there is no constraint that matches a given resource, then the metric will fail. Default constraints are used in cases where no action is needed to constrain a particular resource.

Example: a constraint for modules must define a counter that is set to one only if a function in the module is being executed. The constraint definition must direct Paradyn to insert code to set the flag to one whenever a function in the module is called, and set it to zero when the function exits. In addition, we may want to set the flag to zero whenever a function call is made from a function in the module, and reset it after this that call has returned. In this case, the module would not be considered active when an external function is called. The definition of this metric is the following:

```
01: constraint moduleConstraint /Code is counter {
02:   foreach func in $constraint[0].funcs {
03:         prepend preInsn func.entry (*
04:               moduleConstraint = 1;
05:         *)
06:         append preInsn func.return (*
07:               moduleConstraint = 0;
08:         *)
09:         foreach callsite in func.calls {
10:               append preInsn callsite (*
11:                     moduleConstraint = 0;
12:               *)
13:               prepend postInsn callsite (*
14:                     moduleConstraint = 1;
15:               *)
16:         }
17:   }
18: }
```

Line 1 in this program declares a constraint `moduleConstraint`, of type `counter`, that is to be applied to modules. At metric insertion time, the variable `$constraint[0]` will be set to a particular module in the application (usually selected from the Paradyn where axis). `$constraint[0].funcs` is the list of all functions in the module. Line 3 says that code to set the flag should be inserted before the entry point of each function in the module, and line 6 says that code to reset the flag should be inserted before the return point of each function. Lines 9, 10, and 13 say that the flag should be set before any call made inside the function, and reset after the call returns.

12.9.7 Metric definitions

metricDef ::=

metric `Ident` {
name `string` ;
units `Ident`;
unitsType (**normalized** | **unnormalized** | **sampled**) ;
aggregateOperator (**avg** | **sum** | **min** | **max**) ;
style (**EventCounter** | **SampledFunction**);
flavor { *IdentList* } ;

```
                    { mode ( developer | normal ); }
                    { Constraint }
                    { counter Ident ; }
                    base is ( counter | processTimer | wallTimer )
                    { metricStmt } ;
          }

Constraint ::=
          constraint Ident ; |
          constraint Ident { / Ident } is
          replace ( counter | processTimer | wallTimer )
          { metricStmt } ;
IdentList ::=
          Ident { , Ident }
```

A metric definition defines a new metric with a given internal name (the identifier following the **metric** keyword). A metric definition must specify several fields. The **name** is a string that gives the external name of the metric, that is how Paradyn users refer to this metric. **Units** and **unitsType** specifies the label to be used by Paradyn and visis when displaying values of a metric. **Units** can be any string and **unitsType** must be either **normalized**, **unnormalized** or **sampled**. Figure 57 shows how a label is displayed, for each unit type.

Units Type	Data label	Average label	Total label
normalized	units	units	units_seconds
unnormalized	units/sec	units/sec	units
sampled	units	units	units

Figure 57: Metric labels.

AggregateOperator gives the operator used to combine values of the metric for different processes to compute a single value. The **Style** field specifies how to interpret the metric value. Currently, only event counter and sampled function metrics are supported. In an event counter metric, Paradyn samples the value of a metric at periodic intervals, with the difference since the last sample as the reported value. The sampled function metric, however, does not take the difference. One example is to use the sampled function metric to measure the memory access pattern. The **flavor** field gives the flavors of Paradyn for which this metric should be used. The **mode** field indicates whether the metric is for developers. The default is **normal** if this field is not specified.

The rest of the metric definition gives an optional list of constraints that are used to restrict the metric to specific resources, an optional list of auxiliary counters that can be used in instrumentation requests, and the template code for the instrumentation code to compute the metric. A **constraint** declaration either gives the name of a constraint defined elsewhere, or defines a replacement constraint that replaces the base definition of the metric. If a constraint declaration matches the focus, the constraint is used to restrict the metric to the specified program component. If a constraint definition (inside a metric definition, this is called replace constraint) matches the current focus, the constraint replaces the **base** statement of the metric.

12.9.8 Metric statements

There are four metric statements, **foreach** statement, **if** statement, single instrumentation request, and a block of multiple instrumentation requests.

metricStmt ::=
foreach `Ident` **in** *MetricExpr metricStmt* |
if *MetricExpr metricStmt* |
InstrRequest |
{ { *metricStmt* } **};**

The **foreach** statement evaluates a metric expression that should evaluate to a list, and applies a metric statement to each element in the list. It defines a new variable with a name given by `Ident` and that has the same type as the elements of the list. The scope of this variable is *metricStmt*, and its value is bound at each iteration to one element of the list.

The **if** statement evaluates a metric expression that must be of type integer, and it executes *metricStmt* if the value is non-zero.

A single instrumentation request defines instrumentation to be added to an application code. Each instrumentation request will generate a mini-trampoline in the application core.

12.9.9 Metric expressions

MetricExpr ::=
Literal |
(`Ident`) { `.` `Ident` } |
Call ("Ident" [, *ArgList*]) |
`Ident` **(** [*ArgList*] **)** |
MetricExpr BinOP MetricExpr |
PreUOp MetricExpr |
MetricExpr PostUOp |
Ident *AssignOp MetricExpr* |
`Ident` **[** *`MetricExpr`* **]** |
(*MetricExpr* **)**

Literal ::= `Integer | string`

ArgList ::=
MetricExpr { **,** *MetricExpr* }

BinOp ::=
+ | **-** | **/** | ***** | **<** | **>** | **<=** | **>=** | **==** | **&&** | **||**

PreUOp ::= `&` | `-`

PostUOp ::= `++`

AssignOp ::= = | += | −=

More operators will be supported in the future.

A literal is an expression of type integer or string, that has the integer or string literal as its value. An identifier is an expression that has the type and value of the variable bound to the identifier. If the identifier is followed by an expression that evaluates to an integer n between brackets, the identifier must be bound to a list, and the value of the indexed expression is the nth element of the list. The list elements are numbered from zero, and n must be less than the number of elements in the list.

If an identifier is followed by a sequence of dots and identifiers, it must be of a structured type (procedure or module). The second identifier must be the name of a field in the structure. The value and type of the expression are the value and type of this field.

The arithmetic operators `+`, `–`, `*`, and `/`, and the relational operators `<`, `>`, `<=`, and `>=` can be applied to two binary expressions of type integer or floating-point. The operators have the usual meaning and associativity rules. Parenthesized expressions may be used to enforce a different evaluation order. If the two sub-expressions are not of the same type, the values are converted to floating-point. The logical operators `&&` (and) and `||` (or) can be applied to a pair of integer expressions. A zero value denotes false, and any nonzero value denotes true.

The **&** operator returns the address of a variable. The argument reference expression returns the value of one of the arguments of a function. For example, `arg[0]` returns the value of the first argument. Only the value of arguments passed in registers can be obtained.

12.9.10 Function calls

The *MetricExpr* syntax indicates that an metric expression can be a function call. Usually a function call is an identifier followed by a list of arguments in parenthesis. If the function name has conflict with Paradyn's reserved keywords, the alternative syntax of **Call**("Ident"[,*Arglist*]) can be used, with the Ident being the function name. Note that the name must be quoted. The function name can be any legal identifier. However, there are six words which, although are not reserved, are treated as Paradyn standard functions if used as function names. The six words are: "readSymbol", "readAddress", "startProcessTimer", "stopProcessTimer", "startWallTimer", "stopWallTimer".

The expression `readSymbol("sym")` returns the integer value stored in a memory location named `sym`, where `sym` must be defined in the symbol table of the application. For example. if `_intvar` is an integer variable in an application, the expression `readSymbol("_intvar")` returns the value of `_intvar`. **readAddress** returns the integer value at a given address (in base 10), which must be a valid address for the application. **startProcessTimer**, **stopProcessTimer**, **startWallTimer**, **stopWallTimer** start and stop recording time into a timer variable.

The maximum number of arguments that can be passed to a function call may be limited (the limit is architecture dependent, and usually is the maximum number of arguments that can be

passed in registers). The return value of a function is treated as an integer.

12.9.11 Instrumentation requests

An instrumentation request defines a block of instrumentation code to be inserted at a specific point of an application code:

instrRequest : := *position where point* [**`constrained`**] **`(*`** { *instrumentationCode* } **`*)`**

position ::= `append` | `prepend`

where ::= `preInsn` | `postInsn`

point ::= *metricExpr*

Position gives the order in which this instrumentation block will be inserted in the list of instrumentation blocks for this point, and can be used to control the order of execution of multiple blocks at a point. If position is `append`, then the instrumentation block is inserted at the end of the list of instrumentation blocks at the point. If position is `prepend`, the instrumentation block is inserted as the first block in the list.

Where gives the place where the instrumentation block will be inserted, either before (`preInsn`) or after (`postInsn`) the instruction at the instrumentation point is executed. For example, if the point is a call site, `preInsn` specifies that the instrumentation code is to be inserted before the call is made and `postInsn` specifies that it should be inserted after the call returns.

Point is a metric expression (Section 12.9.9) that must evaluate to a point; it gives the point in a program where instrumentation is to be inserted. Currently, the possible points are function entry and exit points, and function calls.

Constrained determines if constraints should be applied to this request. If **constrained** is not specified, no constraints will be applied to the request.

12.9.12 Instrumentation code

The instrumentation code gives a list of statements to be inserted at a point. A statement is either an if statement or a simple instrumentation statement.

instrumentationCode ::=
if **(** *metricExpr* **)** *instrStmt* |
instrStmt

The **if** statement evaluates the *metricExpr* and if it the result is a nonzero value, then the *instrStmt* is executed.

An instrumentation statement is an *MetricExpr* terminated by a semicolon.

The following are examples of valid instrumentation code:

```
cntr = 1;

cntr += foo(cntr);
cntr += Call ("foo", cntr);

if (readSymbol("_foos") == 1) cntr -= readAddress(123456));

startWallTimer(tmr);

cntr = cntr - $arg[2];
```

The first example sets the value of counter `cntr` to 1. `cntr` must be a variable of type counter declared in a constraint declaration, a metric declaration, or a in a counter declaration. The second example calls a function `foo` in the application code, passing the value of counter `cntr` as an argument, and then adds the value returned by this call to counter `cntr`. `foo` must be a function taking one integer argument and returning an integer value, and it must be defined in the application's symbol table.

The third example reads the value of a global variable `foos` (note that if the variable name is foos it must be referenced as `_foos`) in the application and if the value is equal to 1, subtracts the integer value at address 123456 in the application address space from counter `cntr`. The fourth example starts recording time in timer `tmr`, which must be declared in a metric declaration. The timer will record time until a call to `stopWallTimer` is made on it. The last example subtracts the value of the third argument to a function call (the function that is being instrumented) from timer `cntr`.

12.9.13 Interaction of constraints and metrics

When creating a constraint and metric, one of the things you must do is specify where the primitives (for the constraints) and predicates (for the metrics) are placed. They can go before the instruction you are placing it at, or after. Then, for each location (preInsn or postInsn) there is an ordered list for the instructions; and, you are able to specify if you want to append or prepend the instrumented code. The following rules, and patterns, should be adopted when doing this

Constraints:

```
prepend preInsn func.entry
append preInsn func.return
```

Callsites within constraints (if necessary):

```
foreach callsite in func.calls {
      append preInsn callsite
      prepend postInsn callsite
}
```

Metrics:

```
foreach func in XXX {
      append preInsn func.entry
      prepend preInsn func.return
}
```

This is done to make sure the metrics and predicates are checked at a time when all the constraints and primitives are set with their correct values. This set of rules and patterns have the con-

straints and primitives be the first thing set when entering a function (prepend preInsn func.entry) and then the last thing cleared when returning from a function (append preInsn func.return). Along the same lines, the metrics and predicates are the last thing checked when entering a function (append preInsn func.entry) and the first thing checked when returning from a function (prepend preInsn func.return). For example, if you have a metric M1 using constraint C1, which is set when your in a specific function and cleared when you leave that function. With these rules and patterns, C1 will be the first thing set, before M1 is executed at the beginning of the function; and, at the end of the function M1 will be executed before C1 is cleared again.

12.9.14 A complete example

This section presents a complete metric definition. We will define a metric called SyncWait that computes the time spent on by an application on synchronization operations. The first step in the definition of SyncWait is to identify the synchronization operations (functions) of the application. These depend on the specific system that is being used. For example, for PVM applications, we can consider the functions pvm_send and pvm_recv as synchronization functions.

Next, we defined a resource list with the synchronization functions.

```
resourceList pvm_sync_ops is procedure {
            items { "pvm_send", "pvm_recv" };
            flavor { pvm };
            library true;
}
```

To compute the synchronization time we must start a timer every time one of the functions in pvm_sync_ops is called (lines 2 and 3 in the following code block), and stop the same timer when

the function returns (lines 4 and 5).

```
01: foreach func in pvm_sync_ops {
02:   append preInsn func.entry constrained
03:         (* startWallTimer(p_syncWait); *)
04:   prepend postInsn func.return constrained
05:         (* stopWallTimer(p_syncWait); *)
06: }
```

The complete metric definition must define all of the metric attributes and constraints. The constraints define how to compute the metric for specific resources, such as a function, or a module. To constraint the metric to a function, we need to set a flag when the function is entered (lines 2 and 3 in the code block below), and reset it when the function exits (lines 4 and 5). We also reset the flag before any function call inside the function, and set it when the call returns (lines 6 to 10).

```
1: constraint funcConstraint /Code/* is counter {
2:    prepend preInsn $constraint[0].entry
3:          (* funcConstraint = 1; *)
4:    append postInsn $constraint[0].return
5:          (* funcConstraint = 0; *)
6:    foreach callsite in $constraint[0].calls {
7:          append preInsn callsite
8:                (* funcConstraint = 0; *)
9:          prepend postInsn callsite
10:               (* funcConstraint = 1; *)
11:   }
12: }
```

We can also define a constraint for message tags, in case we are interested in finding the time the application is waiting for a particular message tag. At the entry point of each synchronization function (lines 3 to 5 in the code block below) we must check if the tag of the message (the second argument in a call to pvm_send or pvm_recv) is equal to the tag specified in the focus (line 4), and if so set the constraint flag to one (line 5). The flag is set to zero again at the return point of the function (lines 7 and 8).

```
1: constraint msgTagConstraint /SyncObject/Message is counter {
2:    foreach func in pvm_sync_ops {
3:          prepend preInsn func.entry constrained
4:                (* if ($arg[1] == $constraint[0])
5:                      msgTagConstraint = 1;
6:                *)
7:          append preInsn func.return constrained
8:                (* msgTagConstraint = 0; *)
9:    }
10: }
```

Finally we must specify the remaining attributes of the metric, such as the name that will appear in the Paradyn metric selection menu, `PVM SyncWait`. The unit is seconds since this metric measures time, and the unit style is **normalized**. Aggregate operator is **avg**, so when we aggregate values from different processes, we get the average value. The flavor is `pvm`.

The complete definition of the metric follows. The constraint `moduleConstraint` was

defined in Section 12.9.6.

```
metric p_syncWait {
      name "PVM SyncWait";
      units Seconds;
      unitStyle normalized;
      aggregateOperator avg;
      style EventCounter;
      flavor = { pvm };

      constraint functionConstraint;
      constraint moduleConstraint;
      constraint msgTagConstraint;

      base is wallTimer {
            foreach func in pvm_sync_ops {
                  append preInsn func.entry constrained (*
                        startWallTimer(p_syncWait);
                  *)
                  prepend preInsn func.return constrained (*
                        stopWallTimer(p_syncWait);
                  *)
            }
      }
}
```

For useful hints and guidelines for writing metrics, consult the ***MDL Programmer's Guide***.

n