WorkShop: Beyond the Basics

THE NETWORK IS THE COMPUTER™

SunSoft, Inc.
A Sun Microsystems, Inc. Business
2550 Garcia Avenue
Mountain View, CA 94043 USA
415 960-1300 fax 415 969-9131

Part No.: 802-7044-10
Revision A, December 1996

Copyright 1996 Sun Microsystems, Inc., 2550 Garcia Avenue, Mountain View, California 94043-1100 U.S.A. All rights reserved.

This product or document is protected by copyright and distributed under licenses restricting its use, copying, distribution, and decompilation. No part of this product or document may be reproduced in any form by any means without prior written authorization of Sun and its licensors, if any.

Portions of this product may be derived from the UNIX® system, licensed from Novell, Inc., and from the Berkeley 4.3 BSD system, licensed from the University of California. UNIX is a registered trademark in the United States and other countries and is exclusively licensed by X/Open Company Ltd. Third-party software, including font technology in this product, is protected by copyright and licensed from Sun's suppliers.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions of FAR 52.227-14(g)(2)(6/87) and FAR 52.227-19(6/87), or DFAR 252.227-7015(b)(6/95) and DFAR 227.7202-3(a).

Sun, Sun Microsystems, the Sun logo, SunSoft, Solaris, Sun OS, Sun WorkShop, Sun WorkShop TeamWare, Sun Performance WorkShop, Sun Visual WorkShop, LoopTool, LockLint, Thread Analyzer, Sun C, Sun C++, Sun FORTRAN, Answerbook, and SunExpress are trademarks or registered trademarks of Sun Microsystems, Inc. in the United States and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.

The OPEN LOOK® and Sun™ Graphical User Interfaces were developed by Sun Microsystems, Inc. for its users and licensees. Sun acknowledges the pioneering efforts of Xerox in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a non-exclusive license from Xerox to the Xerox Graphical User Interface, which license also covers Sun's licensees who implement OPEN LOOK GUIs and otherwise comply with Sun's written license agreements.

THIS PUBLICATION IS PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT.

Please
Recycle

Contents

Preface. vii

Part 1 —Using Distributed Make

1. Using DistributedMake 3

Basic Concepts 3

What You Should Know About DMake Before You Use It 7

How to Use DMake 13

Part 2 —Performance Tuning Multithreaded Programs

2. Multithreaded Concepts 19

Basic Concepts 20

The WorkShop MP/MT Solution 20

Other Sources of Information 21

3. Analyzing Loops 23

Basic Concepts 23

Setting Up Your Environment 24

Creating a Loop Timing File 25

Starting LoopTool. 26
Using LoopTool 27
Other Compilation Options 32
Compiler Hints. 34
Compiler Optimizations and How They Affect Loops 38

4. Analyzing Threads 41

Basic Concepts 42
Compiling and Instrumenting the Source. 43
Running an Experiment 44
Starting Thread Analyzer 44
Exiting Thread Analyzer. 46
Loading a Trace Directory. 46
Navigating the Thread Analyzer Glyph Hierarchy. 47
Thread Analyzer Menus 48
Thread Analyzer Usage Scenarios 53

Part 3 —Visualizing Data

5. Data Visualization 69

Basic Concepts 69
Specifying Proper Array Expressions 70
Automatic Updating of Array Displays 72
Changing Your Display. 73
Analyzing Data 77
Fortran Example Program. 80
C Example Program. 81

Part 4 —Advanced Debugging

6. Runtime Checking **85**

Basic Concepts 85

Using RTC. 87

Using Access Checking 87

Using Memory Use Checking. 90

Setting Options. 95

7. Using Fix and Continue **97**

Basic Concepts 97

Fixing Your Program 98

Continuing after Fixing. 99

Changing Variables after Fixing. 100

Index 103

Preface

Sun WorkShop offers advanced programming, analysis, and debugging features that are covered in this volume, *WorkShop*: *Beyond the Basics*, including:

- DMake™
- LoopTool™
- Thread Analyzer™
- Data Visualization
- Advanced Debugging

Who Should Use This Book

WorkShop: *Beyond the Basics* is directed toward the advanced developer who is building, analyzing, debugging, or optimizing distributed, multithreaded, and/or multiprocessor applications and programs.

Before You Read This Book

For an overview and details about using the WorkShop suite of applications, see *WorkShop: Getting Started*.

DMake users will find basic information about the Sun WorkShop TeamWare suite of applications in *Sun WorkShop TeamWare: User's Guide* and *Sun WorkShop TeamWare: Solutions Guide*.

The WorkShop command-line applications and utilities are described in *WorkShop: Command-Line Utilities*.

How This Book Is Organized

WorkShop: Beyond the Basics contains the following chapters:

Chapter 1, “Using DistributedMake,” explains how to use `DMake`.

Chapter 2, “Multithreaded Concepts,” describes the principles behind multithreaded programs.

Chapter 3, “Analyzing Loops,” explains how to use LoopTool to analyze program performance.

Chapter 4, “Analyzing Threads,” describes how to use Thread Analyzer to perform analysis of threaded programs.

Chapter 5, “Data Visualization,” describes how to use the data visualizer to visualize an array.

Chapter 6, “Runtime Checking” describes how to automatically detect runtime errors in an application during the development phase.

Chapter 7, “Using Fix and Continue” describes how to quickly recompile edited source code without stopping the debugging process.

How to Get Help

This release of WorkShop includes a new documentation delivery system as well as online manuals and video demonstrations. To find out more, you can start in any of the following places:

- **Online Help** – A new help system containing extensive task-oriented, context-sensitive help. To access the help, choose Help ➤ Help Contents. Help menus are available in all WorkShop windows.
- **WorkShop Documentation** – A complete set of online manuals. These manuals make up the complete documentation set for WorkShop and are available using AnswerBook™ or (at the user's option) using an HTML browser. To access the online manuals, choose Help ➤ WorkShop Manuals in any WorkShop window.
- **Video Demonstrations** – These demos provide a general overview of the WorkShop and describe how to use WorkShop to build targets or debug programs. To access them, choose Help ➤ Demos in the WorkShop main window.

- **Release Notes** – The Release Notes contain general information about WorkShop and specific information about software limitations and bugs. To access the Release Notes, choose Help ➤ Release Notes.

How to Access the AnswerBook Documentation

To access the AnswerBook online documentation for WorkShop, you must run a script to set up your environment.

1. **To start AnswerBook, type the following at a command prompt:**

   ```
   % workshop-answerbooks
   ```

   The script sets the `AB_CARDCATALOG` environment variable and runs `/usr/openwin/bin/answerbook`. The AnswerBook Navigator opens and displays the available AnswerBook documents.

2. **Add the WorkShop AnswerBook documents to your library by clicking the Modify Library button.**
   The AnswerBook Navigator: Modify Library window is displayed.

3. **Select the AnswerBook documents you wish to add to your library from the list; then click the Apply button.**
   The AnswerBook documents are added to your library.

4. **To view an AnswerBook document, double-click on the title you wish to view.**

Related Books

The Sun WorkShop provides comprehensive documentation. Depending on which version of WorkShop you have, the following books are available in online and printed forms (except where noted). Some documents are available with all WorkShop products, others are not (as noted).

Sun WorkShop Documentation

Available with all WorkShop products.

WorkShop *Roadmap*	(hard copy only) Provides a documentation map to the WorkShop printed and online documentation. Includes a complete list of the documentation included with your WorkShop.
WorkShop Installation and Licensing Guide	Provides instructions about product licensing and installation of Workshop products on Solaris™ 2.x systems. Provides instructions for local or remote installation for single independent license servers, multiple independent license servers, and redundant license servers.
WorkShop Quick Install for Solaris	Provides quick installation instructions for product installation and licensing.
WorkShop: Getting Started	Provides a basic introduction. This book provides the information you need to use the basic WorkShop features.
WorkShop: Beyond the Basics	Contains information about the advanced programming, debugging, browsing, and visualization applications in the WorkShop product suite, including: DMake, LoopTool, Thread Analyzer, WorkShop Browsing, and WorkShop Visual.
WorkShop: Command-Line Utilities	Provides reference information for all of the workshop utilities that can be run directly from the command line, including Loop Report, LockLint Utilities, sbquery, and all the dbx commands.
WorkShop Online Help	(online only) Contains extensive task-oriented information for all the tools included with the WorkShop
WorkShop Video Demonstrations	(online only) Three video demonstrations providing information about WorkShop building and debugging as well as general product information.
Release Notes	(online only) Contains any information that was too late to get into the other documentation. To access the Release Notes, open any Help menu and then click on Release Notes.
Manual Pages	(online only) Provide information about the WorkShop command-line utilities.

Sun WorkShop TeamWare Documentation

Available only with Performance WorkShop Fortran and Visual WorkShop C++.

Sun WorkShop TeamWare: User's Guide	Describes how to use all the tools in the TeamWare toolset, for both the command- line interface and the graphical user interface.
Sun WorkShop TeamWare: Solutions Guide	Provides an in-depth case study and eight scenario-based topics to help users take full advantage of TeamWare's features.
Sun WorkShop TeamWare Online Help	Provides succinct task-oriented information to help you become familiar with the application. Help volume includes video demonstrations.
Manual Pages	(online only) Provide information about the TeamWare command-line commands and utilities.

Sun Visual WorkShop C++ Documentation

Available only with Sun Visual WorkShop C++.

WorkShop documentation	Visual WorkShop C++ contains the entire WorkShop and TeamWare documentation sets.
C++ User's Guide	Describes how to use the Sun C++ compiler to write programs in C++. It covers the C++ compiler options, programs, templates, exception handling, and more. It is intended for the experienced C++ programmer.
C++ Library Reference	Describes how to use the complex, coroutine, and iostream libraries, and it lists the manual pages (man pages) for these libraries.
Tools.h++ User's Guide	Describes how to use the Tools.h++ libraries to make programs more efficient.
Tools.h++ Class Library Reference	Describes how to use the Tools.h++ class library, and also describes a set of C++ classes that can simplify programming while maintaining efficiency.
C++ 4.2 Quick Reference Card	Provides concise descriptions of the C++ compiler flags.
C User's Guide	Describes how to use the Sun ANSI C compiler to write programs in C. It covers the C compiler options, the pragmas, the lint tool, the cscope tool, and more. It is intended for the experienced C programmer.
Making the Transition to ANSI C	(online only) Provides information about the transition from K&R C to ANSI C.
C 4.2 Quick Reference Card	Provides concise descriptions of the C compiler flags.

WorkShop: Visual User's Guide	Explains how to use Visual, an interactive tool for building graphical user interfaces (GUIs) using the widgets of the standard OSF/Motif toolkit or Microsoft Foundation Class. It includes a tutorial as well as reference information for the more advanced user.
Sun WorkShop 2.0 Visual Quick Reference Card	Contains menu shortcuts and icon explanations for Visual.
Numerical Computation Guide	Describes the floating-point software and hardware for the SPARC™, Intel, HP 700, and PowerPC system architectures. It also contains a tutorial on floating-point arithmetic.
Incremental Link Editor	Describes how to use `ild` as an incremental linker to replace `ld` for linking programs. `ild` allows you to complete the development sequence more quickly than is possible with a standard linker.
Performance Profiling Tools	Describes the `prof`(1), `gprof`(1), and `tcov`(1) utilities
Manual Pages	(online only) Provides information about the command-line commands and utilities included with Visual WorkShop C++.

Sun Performance WorkShop Fortran Documentation

Available only with Sun Performance WorkShop Fortran.

WorkShop documentation	Performance WorkShop Fortran contains the entire WorkShop and TeamWare documentation sets.
Fortran User's Guide	Describes how to use the Sun Fortran 77 4.0 and Fortran 90 1.2 compilers, including the compiler command options, debugging and development tools, program profiling and performance tuning, mixing C and Fortran, and making and using libraries. It is intended for programmers with knowledge of Fortran.
FORTRAN 77 Language Reference	Describes and defines the Fortran 77 language accepted by the Sun `f77` compiler under Solaris 1.x and 2.x. It is intended for use by programmers with knowledge of and experience with Fortran.
Fortran Programmer's Guide	Provides the essential information programmers need to develop efficient applications using the Fortran 77 and Fortran 90 compilers. Includes information on input/output, program development, use and creation of software libraries, program analysis and debugging, numerical accuracy, porting, performance, optimization, parallelization, and the C/Fortran interface.

Fortran Library Reference	(online only) Describes the language and routines of the Fortran compilers.
Sun Performance Library 1.2 Quick Reference Card	Provides a quick reference to Sun Performance Library language routines with brief descriptions.
Fortran 90 Handbook	(online only) Contains user-level information about this release of Fortran90.
Fortran 90 Browser	Describes how to use the Sun Fortran 90 Browser, one of the development tools in the `f90` package, to view Fortran 90 source code. It is intended for programmers with knowledge of Fortran 90.
Fortran Quick Reference Card	Lists the `f77` 4.0 compiler's command-line options with brief descriptions.
C User's Guide	Describes how to use the Sun ANSI C compiler to write programs in C. It covers the C compiler options, the pragmas, the lint tool, the cscope tool, and more. It is intended for the experienced C programmer.
Making the Transition to ANSI C	(online only) Provides information about the transition from K&R C to ANSI C.
C 4.2 Quick Reference Card	Describes the C compiler options in a concise and easy-to-read format.
Numerical Computation Guide	Describes the floating-point software and hardware for the SPARC™, Intel, HP 700, and PowerPC system architectures. It also contains a tutorial on floating-point arithmetic.
Incremental Link Editor	Describes how to use `ild` as an incremental linker to replace `ld` for linking programs. `ild` allows you to complete the development sequence more quickly than is possible with a standard linker.
Performance Profiling Tools	Describes the `prof`(1), `gprof`(1), and `tcov`(1) utilities.
Manual Pages	(online only) Provide information about the Fortran command-line commands and utilities included with Performance WorkShop Fortran.

Sun WorkShop Professional Pascal Documentation

Available only with Sun WorkShop Professional Pascal.

WorkShop documentation	WorkShop Professional Pascal contains the entire WorkShop documentation set.

Pascal User's Guide	Describes how to begin writing and compiling Pascal programs for the Solaris computing environment. Pascal is a derivative of the Berkeley Pascal system distributed with UNIX®4.2 BSD. It complies with FIPS PUB 109 ANSI/IEEE 770 X3.97-1983 and BS6192/ISO7185 at both level 0 and level 1, and it includes many extensions to the standard.
Pascal Language Reference	Provides reference material for the Pascal 4.0 compiler, an implementation of the Pascal language that includes all the standard language elements and many extensions. Pascal 4.0 contains a compiler switch, `-xl`, to provide compatibility with Apollo DOMAIN Pascal to ease the task of porting your Apollo Pascal applications to workstations.
Pascal 4.2 Quick Reference Card	Lists all of the Pascal 4.2 compiler options with a brief, one-line description of each option.
Numerical Computation Guide	Describes the floating-point software and hardware for the SPARC™, Intel, HP 700, and PowerPC system architectures. It also contains a tutorial on floating-point arithmetic.
Incremental Link Editor	Describes how to use `ild` as an incremental linker to replace `ld` for linking programs. `ild` allows you to complete the development sequence more quickly than is possible with a standard linker.
Performance Profiling Tools	Describes the `prof`(1), `gprof`(1), and `tcov`(1) utilities.
Manual Pages	(online only) Provide information about the command-line commands and utilities included with WorkShop Professional Pascal.

Sun WorkShop Professional C Documentation

Available with Sun WorkShop Professional C only.

WorkShop documentation	WorkShop Professional C contains the entire WorkShop documentation set.
C User's Guide	Describes how to use the Sun ANSI C compiler to write programs in C. It covers the C compiler options, the pragmas, the lint tool, the cscope tool, and more. It is intended for the experienced C programmer.
Making the Transition to ANSI C	(online only) Provides information about the transition from K&R C to ANSI C.
C 4.2 Quick Reference Card	Describes the C compiler options in a concise and easy-to-read format.

Numerical Computation Guide	Describes the floating-point software and hardware for the SPARC™, x86, HP 700 and PowerPC system architectures. It also contains a tutorial on floating-point arithmetic.
Incremental Link Editor	Describes how to use `ild` as an incremental linker to replace `ld` for linking programs. `ild` allows you to complete the development sequence more quickly than is possible with a standard linker.
Performance Profiling Tools	Describes the `prof`(1), `gprof`(1), and `tcov`(1) utilities.
Manual Pages	(online only) Provides information about the command-line commands and utilities included with WorkShop Professional C.

Other Related Documentation

Threads Primer: A Guide to Multithreaded Programming	(ISBN 0-13-443698-9) This SunSoft Press book by Bill Lewis and Daniel J. Berg provides a basic understanding of threads—what they are, how they work, and why they are useful.

Ordering Additional Hardcopy Documentation

You can order additional copies of the hard copy documentation by calling SunExpress at 1-800-USE-SUNX, or visiting their web page at

`http://sunexpress.usec.sun.com`

Sun on the World Wide Web

World Wide Web (WWW) users can view Sun's Developer Products site at the following URL:

`http://sun-www.EBay.Sun.COM:80/sunsoft/Developer-products/products.html`

This area is updated regularly and contains helpful information, including current release and configuration tables, special programs, and success stories.

Sun Education Classes

Sun Educational Services offers the following class for programmers who are developing multithreaded applications:

Multithreaded Applications Programming (#SI-260)

This class shows you how to design, write, and debug multithreaded applications using the Solaris 2+ `libthread` user threads library. The class is designed for experienced C programmers who are proficient in system interface programming.

For more information about this class, contact Sun Education by telephone or email:

Sun Education Registrar 1-800-422-8020 or (408) 263-9367

`training_seats@sun.com` (schedule and availability)

`edbrochure@sun.com` (class description)

Or go to the Sun Educational Services Web site:

`http://www.sun.com/sunservice/suned`

Solaris Technology Camp

Multithreading Camp is offered through Solaris Technology Camps. Space and availability are extremely limited. For dates, locations, prerequisites, course overview, an online registration form, and other information, open the following URL:

`http://www.sun.com/cgi-bin/show?sunsoft/Dev-progs/oldfiles/tech-camp.html`

What Typographic Changes Mean

The following table describes the typographic changes used in this book.

Table P-1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% You have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
AaBbCc123	Command-line placeholder: replace with a real name or value	To delete a file, type `rm` *filename*.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.

Shell Prompts in Command Examples

The following table shows the default system prompt and superuser prompt for the C shell, Bourne shell, and Korn shell.

Table P-2 Shell Prompts

Shell	Prompt
C shell prompt	`machine_name%`
C shell superuser prompt	`machine_name#`
Bourne shell and Korn shell prompt	`$`
Bourne shell and Korn shell superuser prompt	`#`
Korn shell running dbx	`(dbx)`

Part 1 — Using Distributed Make

Using DistributedMake 1

This chapter describes the way DistributedMake (DMake) distributes builds over several hosts to build programs concurrently over a number of workstations or multiple CPUs.

Basic Concepts	*page 3*
What You Should Know About DMake Before You Use It	*page 7*
How to Use DMake	*page 13*

Basic Concepts

DistributedMake (DMake) allows you to concurrently distribute the process of building large projects, consisting of many programs, over a number of workstations and, in the case of multiprocessor systems, over multiple CPUs. DMake parses your makefiles and:

- Determines which targets can be built concurrently
- Distributes the build of those targets over a number of hosts designated by you

DMake is a superset of the `make` utility.

To understand DMake, you should know about the following:

- Configuration files
  - Runtime
  - Build server

- The DMake host
- The build server

Configuration Files

DMake consults two files to determine to which build servers jobs are distributed and how many jobs can be distributed to each.

Runtime Configuration File

DMake searches for a runtime configuration file on the DMake host to know where to distribute jobs. Generally, this file is in your home directory on the DMake host and is named `.dmakerc`. It consists of a list of build servers and the number of jobs to be distributed to each build server. See "The DMake Host" on page 4 for more information.

Build Server Configuration File

The `/etc/opt/SPROdmake/dmake.conf` file is in the file system of build servers. It is used to specify the maximum total number of DMake jobs that can be distributed to it by all DMake users. See "The Build Server" on page 7 for more information.

The `DMake` *Host*

DMake searches for a runtime configuration file to know where to distribute jobs. Generally, this file must be in your home directory on the DMake host and is named `.dmakerc`. DMake searches for the runtime configuration file in these locations and in the following order:

1. The path name you specify on the command line using the `-c` option
2. The path name you specify using the DMAKE_RCFILE makefile macro
3. The path name you specify using the DMAKE_RCFILE environment variable
4. $(HOME)/`.dmakerc`

If a runtime configuration file is not found, DMake distributes two jobs to the DMake host. You edit the runtime configuration file so that it consists of a list of build servers and the number of jobs you want distributed to each build server. The following is an example of a `.dmakerc` file:

```
# My machine. This entry causes dmake to distribute to it.
falcon  { jobs = 1 }
hawk
eagle   { jobs = 3 }
# Manager's machine. She's usually at meetings
heron   { jobs = 4 }
avocet
```

- The entries: falcon, hawk, eagle, heron, and avocet are listed build servers.
- You can specify the number of jobs you want distributed to each build server. The default number of jobs is two.
- Any line that begins with the "#" character is interpreted as a comment.

Note – This list of build servers includes falcon which is also the DMake host. The DMake host can also be specified as a build server. If you do not include it in the runtime configuration file, no DMake jobs are distributed to it.

You can also construct groups of build servers in the runtime configuration file. This provides you with the flexibility of easily switching between different groups of build servers as circumstances warrant. For instance you may define a different group of build servers for builds under different operating systems, or on groups of build servers that have special software installed on them.

The following is an example of a runtime configuration file that contains groups of build servers:

```
earth   { jobs = 2 }
mars    { jobs = 3 }

group lab1 {
        host falcon{ jobs = 3 }
        host hawk
        host eagle { jobs = 3 }
}

group lab2 {
        host heron
        host avocet{ jobs = 3 }
        host stilt { jobs = 2 }
}

group labs {
        group lab1
        group lab2
}

group sunos5.x {
        group labs
        host jupiter
        host venus{ jobs = 2 }
        host pluto { jobs = 3 }
}
```

- Formal groups are specified by the “group” directive and lists of their members are delimited by braces ({}).
- Build servers that are members of groups are specified by the optional “host” directive.
- Groups can be members of other groups.
- Individual build servers can be listed in runtime configuration files that also contain groups of build servers; in this case DMake treats these build servers as members of the *unnamed* group.

In order of precedence, DMake distributes jobs to:

1. The formal group specified on the command-line as an argument to the `-g` option
2. The formal group specified by the DMAKE_GROUP makefile macro
3. The formal group specified by the DMAKE_GROUP environment variable
4. The first group specified in the runtime configuration file.

The Build Server

The `/etc/opt/SPROdmake/dmake.conf` file is in the file system of build servers. Use this file to limit the maximum total number of DMake jobs (from all users) that can run concurrently on a build server. The following is an example of an `/etc/opt/SPROdmake.conf file`. This file sets the maximum number of DMake jobs permitted to run on a build server (from all DMake users) to be eight.

```
jobs: 8
```

Note – If the `/etc/opt/SPROdmake.conf` file does not exist on a build server, no DMake jobs will be allowed to run on that server.

What You Should Know About DMake Before You Use It

To use DMake, you use the executable file (`dmake`) in place of the standard `make` utility. You should understand the Solaris `make` utility before you use DMake. If you need to read more about the `make` utility see the *Programming Utilities Guide* in the *Solaris 2.5 Software Developer AnswerBook* documentation set. If you use the `make` utility, the transition to DMake requires little if any alteration.

DMake's Impact on Makefiles

The methods and examples shown in this section present the kinds of problems that lend themselves to solution with DMake. This section does not suggest that any one approach or example is the best. Compromises between clarity and functionality were made in many of the examples.

As procedures become more complicated, so do the makefiles that implement them. You must know which approach will yield a reasonable makefile that works. The examples in this section illustrate common code-development predicaments and some straightforward methods to simplify them using DMake.

Using Makefile Templates

If you use a makefile template from the outset of your project, custom makefiles that evolve from the makefile templates will be:

- More familiar
- Easier to understand
- Easier to integrate
- Easier to maintain
- Easier to reuse

The less time you spend editing makefiles, the more time you have to develop your program or project.

Building Targets Concurrently

Large software projects typically consist of multiple independent modules that can be built concurrently. DMake supports concurrent processing of targets on a multiple machines over a network. This concurrency can markedly reduce the time required to build a large project.

When given a target to build, DMake checks the dependencies associated with that target, and builds those that are out of date. Building those dependencies may, in turn, entail building some of their dependencies. When distributing jobs, DMake starts every target that it can. As these targets complete, DMake starts other targets. Nested invocations of DMake are not run concurrently by default, but this can be changed (see “Restricting Parallelism” on page 12 for more information).

Since DMake builds multiple targets concurrently, the output of each build is produced simultaneously. To avoid intermixing the output of various commands, DMake collects output from each build separately. DMake displays the commands before they are executed. If an executed command generates

any output, warnings, or errors, DMake displays the entire output for that command. Since commands started later may finish earlier, this output may be displayed in an unexpected order.

Limitations on Makefiles

Concurrent building of multiple targets places some restrictions on makefiles. Makefiles that depend on the implicit ordering of dependencies may fail when built concurrently. Targets in makefiles that modify the same files may fail if those files are modified concurrently by two different targets. Some examples of possible problems are discussed in this section.

Dependency Lists

When building targets concurrently, it is important that dependency lists be accurate. For example, if two executables use the same object file but only one specifies the dependency, then the build may cause errors when done concurrently. For example, consider the following makefile fragment:

```
all: prog1 prog2
prog1: prog1.o aux.o
    $(LINK.c) prog1.o aux.o -o prog1
prog2: prog2.o
    $(LINK.c) prog2.o aux.o -o prog2
```

When built serially, the target `aux.o` is built as a dependent of `prog1` and is up-to-date for the build of `prog2`. If built in parallel, the link of `prog2` may begin before `aux.o` is built, and is therefore incorrect. The `.KEEP_STATE` feature of `make` detects some dependencies, but not the one shown above.

Explicit Ordering of Dependency Lists

Other examples of implicit ordering dependencies are more difficult to fix. For example, if all of the headers for a system must be constructed before anything else is built, then everything must be dependent on this construction. This causes the makefile to be more complex and increases the potential for error when new targets are added to the makefile. The user can specify the special target `.WAIT` in a makefile to indicate this implicit ordering of dependents. When DMake encounters the `.WAIT` target in a dependency list, it finishes processing all prior dependents before proceeding with the following

dependents. More than one .WAIT target can be used in a dependency list. The following example shows how to use .WAIT to indicate that the headers must be constructed before anything else.

```
all: hdrs .WAIT libs functions
```

You can add an empty rule for the .WAIT target to the makefile so that the makefile is backward-compatible.

Concurrent File Modification

You must make sure that targets built concurrently do not attempt to modify the same files at the same time. This can happen in a variety of ways. If a new suffix rule is defined that must use a temporary file, the temporary file name must be different for each target. You can accomplish this by using the dynamic macros $@ or $*. For example, a .c.o rule which performs some modification of the .c file before compiling it might be defined as:

```
.c.o:
    awk -f modify.awk $*.c > $*.mod.c
    $(COMPILE.c) $*.mod.c -o $*.o
    $(RM) $*.mod.c
```

Concurrent Library Update

Another potential concurrency problem is the default rule for creating libraries that also modifies a fixed file, that is, the library. The inappropriate .c.a rule causes DMake to build each object file and then archive that object file. When DMake archives two object files in parallel, the concurrent updates will corrupt the archive file.

```
.c.a:
    $(COMPILE.c) -o $% $<
    $(AR) $(ARFLAGS) $@ $%
    $(RM) $%
```

A better method is to build each object file and then archive all the object files after completion of the builds. An appropriate suffix rule and the corresponding library rule are:

```
.c.a:
    $(COMPILE.c) -o $% $<

lib.a: lib.a($(OBJECTS))
    $(AR) $(ARFLAGS) $(OBJECTS)
    $(RM) $(OBJECTS)
```

Multiple Targets

Another form of concurrent file update occurs when the same rule is defined for multiple targets. An example is a `yacc`(1) program that builds both a program and a header for use with `lex`(1). When a rule builds several target files, it is important to specify them as a group using the + notation. This is especially so in the case of a parallel build.

```
y.tab.c y.tab.h: parser.y
    $(YACC.y) parser.y
```

This rule is actually equivalent to the two rules:

```
y.tab.c: parser.y
    $(YACC.y) parser.y
y.tab.h: parser.y
    $(YACC.y) parser.y
```

The serial version of `make` builds the first rule to produce `y.tab.c` and then determines that `y.tab.h` is up-to-date and need not be built. When building in parallel, DMake checks `y.tab.h` before `yacc` has finished building `y.tab.c` and notices that it *does* need to be built, it then starts another `yacc` in parallel with the first one. Since both `yacc` invocations are writing to the same files (`y.tab.c` and `y.tab.h`), these files are apt to be corrupted and incorrect. The correct rule uses the + construct to indicate that both targets are built simultaneously by the same rule. For example:

```
y.tab.c + y.tab.h: parser.y
    $(YACC.y) parser.y
```

Restricting Parallelism

Sometimes file collisions cannot be avoided in a makefile. An example is xstr(1), which extracts strings from a C program to implement shared strings. The xstr command writes the modified C program to the fixed file x.c and appends the strings to the fixed file strings. Since xstr must be run over each C file, the following new .c.o rule is commonly defined:

```
.c.o:
    $(CC) $(CPPFLAGS) -E $*.c | xstr -c -
    $(CC) $(CFLAGS) $(TARGET_ARCH) -c x.c
    mv x.o $*.o
```

DMake cannot concurrently build targets using this rule since the build of each target writes to the same x.c and strings files, nor is it possible to change the files used. You can use the special target .NO_PARALLEL: to tell DMake not to build these targets in concurrently. For example, if the objects being built using the .c.o rule were defined by the OBJECTS macro, the following entry would force DMake to build those targets serially:

```
.NO_PARALLEL: $(OBJECTS)
```

If most of the objects must be built serially, it is easier and safer to force all objects to default to serial processing by including the .NO_PARALLEL: target without any dependents. Any targets that can be built in parallel can be listed as dependencies of the .PARALLEL: target:

```
.NO_PARALLEL:
.PARALLEL: $(LIB_OBJECT)
```

Nested Invocations of DistributedMake

When DMake encounters a target that invokes another DMake command, it builds that target serially, rather than concurrently. This prevents problems where two different DMake invocations attempt to build the same targets in the same directory. Such a problem might occur when two different programs are built concurrently, and each must access the same library. The only way for each DMake invocation to be sure that the library is up-to-date is for each to invoke DMake recursively to build that library. DMake only recognizes a nested invocation when the $(MAKE) macro is used in the command line.

If you nest commands that you know will not collide, you can force them to be done in parallel by using the `.PARALLEL:` construct.

When a makefile contains many nested commands that run concurrently, the load-balancing algorithm may force too many builds to be assigned to the local machine. This may cause high loads and possibly other problems, such as running out of swap space. If such problems occur, allow the nested commands to run serially.

How to Use DMake

You execute `dmake` on a *DMake host* and distribute jobs to *build servers*. You can also distribute jobs to the DMake host, in which case it is also considered to be a build server. DMake distributes jobs based on makefile targets that DMake determines (based on your makefiles) can be built concurrently. You can use any machine as a build server that meets the following requirements:

- From the DMake host (the machine you are using) you must be able to use `rsh`, without being prompted for a password, to remotely execute commands on the build server. See man `rsh`(1) or the system AnswerBook for more information about the `rsh` command. For example:

```
demo% rsh build_server which dmake
/opt/SUNWspro/bin/dmake
```

- The `bin` directory in which the DMake software is installed must be accessible from the build server. See the `share` (1M) and `mount` (1M) man pages or the system AnswerBook for more information.
- The `bin` directory in which the DMake software is installed must be in your execution path when you `rsh` to the build server. Be sure this directory is added to the PATH variable in your `.cshrc` file (or equivalent), ***not*** in your `.login` file. You can verify this as follows:

```
demo% rsh build_server which dmake
/opt/SUNWspro/bin/dmake
```

- The source hierarchy you are building must be:
  - accessible from the build server

- mounted under the same name

From the DMake host you can control which build servers are used and how many DMake jobs are allotted to each build server. The number of DMake jobs that can run on a given build server can also be limited on that server.

Notes

- If you specify the `-m` option with the "parallel" argument, or set the DMAKE_MODE variable or macro to the value "parallel," DMake does not scan your runtime configuration file. Therefore, you must specify the number of jobs using the `-j` option or the DMAKE_MAX_JOBS variable/macro. If you do not specify a value this way, a default of two jobs is used.
- If you modify the maximum number of jobs using the `-j` option, or the DMAKE_MAX_JOBS variable/macro when using DMake in distributed mode (DMake default, or specified either by option, variable or macro), the value you specify overrides the values listed in the runtime configuration file. The value you specify is used as the total number of jobs that can be distributed to all build servers.

Controlling DistributedMake Jobs

The distribution of DMake jobs is controlled in two ways:

1. A DMake user on a DMake host can specify the machines they want to use as build servers and the number of jobs they want to distribute to each build server.
2. The "owner" on a build server can control the maximum total number of DMake jobs that can be distributed to that build server. The owner is a user that can alter the `/etc/opt/SPROdmake/dmake.conf` file.

Note – If you access DMake from the GUI (Building) use the online help to know how to specify your build servers and jobs. If you access DMake from the CLI see the DMake man page (`dmake.1`).

Getting Help on the GUI or the CLI

DMake is fully implemented in both the GUI and CLI. To use DMake from the GUI, see the Sun WorkShop TeamWare online help.

To Access the Online Help

1. **Open any Sun WorkShop TeamWare GUI from the Sun WorkShop.**
2. **Or, open any Sun WorkShop TeamWare from the command line. For example, to open the Configuring GUI, enter the following:**

```
demo% twconfig &
```

3. **Open the pull-down menu from the Help button.**
4. **Click on Help Contents.**

To Access Help for the CLI

♦ **To access the manual page for information on how to use DMake from the CLI, enter the following. The manual page gives information on all command-line options, variables, and macros necessary to use DMake.**

```
demo% man dmake.1
```

Part 2 — Performance Tuning Multithreaded Programs

Multithreaded Concepts 2

This chapter provides a brief overview of Sun WorkShop multithreaded development tools, extending Sun's development tools and SPARCompiler language systems with support for development of high-performance MP/MT (multiprocessing/multithreaded) applications.

Sun WorkShop includes multithreaded extensions to the SPARCworks Debugger as well as multiprocessor extensions to the SPARCompiler Fortran and C language systems. Sun WorkShop includes three tools to help you develop MP/MT applications:

- LoopTool, a graphical user interface (GUI) application with full on-line help
- Thread Analyzer, which uses a GUI, but does not currently support on-line help
- LockLint, a command-line application, which is covered in *Sun WorkShop: Command Line Utilities*

This chapter contains the following sections:

Basic Concepts	*page 20*
The WorkShop MP/MT Solution	*page 20*
Other Sources of Information	*page 21*

2

Basic Concepts

Multiprocessing systems and multithreaded operating environments enable new levels of performance, responsiveness, and flexibility for applications that use parallelism in their implementation. Solaris user-level threads offer a powerful model for parallel, multithreaded applications that take advantage of SPARC multiprocessor and uniprocessor systems.

The benefits of threading your application include:

- Increased performance on multiprocessor systems (which are I/O and compute intensive)
- Increased performance on uniprocessor systems (which are I/O intensive)
- More efficient resource usage (for existing parallel applications that run in multiple processes)

The WorkShop MP/MT Solution

Sun WorkShop supports two approaches to MP/MT application development:

- Most C and C++ developers (and some Fortran developers) thread their applications by programming directly against Solaris user-level threads via the `libthread` library. With WorkShop's MP/MT applications, you can run the multithread extension to the debugger for debugging programs that use Solaris user-level threads. Use LockLint to statically check your program for consistent use of locks and potential race conditions. Use the Thread Analyzer to gather and display `prof` and `gprof` information for threaded (including auto-parallelized) programs.
- Many compute-intensive Fortran applications exhibit loop-level parallelism, where computations are performed over large datasets in loops. Sun WorkShop lets you use this parallelism through automatic parallelization of Fortran 77, Fortran 90, and C programs. Use the compiler to parallelize loops in your program. Use LoopTool to examine loops parallelized by the Fortran 77, Fortran 90, or C compiler.

Other Sources of Information

The following documents contain code examples as well as conceptual information about Solaris threads on Solaris 2, and are suggested reading for developers who are designing and writing MP/MT applications.

Document	**Author/Part Number**
Threads Primer: A Guide to Multithreaded Programming	p/n 801-3176-10
Implementing Lightweight Threads	D. Stein, D. Shah — Sunsoft, Inc USENIX—June 1992—San Antonio, TX
SunOS Multithreaded Architecture	M.L. Powell, S.R. Kleiman, S. Barton, D. Shah, D. Stein, M. Weeks — Sun Microsystems, Inc. USENIX—Winter 1991— Dallas, TX
Writing Multithreaded Code in Solaris	S. Kleiman, B. Smaalders, D. Stein, D. Shah — SunSoft, Inc.

Analyzing Loops 3

The Fortran MP and MP C compilers automatically parallelize loops for which they determine that it is safe and profitable to do so. LoopTool is a performance analysis tool that reads loop timing files created by these compilers. LoopTool has a graphical user interface (GUI); LoopReport (which is discussed in *Sun WorkShop: Command Line Utilities*) is the command-line version of LoopTool.

This chapter is organized as follows:

Basic Concepts	*page 23*
Setting Up Your Environment	*page 24*
Creating a Loop Timing File	*page 25*
Starting LoopTool	*page 26*
Using LoopTool	*page 27*
Other Compilation Options	*page 32*
Compiler Hints	*page 34*
Compiler Optimizations and How They Affect Loops	*page 38*

Basic Concepts

LoopTool's main features include the ability to:

- Time all loops, whether serial or parallel
- Produce a table of loop timings

- Collect hints from the compiler during compilation. These hints can help you parallelize loops that were not parallelized. Hints are described further in “Compiler Hints” on page 34.

LoopTool displays a graph of loop runtimes and shows which loops were parallelized. You can go directly from the graphical display of loops to the source code for any loop you want, so you can edit your source code while in LoopTool.

LoopReport is the command-line version of LoopTool. For more information about LoopReport, see *SunSoft WorkShop: Command Line Options*.

Using LoopTool is like using `gprof`. The three major steps are: compile, run, and analyze.

Note – The following examples use the Fortran MP (f77 and f90) compiler. The options shown (such as `-xparallel`, `-Zlp`) also work for MP C.

Setting Up Your Environment

1. **Before compiling, set the environment variable `PARALLEL` to the number of processors on your machine.**
   The following command makes use of `psrinfo`, a system utility. *Note the backquotes*:

```
% setenv PARALLEL `/usr/sbin/psrinfo | wc -l`
```

Note – If you have installed LoopTool in a nondefault directory, substitute that path for the one shown here.

---

2. **Before starting LoopTool, make sure the environment variable `XUSERFILESEARCHPATH` is set:**

```
% setenv XUSERFILESEARCHPATH \
    /opt/SUNWspro/lib/sunpro_defaults/looptool.res
```

3. **Set** LD_LIBRARY_PATH.
   If you are running Solaris 2.5:

```
% setenv LD_LIBRARY_PATH /usr/dt/lib:$LD_LIBRARY_PATH
```

   If you are running Solaris 2.3 or 2.4:

```
% setenv LD_LIBARY_PATH \
 /opt/SUNWspro/Motif_Solaris24/dt/lib:$LD_LIBRARY_PATH
```

You may want to put these commands in a shell startup file (such as .cshrc or .profile).

Creating a Loop Timing File

To compile for automatic parallelization, typical compilation switches are -xparallel and -xO4. To compile for LoopTool, add -Zlp, as shown in the following example:

```
% f77 -xO4 -xparallel -Zlp source_file
```

Note – All examples apply to Fortran 77, Fortran 90 and C programs.

For additional information, see “Loading a Timing File” on page 26.

There are a number of other useful options for looking at and parallelizing loops. Some of these options are shown in Table 3-1 below.

Table 3-1 Some Useful Compiler Options

Option	Effect
-o *program*	Renames the executable to *program*
-xexplicitpar	Parallelizes loops marked with DOALL pragma
-xloopinfo	Prints hints to stderr for redirection to files

For more information, see “Other Compilation Options” on page 32.

Run The Program

After compiling with `-Zlp`, run the instrumented executable. This creates the loop timing file, `program.looptimes`. LoopTool processes two files: the instrumented executable and the loop timing file.

Starting LoopTool

You can start LoopTool by giving it the name of a program (that is, an executable) to load:

```
% looptool program &
```

You can also start the tools with no files specified. In this case, LoopTool's file chooser comes up automatically so you can select a file to examine:

```
% looptool &
```

LoopReport is usually started like this:

```
% loopreport program &
```

Loading a Timing File

LoopTool reads the timing file associated with your program. The timing file contains information about loops. Typically, this file has a name of the format *program*.`looptimes` and is in the same directory as your program.

By default, LoopTool looks in the executable's directory for a timing file. Therefore, if the timing file is there (the usual case), you don't need to specify where to look for it:

```
% looptool program &
```

If you name a timing file on the command line, then LoopTool and LoopReport use it.

```
% looptool program program.looptimes &
```

If you use the command line option `-p`, LoopTool and LoopReport check for a timing file in the directory indicated by `-p`:

```
% looptool -p timing_file_directory program &
```

If the environment variable `LVPATH` is set, the tools check that directory for a timing file .

```
% setenv LVPATH timing_file_directory
% looptool program &
```

Using LoopTool

The Main Window

The main window displays the runtimes of your program's loops in a bar chart arranged in the order that the source files were presented to the compiler.

Figure 3-1 shows the components of the main window.

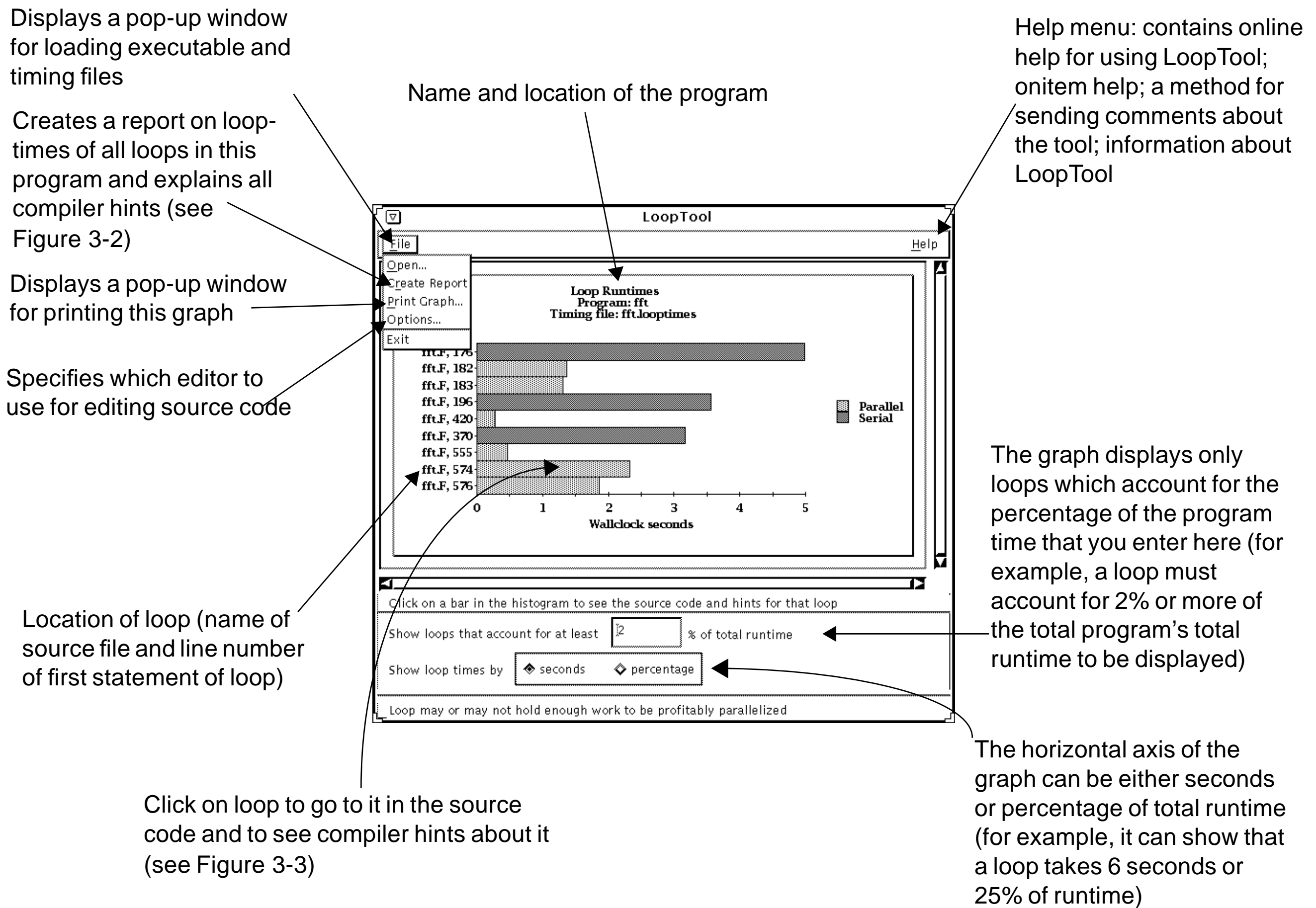


Figure 3-1 LoopTool Main Window

Opening Files

Choose File ➤ Open from the File menu in the main window to open executable and timing files.

There are two ways to specify the files you want to open:

- Type in the name of the files to open

- Bring up a file chooser.

Once you've typed in the executable's path, you don't need to type in the timing file, unless it's in a different directory or has a non-default name (or both).

For more information about opening files, see the LoopTool section of the WorkShop Online Help.

Creating a Report on All Loops

Choose File ➤ Create Report from the File menu in the main window to open a window with detailed information on all the loops in your program (see Figure 3-2). The Help button in the report window links to the WorkShop Online Help section containing compiler hints.

Explains the loop report and all compiler hints

Figure 3-2 LoopReport

Printing the LoopTool Graph

1. **Choose File ➤ Print Graph from the File menu in the main window to open the Print pop-up window.**

2. **Choose whether to print the graph of put it in a file.**
3. **Enter the name of the printer or filename where you want to send the graph.**

For more information about printing see the WorkShop Online Help.

Choosing an Editor

Choose File ➤ Options from the File menu in the main window to open the Options pop-up window.

The Options pop-up window lets you choose an editor for editing source code. The editors are `vi`, `gnuemacs`, and `xemacs`. See "Getting Hints and Editing Source Code" on page 30 for more on editing source code.

Note – `vi` and `xemacs` are installed with LoopTool into your install directory (usually `/opt/SUNWspro/bin`) if they're not already on your system. You must provide `gnuemacs` yourself. In all cases, the editor you want must be in a directory that's in your search path in order for LoopTool to find it. For example, your `PATH` environment variable should include `/usr/ucb` if that's where `vi` is located on your system.

For more information about choosing an editor see the WorkShop Online Help.

Getting Hints and Editing Source Code

Clicking a loop in the main window (Figure 3-1) does two things:

- It brings up a window in which you can edit your source code (Figure 3-3). The available editors are `vi`, `xemacs`, and `gnuemacs`. See "Choosing an Editor" on page 30 for more information on choosing an editor.

  For information on `vi`, see the `vi`(1) manual page. `xemacs` and `gnuemacs` have online help (click the Help button).

  The WorkShop `vi` editor has a special menu, Version, that allows you to make use of the SCCS (Source Code Control System) utility for sharing files. See the LoopTool online help, as well as the `sccs`(1) manual page, for more information.

3. It brings up a separate window that displays one or more hints about the loop you've selected. The Help button in this window displays the WorkShop online help compiler hints section. See also "Compiler Hints" on page 34, which explains the hints in detail.

Figure 3-3 shows the editor and hint windows:

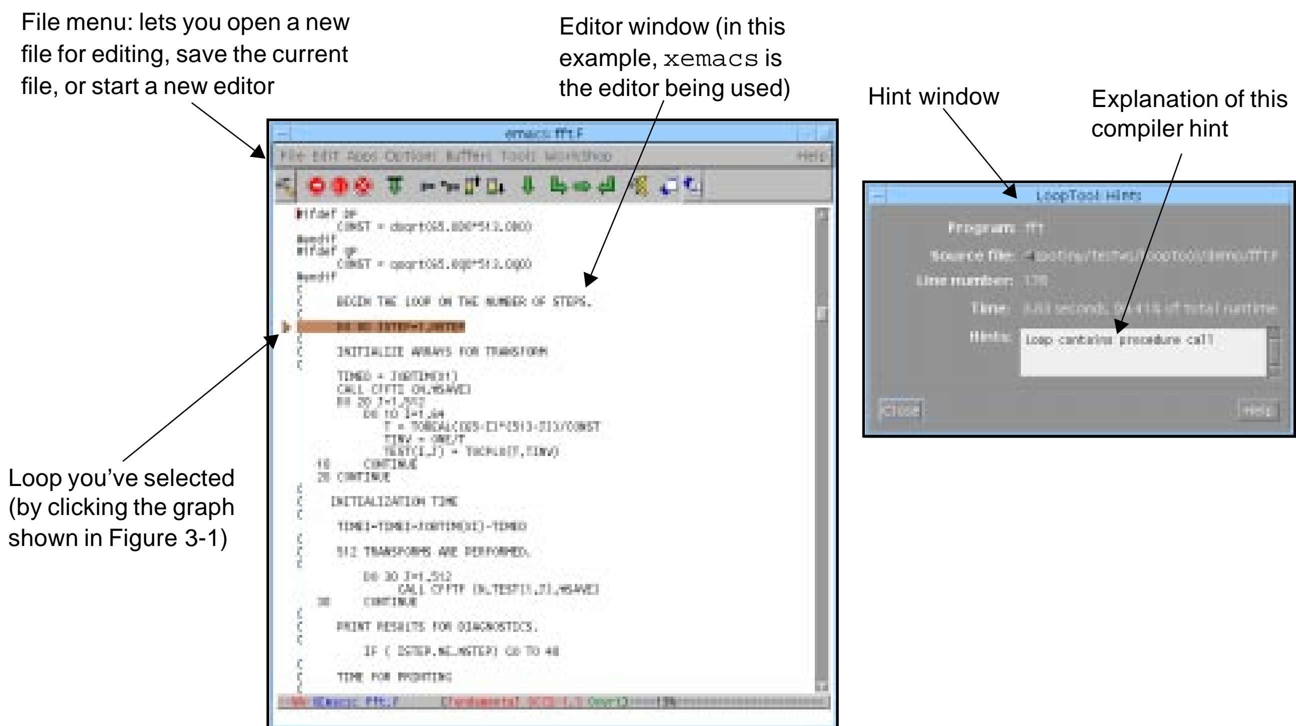


Figure 3-3 The Editor and Hints Windows

Warning – If you edit your source code, line numbers shown by LoopTool may become inconsistent with the source. You must save and recompile the edited source and then run LoopTool with the new executable, producing new loop information, for the line numbers to remain consistent.

Getting Help and Sending Comments

Choose from the Help menu (shown in Figure 3-1) to:

- See general help about starting and using LoopTool (Help Contents)
- Send comments about LoopTool (Send Comments)
- Get last-minute information (Release Notes)
- Invoke On Item Help (On Item)
- Access video demos of LoopTool and WorkShop features (Demos)
- Access WorkShop HTML documentation (WorkShop Manuals)

Other Compilation Options

Many combinations of compile switches work for LoopTool.

Either `-xO3` or `-xO4` can be used with `-xparallel`. If you don't specify `-xO3` or `-xO4` but you do use `-xparallel`, then `-xO3` is added. Table 3-2 summarizes how switches are added.

Table 3-2 Promotion of Compiler Switches

You type:	Bumped Up To:
`-xparallel`	`-xparallel -xO3`
`-xparallel -Zlp`	`-xparallel -xO3 -Zlp`
`-xexplicitpar`	`-xexplicitpar -xO3`
`-xexplicitpar -Zlp`	`-xexplicitpar -xO3 -Zlp`
`-Zlp`	`-xdepend -xO3 -Zlp`

Other compilation options include `-xexplicitpar` and `-xloopinfo.`

The Fortran MP compiler switch `-xexplicitpar` is used with the pragma `DOALL`. If you insert `DOALL` before a loop in your source code, you are explicitly marking that loop for parallelization. The compiler will parallelize this loop when you compile with `-xexplicitpar`.

The following code fragment shows how to mark a loop explicitly for parallelization.

```
    subroutine adj(a,b,c,x,n)
      real*8 a(n), b(n), c(-n:0), x
        integer n
c$par DOALL
    do 19 i = 1, n*n
      do 29 k = i, n*n
        a(i) = a(i) + x*b(k)*c(i-k)
29    continue
19  continue
    return
    end
```

When you use `-Zlp` by itself, `-xdepend` and `-xO3` are added. The switch `-xdepend` instructs the compiler to perform the data dependency analysis that it needs to do to identify loops. The switch `-xparallel` includes `-xdepend`, but `-xdepend` does not imply (or trigger) `-xparallel`.

The `-xloopinfo` option prints hints about loops to `stderr` (the UNIX standard error file, on file descriptor 2) when you compile your program. The hints include the routine names, the line number for the start of the loop, whether the loop was parallelized, and the reason it was not parallelized, if applicable.

The following example redirects hints about loops in the source file `gamteb.F` to the file `gamtab.loopinfo`:

```
% f77 -xO3 -parallel -xloopinfo -Zlp gamteb.F 2> gamteb.loopinfo
```

The main difference between `-Zlp` and `-xloopinfo` is that in addition to providing compiler hints about loops, `-Zlp` also instruments your program so that timing statistics are recorded at runtime. For this reason, also, LoopTool and LoopReport analyze only programs that have been compiled with `-Zlp`.

3

Compiler Hints

LoopTool and LoopReport present somewhat cryptic hints about the optimizations applied to a particular loop, and in particular, about why a particular loop may not have been parallelized. Some of the hints may seem to mean essentially the same thing.

Note – The hints are heuristics gathered by the compiler during the optimization pass. They should be understood in that context; they are *not* absolute facts about the code generated for a given loop. However, the hints are often very useful indications of how you can transform your code so that the compiler can perform more aggressive optimizations, including parallelizing loops.

For some useful explanations and tips, read the sections in the *Sun WorkShop Fortran: User's Guide* that address parallelization.

Table 3-3 lists the hints about optimizations applied to loops.

Table 3-3 LoopTool Hints

Hint #	Hint Definition
0	No hint available
1	Loop contains procedure call
2	Compiler generated two versions of this loop
3	Loop contains data dependency
4	Loop was significantly transformed during optimization
5	Loop may or may not hold enough work to be profitably parallelized
6	Loop was marked by user-inserted pragma, `DOALL`
7	Loop contains multiple exits
8	Loop contains I/O, or other function calls, that are not MT safe
9	Loop contains backward flow of control
10	Loop may have been distributed
11	Two or more loops may have been fused
12	Two or more loops may have been interchanged

0. No hint available
None of the other hints applied to this loop. This hint does not mean that none of the other hints might apply; it means that the compiler did not infer any of those hints.

1. Loop contains procedure call
The loop could not be parallelized since it contains a procedure call that is not MT safe. If such a loop were parallelized, multiple copies of the loop might instantiate the function call simultaneously, trample on each other's use of any variables local to that function, or trample on return values, and generally invalidate the function's purpose. If you are certain that the procedure calls in this loop are MT safe, you can direct the compiler to parallelize this loop no matter what by inserting the `DOALL` pragma before the body of the loop. For example, if `foo` is an MT-safe function call, then you can force it to be parallelized by inserting `c$par DOALL`:

```
c$par DOALL
     do 19 i = 1, n*n
         do 29 k = i, n*n
            a(i) = a(i) + x*b(k)*c(i-k)
            call foo()
 29    continue
 19  continue
```

The computer interprets the `DOALL` pragmas only when you compile with `-parallel` or `-explicitpar`; if you compile with `-autopar`, then the compiler ignores the `DOALL` pragmas.

2. Compiler generated two versions of this loop
The compiler couldn't tell at compile time if the loop contained enough work to be profitable to parallelize. The compiler generated two versions of the loop, a serial version and a parallel version, and a runtime check that will choose at runtime which version to execute. The runtime check determines the amount of work that the loop has to do by checking the loop iteration values.

3. **Loop contains data dependency**
   A variable inside the loop is affected by the value of a variable in a previous iteration of the loop. For example:

```
do 99 i=1,n
   do 99 j = 1,m
      a[i, j+1] = a[i,j] + a[i,j-1]
99 continue
```

   This is a contrived example, since for such a simple loop the optimizer would simply swap the inner and outer loops, so that the inner loop could be parallelized. But this example demonstrates the concept of data dependency, often referred to as “data-carried dependency.”

   The compiler will often be able to tell you the names of the variables that cause the data-carried dependency. If you rearrange your program to remove (or minimize) such dependencies, then the compiler will be able to perform more aggressive optimizations.

4. **Loop was significantly transformed during optimization**
   The compiler performed some optimizations on this loop that might make it almost impossible to associate the generated code with the source code. For this reason, line numbers may be incorrect. Examples of optimizations that can radically alter a loop are loop distribution, loop fusion, and loop interchange (see Hint 10, Hint 11, and Hint 12).

5. **Loop may or may not hold enough work to be profitably parallelized**
   The compiler was not able to determine at compile time whether this loop held enough work to warrant parallelizing. Often loops that are labeled with this hint may also be labeled “parallelized,” meaning that the compiler generated two versions of the loop (see Hint 2), and that it will be decided at runtime whether the parallel version or the serial version should be used.

   Since all the compiler hints, including the flag that indicates whether or not a loop is parallelized, are generated at compile time, there’s no way to be certain that a loop labeled “parallelized” actually executes in parallel. To determine whether a loop executes in parallel, you need to perform additional runtime tracing, such as can be accomplished with the Thread Analyzer. You can compile your programs with both `-Zlp` (for LoopTool)

and -Ztha (for Thread analyzer) and compare the analysis of both tools to get as much information as possible about your program's runtime behavior.

6. **Loop was marked by user-inserted pragma,** DOALL
   This loop was parallelized because the compiler was instructed to do so by the DOALL pragma. This hint is a useful reminder to help you easily identify those loops that you explicitly wanted to parallelize.

   The DOALL pragmas are interpreted by the compiler only when you compile with -parallel or -explicitpar; if you compile with -autopar, then the compiler will ignore the DOALL pragmas.

7. **Loop contains multiple exits**
   The loop contains a GOTO or some other branch out of the loop other than the natural loop end point. For this reason, it is not safe to parallelize the loop, since the compiler has no way of predicting the loop's runtime behavior.

8. **Loop contains I/O, or other function calls, that are not MT safe**
   This hint is similar to Hint 1; the difference is that this hint often focuses on I/O that is not MT safe, whereas Hint 1 can refer to any sort of MT-unsafe function call.

9. **Loop contains backward flow of control**
   The loop contains a GOTO or other control flow up and out of the body of the loop. That is, some statement inside the loop appears to the compiler to jump back to some previously executed portion of code. As with the case of a loop that contains multiple exits, this loop is not safe to parallelize.

   If you can reduce or minimize backward flows of control, the compiler will be able to perform more aggressive optimizations.

10. **Loop may have been distributed**
    The contents of the loop may have been distributed over several iterations of the loop. That is, the compiler may have been able to rewrite the body of the loop so that it could be parallelized. However, since this rewriting takes place in the language of the internal representation of the optimizer, it's very difficult to associate the original source code with the rewritten version. For this reason, hints about a distributed loop may refer to line numbers that don't correspond to line numbers in your source code.

11. **Two or more loops may have been fused**
    Two consecutive loops were combined into one, so the resulting larger loop contains enough work to be profitably parallelized. Again, in this case, source line numbers for the loop may be misleading.

12. **Two or more loops may have been interchanged**
    The loop indices of an inner and an outer loop have been swapped, to move data dependencies as far away from the inner loop as possible, and to enable this nested loop to be parallelized. In the case of deeply nested loops, the interchange may have occurred with more than two loops.

Compiler Optimizations and How They Affect Loops

As you might infer from the descriptions of the compiler hints, associating optimized code with source code can be tricky. Clearly, you would prefer to see information from the compiler presented to you in a way that relates as directly as possible to your source code. Unfortunately, the compiler optimizer "reads" your program in terms of its internal language, and although it tries to relate that to your source code, it is not always successful.

Some particular optimizations that can cause confusion are described in the following sections.

Inlining

Inlining is an optimization applied only at optimization level `-O4` and only for functions contained with one file. That is, if one file contains 17 Fortran functions, 16 of those can be inlined into the first function, and you compile at `-O4`, then the source code for those 16 functions may be copied into the body of the first function. Then, when further optimizations are applied, it becomes difficult to determine which loop on which source line number was subjected to which optimization.

If the compiler hints seem particularly opaque, consider compiling with `-O3 -parallel -Zlp`, so that you can see what the compiler says about your loops before it tries to inline any of your functions.

In particular, "phantom" loops—that is, loops that the compiler claims exist, but you know do not exist in your source code—could well be a symptom of inlining.

Loop Transformations—Unrolling, Jamming, Splitting, and Transposing

The compiler performs many loop optimizations that radically change the body of the loop. These include optimizations, unrolling, jamming, splitting, and transpositing.

LoopTool attempts to provide hints that make as much sense as possible, but given the nature of the problem of associating optimized code with source code, the hints may be misleading. For more information on what optimizations do for your code, refer to compiler books such as *Compilers: Principles, Techniques and Tools* by Aho, Sethi and Ullman.

Parallel Loops Nested Inside Serial Loops

If a parallel loop is nested inside a serial loop, the runtime information reported by LoopTool and LoopReport may be misleading because each loop is stipulated to use the wall-clock time of each of its loop iterations. If an inner loop is parallelized, it is assigned the wall-clock time of each iteration, although some of those iterations are running in parallel.

However, the outer loop is assigned only the runtime of its child, the parallel loop, which will be the runtime of the longest parallel instantiation of the inner loop. This double timing leads to the anomaly of the outer loop apparently consuming less time than the inner loop.

Analyzing Threads

Thread Analyzer is a tool for viewing trace information. It displays `gprof` and `prof` tables on a per-thread basis.

This chapter describes what you need to do to prepare your program for analysis, how to navigate the Thread Analyzer glyph hierarchy, and how to use Thread Analyzer's menu-driven interface.

Basic Concepts	*page 42*
Compiling and Instrumenting the Source	*page 43*
Running an Experiment	*page 44*
Starting Thread Analyzer	*page 44*
Exiting Thread Analyzer	*page 46*
Loading a Trace Directory	*page 46*
Navigating the Thread Analyzer Glyph Hierarchy	*page 47*
Thread Analyzer Menus	*page 48*
Thread Analyzer Usage Scenarios	*page 53*

Warning – If you are using C++ to create shared libraries, then you need to be careful not to use `-Ztha` on source files containing static constructors. These constructors will end up trying to record data before the data collection code has been initialized, causing your program to seg fault. Unfortunately, the most common thing that creates static constructors is including `<stream.h>`.

4

Basic Concepts

Thread Analyzer displays standard profiling information for each thread in your program, as well as metrics specific to a particular thread such as Mutex wait time and semaphore wait time.

Thread Analyzer displays:

- Tables of metrics
- Graphs of these metrics

Metrics Collected by Thread Analyzer

Metrics apply to objects. An object can be the entire program, a single thread, or a single function. If a metric refers to a single function, it refers to the total number of times that function was called by a particular thread. Function calls are implicitly divided up according to the thread that made the call.

Note – Thread Analyzer does not support metrics for forked programs.

Table 4-1 briefly describes each metric collected by Thread Analyzer.

Table 4-1 Metrics Collected by Thread Analyzer

Metric	Description
`CPU Time`	Measures the amount of time an object was scheduled by the operating system and was running on a CPU.
`Wall Clock Time`	Measures the amount of time between when an object was created and when it was destroyed. The obvious difference from CPU time is that when a thread is suspended (for instance, waiting for a mutex), the thread still exists even though it isn't taking up CPU time.
`Mutex Wait Time`	Measures the wall-clock time an object spends suspended waiting to acquire a mutex lock.
`Join Wait Time`	Measures the wall-clock time an object spends suspended in the `thr_join` function waiting for another thread to terminate.
`Semaphore Wait Time`	Measures the wall-clock time an object spends suspended waiting to acquire a semaphore.
`Condition Variable Wait Time`	Measures the wall-clock time an object spends suspended waiting to be signaled on a condition variable.

Table 4-1 Metrics Collected by Thread Analyzer *(Continued)*

Metric	Description
`RW Read Lock Wait`	Measures the wall-clock time an object spends suspended waiting to acquire a read lock on a Reader/Writer style lock.
`RW Write Lock Wait`	Measures the wall-clock time an object spends suspended waiting to acquire a write lock on a Reader/Writer style lock.
`Total Sync Wait Time`	Measures the total wall-clock time spent waiting on any of the six forms of threads synchronization.
`Read Wait Time`	Measures the amount of time spent blocked on read system calls.
`File Write (bytes)`	Measures the number of bytes written per second by write system calls.
`File Writes (ops)`	Measures the number of write system calls made per second by a certain object.
`File Reads (bytes)`	Measures the number of bytes read per second by read system calls.
`File Reads (ops)`	Measures the number of read system calls made per second by a certain object.
`File IO (bytes)`	Measures the number of bytes read or written per second by an object via the read and write system calls.
`File IO (ops)`	Measures the combined number of read and write system calls made by an object per second.

Compiling and Instrumenting the Source

Compile your program with the `-Ztha` option. This option instruments your program; that is, it inserts instrumentation points at the beginning and end of each function.

Note – If you execute an instrumented program that calls `fork`, the forked process does not contribute trace data until it calls `exec`.

The `-Ztha` option instruments C++ and Fortran programs as well as C programs. The C syntax looks like this:

```
% cc -Ztha -o prog1 prog1.c
```

Note – Do not use `-Ztha` on files containing definitions of the C++ generic “new” operator, signal handlers, `malloc`, or `free`.

Running an Experiment

Run the executable file to write trace data to files in a `tha.`*pid* directory, where *pid* is the unique process id of the particular invocation of the object.

Caution – Do not use Control-C to stop a running instrumented program. Control-C can leave the trace directory in an inconsistent state. A subsequent invocation of Thread Analyzer on that trace directory might not process its trace data.

Starting Thread Analyzer

To analyze the trace data collected for your program, start Thread Analyzer from the command line:

```
% tha &
```

Optionally, you can direct Thread Analyzer to load a `tha.`*pid* trace data directory from the command line.

Note – Thread Analyzer expects to find the executable in the same directory as the trace directory. There are known problems related to correctly finding the executable associated with a trace directory. If you encounter problems in this area, use the `-exec` command-line option to specify the path name to the executable.

You can use Thread Analyzer’s `-exec` option, in conjunction with the SunOS `ls` command, to specify a relative path name or full path name to the executable to be used in conjunction with a particular trace data directory.

The following examples show two variations of specifying a full path name for the executable. In each case, `tha.1234` is the directory containing the trace information files.

```
% tha -exec /bin/ls tha.1234 &
```

is equivalent to

```
% tha tha.1234 -exec /bin/ls &
```

The following example shows how to specify a relative path name:

```
% tha -exec ls tha.1234 &
```

Figure 4-1 shows Thread Analyzer's main window:

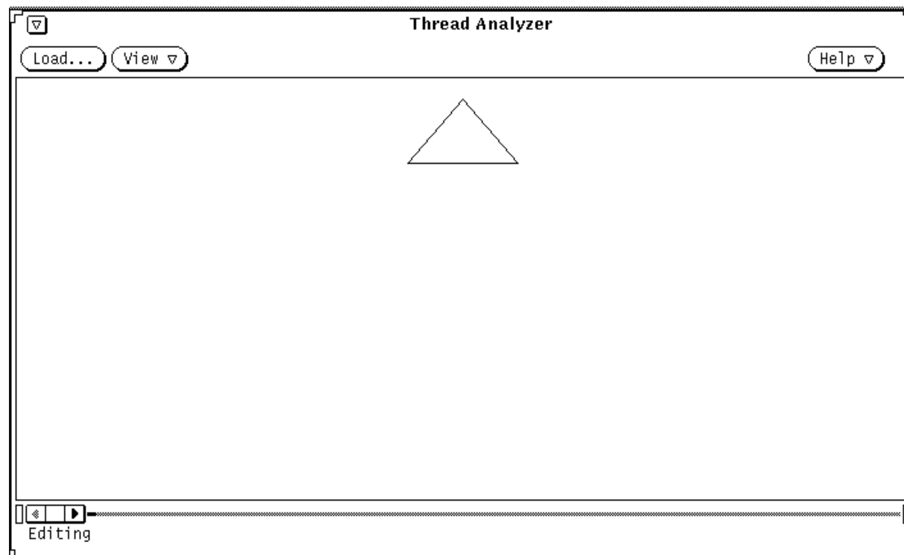


Figure 4-1 Thread Analyzer Main Window

This window displays a hierarchy of glyphs representing the threads and functions that make up a program. The glyph hierarchy is described in "Navigating the Thread Analyzer Glyph Hierarchy" on page 47.

The Load button loads a trace directory. The View menu allows you to choose the metric you are interested in and whether to display the information as a table or graph. For more information see "Load Button" and "View Button" on page 49.

Exiting Thread Analyzer

From the Window Manager, select ➤ Quit to exit the Thread Analyzer session.

Loading a Trace Directory

If you did not specify the `tha.`***pid*** trace data file in the command line, use the Load button to display a window, shown in Figure 4-2, where you can specify the path to your trace data directory.

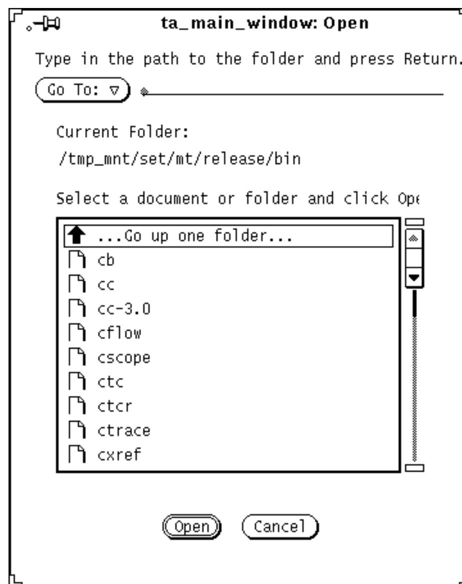


Figure 4-2 Thread Analyzer Load File Window

Note – If you load a trace file with the command line, Thread Analyzer makes the Load button unavailable; that is, Thread Analyzer loads only one trace directory per session.

Type the path name of the trace directory generated by the instrumented program, or click the icon of the desired trace directory in the displayed list. Then click Open. If Thread Analyzer does not recognize the path name you type, check for typographic errors and retype the path name.

Navigating the Thread Analyzer Glyph Hierarchy

Thread Analyzer glyphs are ordered as shown in Figure 4-3.

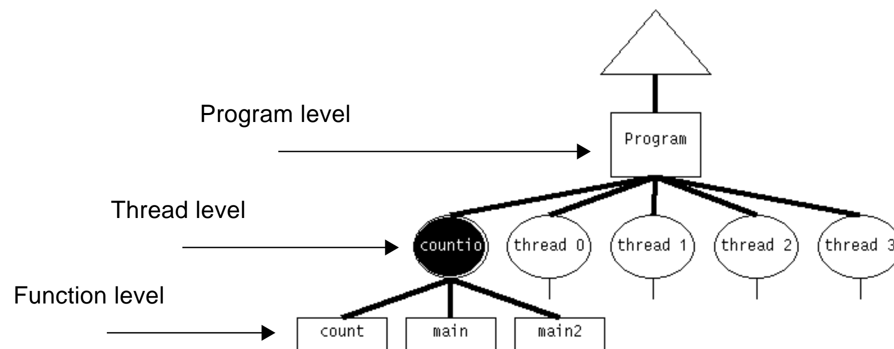


Figure 4-3 The Thread Analyzer Glyph Hierarchy

The *program glyph* is the root of the hierarchy.

The *thread level* is next in the hierarchy. All threads are displayed across the width of the main window. To see obscured threads, move the slider along the horizontal scrollbar located at the bottom of the window. (See Figure 4-4.)

The *function level* is the bottom of the tree. Thread Analyzer displays the function (thread children) glyphs in alphabetical order, from left to right. The children of a given thread are the procedures called within that thread. Use the scrollbar to scan function glyphs hidden from view (see Figure 4-4).

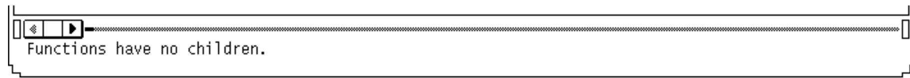


Figure 4-4 Scrollbar and Footer Help

To select a particular glyph, place the pointer over that glyph and click the SELECT mouse button. The selected glyph becomes highlighted.

To expand the glyph hierarchy (that is, show a node's children) or collapse the glyph hierarchy (that is, hide a node's children), place the pointer over the desired glyph and click the MENU mouse button. The MENU mouse button acts as a toggle to alternate between expand and collapse.

To hide an individual node, place the pointer over the desired glyph and click the ADJUST mouse button.

As shown in Figure 4-4, Thread Analyzer displays error messages in the footer of the main canvas.

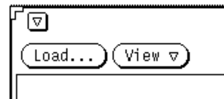
The thread that contains `main()` is named with the program name; all other threads are named numerically in order of creation.

Thread Analyzer Menus

Thread Analyzer uses the following conventions for menus:

- An inverted triangle (▼) in a button indicates there are more entries to be revealed. Place the pointer over the glyph and click the MENU mouse button to reveal the entries.
- Checkboxes (❐) in property sheets allow you to select items from a list of properties. Click the SELECT mouse button to select or unselect an item.
- Buttons allow you to specify an action such as Apply or Cancel. Click the SELECT mouse button to activate.
- Menus are pinnable; that is, they stay up if you pin them to the canvas.

The Load and View buttons are the main menu buttons.



Load Button

The Load button allows you to specify the directory that contains the trace data file. See "Loading a Trace Directory" on page 46.

View Button

Use the View menu to specify the particular metric you are interested in and whether you wish to see it as a table or graph. You can apply a CPU time filter to the glyph display to show only those threads or functions that exceed or equal a particular threshold.

To open the View Menu (see Figure 4-5), click the View button. You can also open the menu with the mouse MENU button and use the pushpin to pin the menu to the canvas.

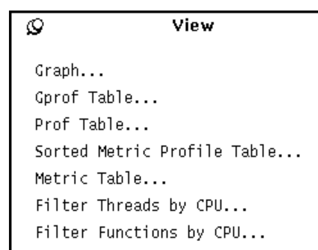


Figure 4-5 Thread Analyzer View Menu

Metric Graph Property Sheet

Graphs plot the value of a metric against wall-clock time.

1. **To display the Metric Graph Property Sheet (see Figure 4-6), choose View ➤ Graph from the View menu.**

2. **Click the SELECT mouse button when the cursor is over the checkboxes for the particular metrics you want to graph.**

3. **Click Apply to generate the graph.**

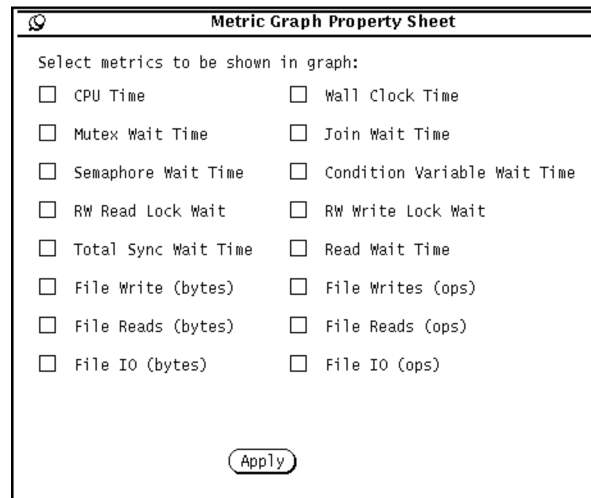


Figure 4-6 Thread Analyzer Metric Graph Property Sheet

Note – Thread Analyzer allows you to display a maximum of ten graphs at a time and emits an audible beep if you attempt to simultaneously display more than the ten-graph maximum.

See “Scenario 2: Identifying Initial Bottlenecks From Graphical Data” on page 60 for an example of how to display CPU time versus wall-clock time.

`gprof` *Table*

`gprof` tables display call-graph profile data in the form of a list of functions called by threads.

To generate a `gprof` table for a particular level in the hierarchy, select a glyph at that particular level. Then choose View ➤ Gprof Table from the View menu.

See page 56 for an example of how to display a `gprof` table.

`prof` *Table*

`prof` tables display profile data for the program, threads, and functions. The program-level `prof` table shows the total time for the entire program. The thread-level `prof` table has an entry for each thread (with a total of all function calls made by the thread). The function-level `prof` table has an entry for each thread-function combination.

To generate a `prof` table for a particular level in the hierarchy, select a glyph at that particular level. Then choose View ➤ Prof Table from the View menu.

See page 55 for an example of how to display a `prof` table.

Sorted Metric Profile Table Property Sheet

A sorted metric profile table shows a designated metric for all nodes for a particular glyph level.

To display the sorted metric profile table property sheet, select a glyph from the desired level in the hierarchy. Then choose View ➤ Sorted Metric Profile Table from the View menu.

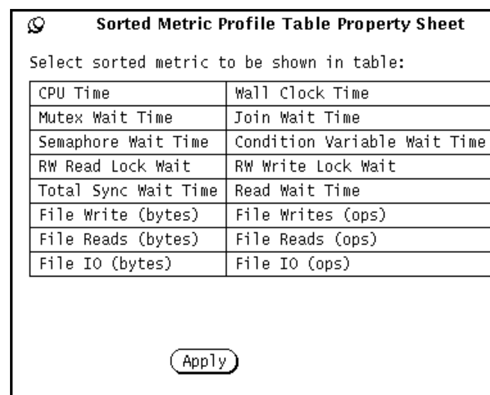


Figure 4-7 Thread Analyzer Sorted Metric Profile Table Property Sheet

Click the SELECT mouse button when the cursor is over the metric you want to graph—you may select only one from this list. Grayed-out entries signify metrics that are not available. Click Apply to generate the table.

See page 59 for an example of how to generate a sorted metric table.

Metric Table

A metric table shows multiple metrics for a particular thread or function. For a function, the metrics are shown for calls of that function made by the thread that is that function's parent.

To display the metric table property sheet, select a glyph from the desired level in the hierarchy. Then choose View ➤ Metric Table from the View menu.

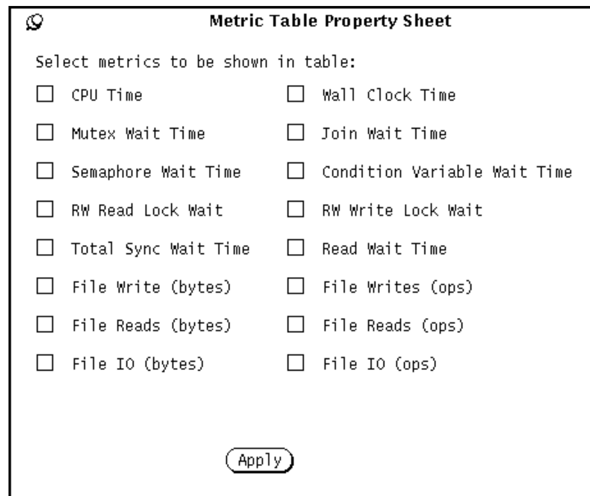


Figure 4-8 Thread Analyzer Metric Table Property Sheet

Click the SELECT button when the cursor is over the checkbox for the metrics you want to graph. Then click Apply to generate the table.

See page 58 for an example of how to generate a metric table.

Filter Threads by CPU

The filter shown in Figure 4-9 displays the thread glyphs whose percent of CPU time is equal to or greater than a designated threshold.

Figure 4-9 Thread Analyzer: Filter Threads by CPU Time

Type the threshold value and click Apply.

See page 63 for an example of how to filter threads by CPU time.

Filter Functions by CPU

The filter shown in Figure 4-10 displays the function glyphs whose percent of CPU time is equal to or greater than a designated threshold.

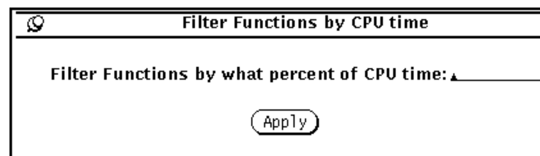


Figure 4-10 Thread Analyzer: Filter Functions by CPU Time

Type the threshold value and click Apply.

Thread Analyzer Usage Scenarios

This section contains scenarios demonstrating the use of Thread Analyzer.

- *Scenario 1: Initial Investigation —CPU Time Tabular Data*
- *Scenario 2: Identifying Initial Bottlenecks From Graphical Data*
- *Scenario 3: Identifying Focused Bottlenecks From Graphical Data*
- *Scenario 4: CPU Time Filter*

Scenario 1: Initial Investigation —CPU Time Tabular Data

The initial investigation gathers an overview of what your program is doing.

Looking at CPU time is a good starting point to discover where your program is spending its time.

1. **Display the thread-level nodes.**
   If the thread-level nodes are not already displayed, place the cursor over the Program node and click the mouse MENU button to expand the glyph display to the thread-level nodes (see Figure 4-11).

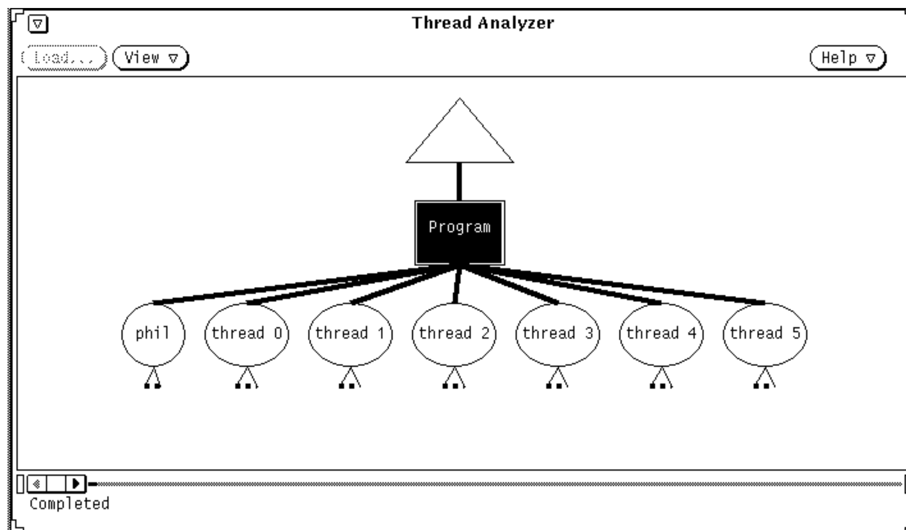


Figure 4-11 Displaying Thread-Level Nodes

2. **Request a `prof` table for all threads.**
   Select any thread glyph. Choose View ➤ Prof Table from the View menu to display `prof` information for all the threads in your program (see Figure 4-12).

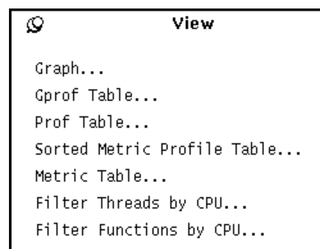
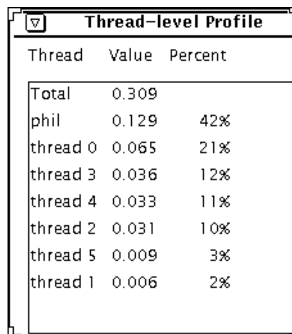


Figure 4-12 `prof` Information for Threads

3. **Analyze the `prof` table data.**
   The `prof` table shows the percent of CPU time spent by all threads of the program. Table elements are arranged in descending order of CPU time.



Thread-level Profile

Thread	Value	Percent
Total	0.309	
phil	0.129	42%
thread 0	0.065	21%
thread 3	0.036	12%
thread 4	0.033	11%
thread 2	0.031	10%
thread 5	0.009	3%
thread 1	0.006	2%

Figure 4-13 Thread-Level Profile

See if any threads consume a disproportionate amount of CPU time. In Figure 4-13 the `phil` thread accounts for 42 percent of the CPU time, and is a likely candidate for further investigation.

4. **Focus on a particular thread.**
   Move the cursor to the `phil` node and click the mouse MENU button; this action causes the children (function nodes) of the `phil` node to be displayed as shown in Figure 4-14.

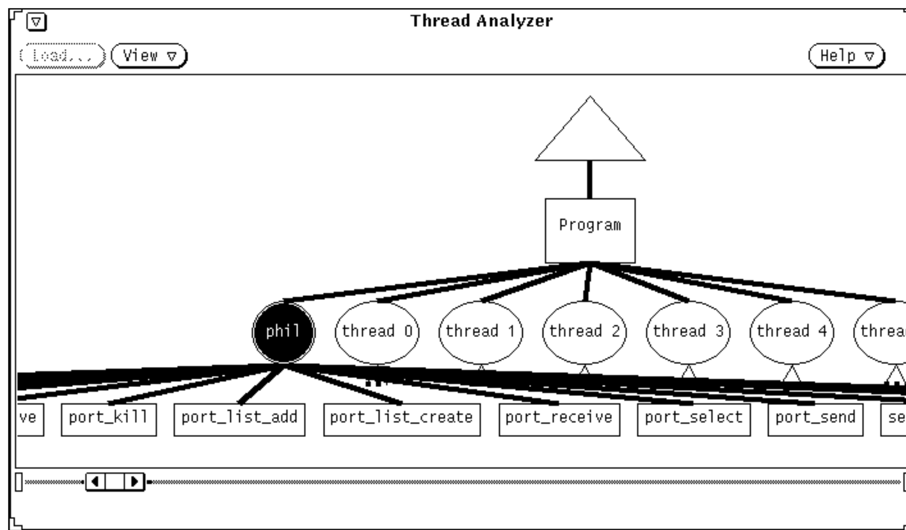


Figure 4-14 Focus on a Particular Thread

5. **Request a `gprof` table for a particular thread.**
   To generate a `gprof` table for a particular thread—`phil` is still selected—choose View ➤ Gprof Table from the View menu.

   Thread Analyzer displays a thread-specific `gprof` table for the `phil` thread, as shown in Figure 4-15. See the `gprof(1)` manual page for more information.

Gprof Table : phil

View ▽

parents <cycle> name children <cycle>	%time	self	descendents	called/total called+self called/total
main	100.00	0.06	0.05	0
port_receive		0.01	0.00	28/28
port_select		0.01	0.00	28/28
utensiles_request		0.00	0.01	14/14
utensiles_return		0.00	0.01	10/10
port_is_active		0.00	0.00	32/32
port_kill		0.00	0.00	2/2
port_create		0.00	0.00	4/4
port_list_add		0.00	0.00	2/2
port_list_create		0.00	0.00	1/1
fum2		0.00	0.00	1/1
fum1		0.00	0.00	1/1
utensiles_return		0.00	0.01	10/24
utensiles_request		0.00	0.01	14/24
send_to_philosopher	14.68	0.00	0.01	24
port_send		0.01	0.00	24/24
send_to_philosopher		0.01	0.00	24/24
port_send	12.03	0.01	0.00	24
port_getmsg		0.00	0.00	24/24
main		0.01	0.00	28/28
port_receive	11.16	0.01	0.00	28
port_freemsg		0.00	0.00	28/29
main		0.01	0.00	28/28
port_select	10.23	0.01	0.00	28
main		0.00	0.01	14/14
utensiles_request	10.02	0.00	0.01	14
send_to_philosopher		0.00	0.01	14/24
main		0.00	0.01	10/10
utensiles_return	8.02	0.00	0.01	10

Figure 4-15 Thread Analyzer: `gprof` Table

Note – `called+self` shows direct recursion; `<cycle>` shows indirect recursion.

6. **Request a metric table for a particular function.**
   Select the glyph for a particular function. Then choose View ➤ Metric Table from the View menu. Click the CPU Time checkbox, then click Apply. You can select multiple metrics from this menu).

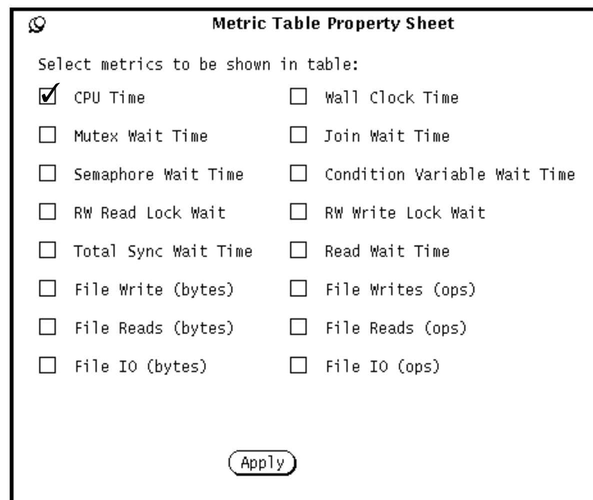
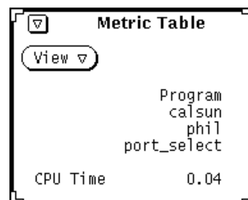
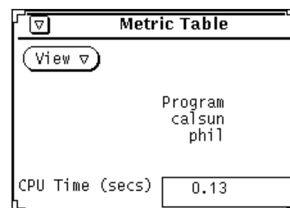


Figure 4-16 Thread Analyzer Metric Table Property Sheet

The resultant table shows the CPU time used by the `port_select` function in the `phil` thread.



To display a metric table for a thread, select its glyph in the hierarchy. Repeat the request for a metric table for CPU Time. The resultant table shows the CPU time used by the `phil` thread



7. **Request a sorted metric profile table for all threads.**
   Thread Analyzer displays a sorted metric profile table for a single metric for all functions in all threads (see Figure 4-17).

   Select any function in the glyph hierarchy. Then choose View➤ Sorted Metric Profile Table from the View menu. Select CPU Time in the property sheet, then click Apply.

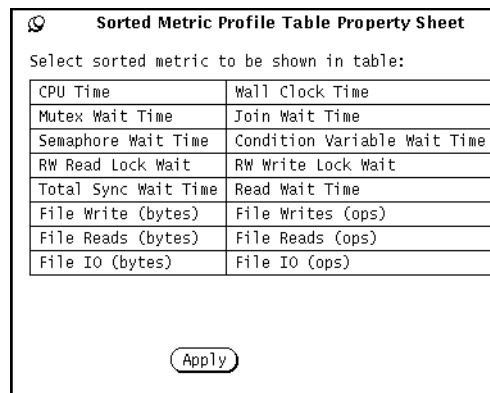


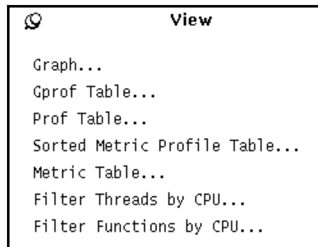
Figure 4-17 Sorted Metric Profile Table Property Sheet

Thread Analyzer displays CPU time for all functions in all threads. The table is sorted in descending order of percentage of time spent.

Scenario 2: Identifying Initial Bottlenecks From Graphical Data

Thread Analyzer can plot time graphs of the metrics to help discern the performance of your program. In particular, this feature helps discover performance bottlenecks. A common metric to view is the consumption of CPU time versus wall-clock time.

1. **Select the node of interest**
   Select the glyph for the thread to be analyzed. Then choose View ➤ Graph from the View menu.



2. **Designate the metric**
   Click the CPU Time checkbox and then click Apply (see Figure 4-18).

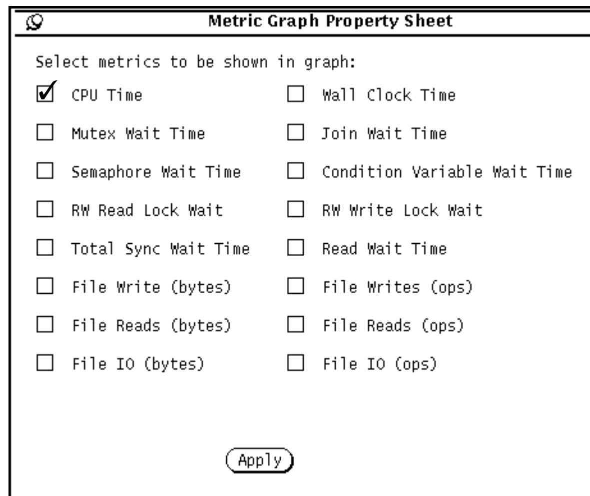


Figure 4-18 Metric Graph Property Sheet

Thread Analyzer plots percentage of CPU Time (y axis) versus wall-clock time (x axis) as shown in Figure 4-19.

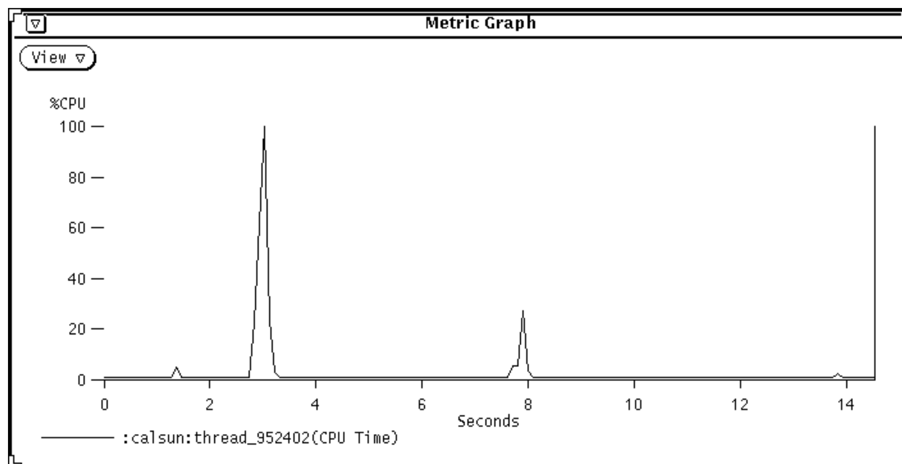


Figure 4-19 CPU Time versus Wall-clock Time

Except for a few “spikes,” a small amount of CPU time is being consumed. This fact suggests a bottleneck in I/O or thread synchronization (see “Scenario 3: Identifying Focused Bottlenecks From Graphical Data” next).

Scenario 3: Identifying Focused Bottlenecks From Graphical Data

Thread Analyzer graphs can plot the time spent waiting on a resource. Such graphs can help you diagnose performance bottlenecks caused by blocking on I/O or thread synchronization.

1. **Select the node of interest**
   Select the `phil` glyph in the hierarchy. Then choose View ➤ Graph from the View menu.

2. **Designate the metric**
   In the metric graph property sheet, click the checkbox for the metrics you wish to graph, then click Apply. In Figure 4-20, the synchronization-related metrics are selected.

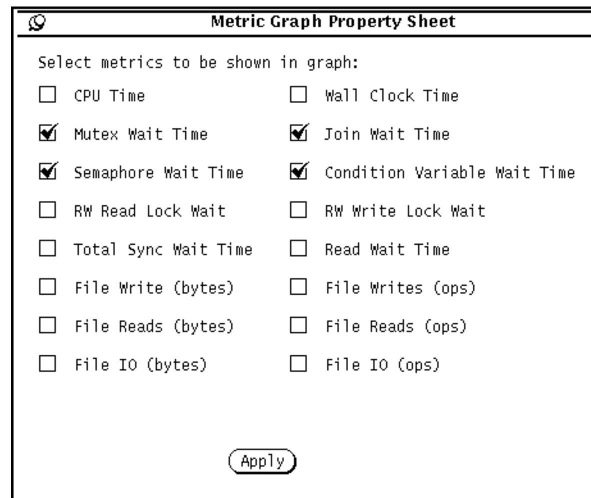


Figure 4-20 Metric Graph Property Sheet

Thread Analyzer plots percentage of time waiting for a resource (y axis) versus wall-clock time (x axis). Thread Analyzer shows each metric with a different line representation (see the legend at the bottom of the graph in Figure 4-21).

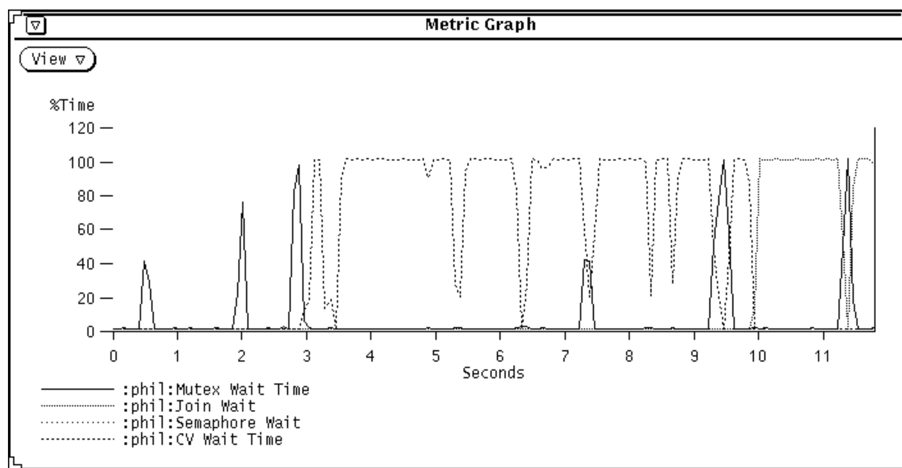


Figure 4-21 Percent of Resource-Waiting Time Versus Wall-clock Time

This graph shows that most of the time waiting for a resource was spent in CV (condition variable) wait time.

Scenario 4: CPU Time Filter

The filter threads by CPU operation modifies the glyph display to show only those threads that spend the designated percentage of total CPU time.

1. **Display the thread level nodes**
   Expand the glyph hierarchy to show all threads, as shown in Figure 4-22. Select a thread icon.

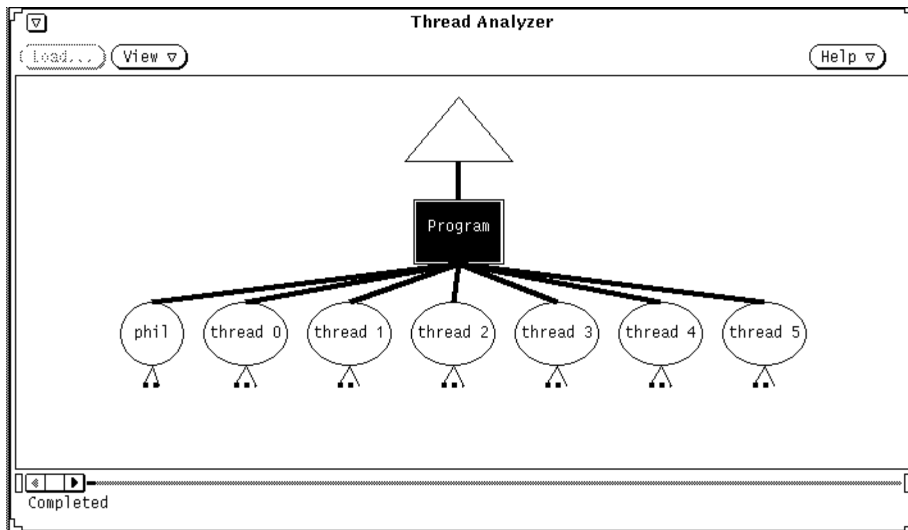


Figure 4-22 Expanded Glyph Hierarchy

2. **Specify the Filter Value**
   Choose View ➤ Filter Threads by CPU from the View menu.

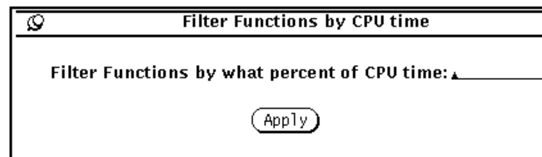


Figure 4-23 Filter Functions by CPU

In the box that appears (see Figure 4-23) type the threshold value (8 for our example) and click Apply.

The canvas collapses to show only those threads that account for at least 8 percent of the CPU time of the program run (see Figure 4-24).

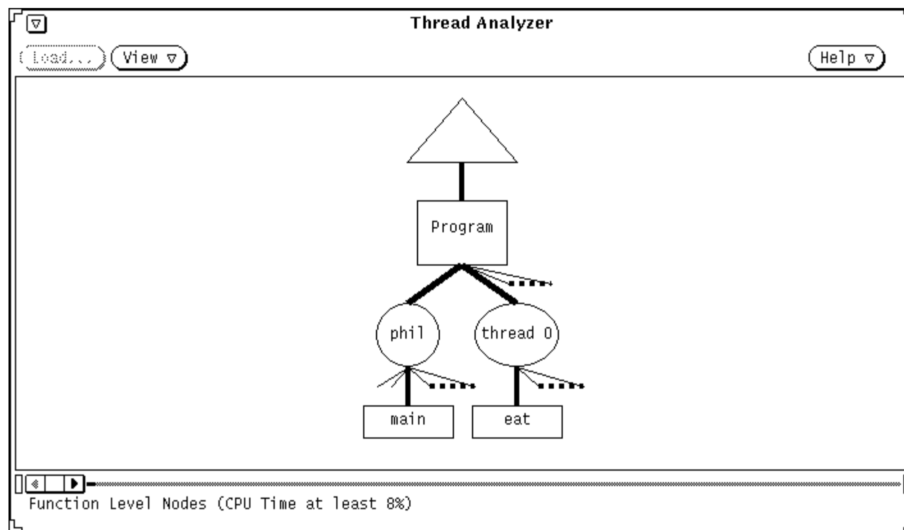


Figure 4-24 Threads Displayed Above CPU Time Threshold

Part 3 — Visualizing Data

Data Visualization 5

If you need a way to display your data graphically as you debug your program from Sun WorkShop, you can use Data Visualization.

This chapter contains the following sections:

Basic Concepts	*page 69*
Specifying Proper Array Expressions	*page 70*
Automatic Updating of Array Displays	*page 72*
Changing Your Display	*page 73*
Analyzing Data	*page 77*
Fortran Example Program	*page 80*
C Example Program	*page 81*

Basic Concepts

Data visualization can be used during debugging to help you explore and comprehend large and complex datasets, simulate results, or interactively steer computations. The Data Graph window gives you the ability to “see” program data and analyze that data graphically. The graphs can be printed out or printed to a file.

Specifying Proper Array Expressions

To display your data you must specify the array, and how it should be displayed. You can invoke the Data Graph window from the WorkShop Debugging window by typing an array name in the Expression text box. All scalar array types are supported except for complex (Fortran) array types.

Single-dimensional arrays are graphed (a vector graph) with the x-axis indicating the index and the y-axis indicating the array values. In the default graphic representation of a two-dimensional array (an area graph), the x-axis indicates the index of the first dimension, the y-axis indicates the index of the second dimension, while the z-axis represents the array values. You can visualize arrays of *n* dimensions, but at most, only two of those dimensions can vary.

You do not have to examine an entire dataset. You can specify slices of an array, as shown in the following examples. The figures show the `bf` array from the sample Fortran program given at the end of this chapter.

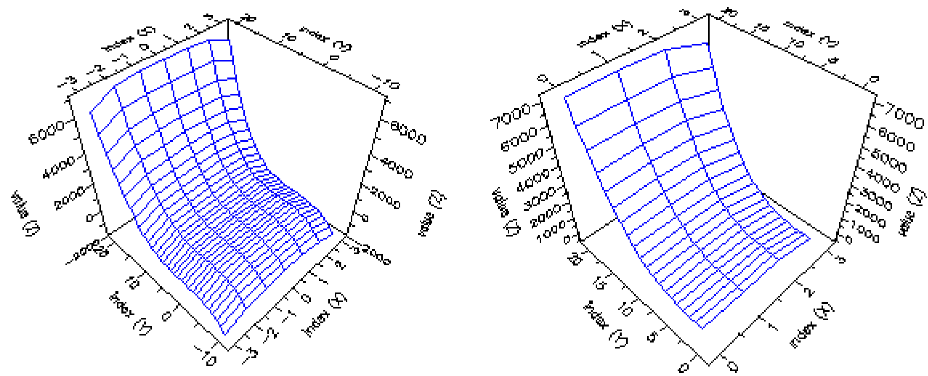


Figure 5-1 Graph of (left) bf array name only, (right) bf(0:3,1:20)

The next two figures show the array with a range of values

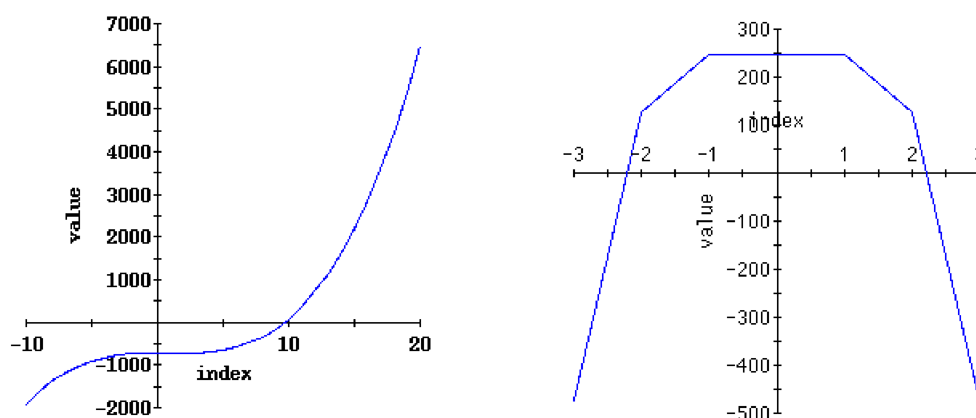


Figure 5-2 Graph of array `bf`: (left) `bf(-3:3,:)`, (right) `bf(:,-7:7)`

Graphing an Array

Before you can graph an array, you need to follow these preliminary steps:

1. **Load a program into the Debugging window.**
   Choose Debug ➤ New Program to load your program. If the program was previously loaded in the current WorkShop session, choose the program from the program list in the Debug menu.

2. **Set at least one breakpoint in the program.**
   You can set a single breakpoint at the end of the program or you can set one or more at points of interest in your program.

3. **Run your program.**
   When the program stops at the breakpoint, decide which array you want to examine.

Now you can graph the array. WorkShop provides multiple ways to graph an array through WorkShop:

From the Debugging window—you can enter an array in the Expression text field and choose Data ➤ Graph Expression or you can select an array in a text editor and choose Data ➤ Graph Selected in the Debugging window.

From the Data Display window—you can choose the Graph command from the Data menu or from the identical pop-up menu (right-click to open the pop-up). If the array in the Data Display window can be graphed, the Graph command is active.

From the Data Graph window—you can choose Graph ➤ New, enter an array name in the Expression text field in the Data Graph: New window and click Graph.

If you click the Replace current graph button, the new graph is replaces the current one. Otherwise, a new Data Graph window is opened.

From the Dbx Commands window—you can display a Data Grapher directly from the dbx command line with the `vitem` command (you must have opened the Dbx Commands window from WorkShop):

```
(dbx) vitem -new array-expr
```

array-expr specifies the array expression to be displayed. Additional options and arguments for the `vitem` command can be found in the manual, *WorkShop: Command Line Utilities*.

Automatic Updating of Array Displays

The value of an array expression can change as the program runs. You can choose whether to show the array expression's new values at specific points within the program or at fixed time intervals with the Update at field in the graph options.

If you want the values of an array updated each time the program stops at a breakpoint, you must turn on the Update at: Program stops option. As you step through the program, you can observe the change in array value at each stop. Use this option when you want to view the data change at each breakpoint. The default setting for this feature is off.

If you have a long-running program, choose the Update at: Fixed time interval option to observe changes in the array value over time. With a fixed time update, a timer is automatically set for every *n*th period. The default time is set for 1 second intervals. To change the default setting, choose Graph ➤ Default Options and change the time interval in the Debugging Options dialog box (make sure you are in the Data Grapher category).

Note – Every time the timer reaches the nth period, WorkShop tries to update the graph display; however, the array could be out of scope at that particular time and no update can be made.

Since the timer is also used in collecting data and when checking for memory leaks and access checking, you cannot use the Update at Fixed time interval setting when you are running the Behavior Data Collector or the run-time checking feature.

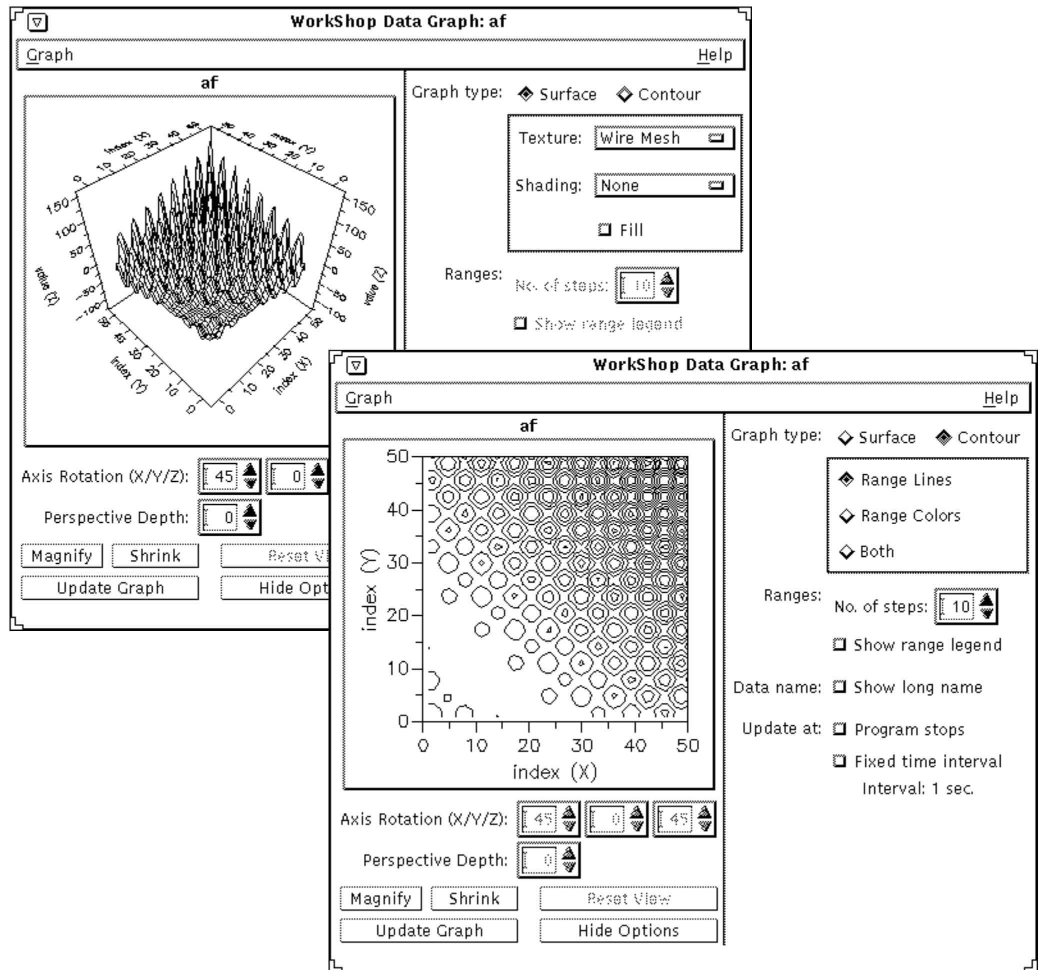
Changing Your Display

Once your data is graphically displayed, you can adjust and customize the display using the controls in the Data Graph window. This section presents examples of some of graph displays that you can examine.

The Data Graph window opens with the Magnify and Shrink options present. when graphing any type of array. With an area graph, the window includes the Axis Rotation and Perspective Depth fields. These options enable you to change your view of the graph by rotating its axis or increasing or decreasing the depth perspective. To quickly rotate the graph, hold the right mouse button down with the cursor on the graph and drag to turn the graph. You can also enter the degree of rotation for each axis in the Axis Rotation field.

Click the Show Options button for more options. If the options are already shown, click on the Hide button to hide the additional options.

The following figure shows two displays of the same array. The figure on the left shows the array with a Surface graph type with the Wire Mesh texture. The figure on the right shows the array with a Contour graph type delineated by range lines.



When you choose a Contour graph type, the range options enable you to see areas of change in the data values.

The display options for the Surface graph type are texture, shading, and fill. Texture choices include Wire Mesh or Range Lines as shown here:

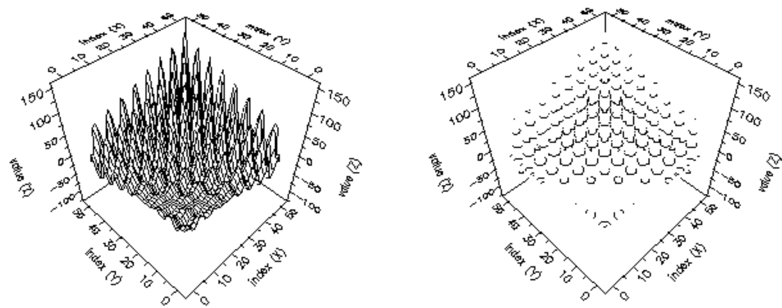
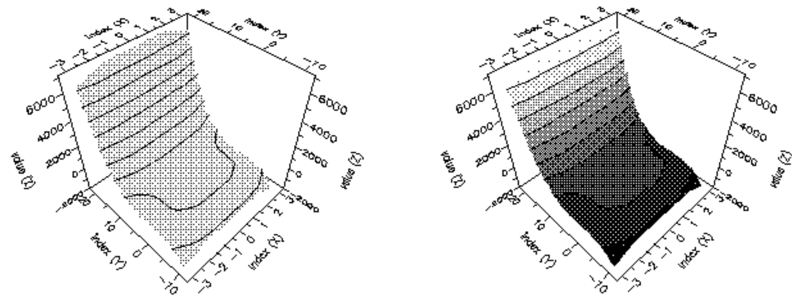
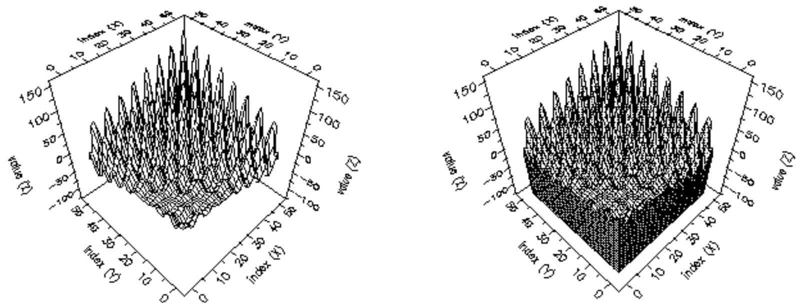


Figure 5-3 Surface graph with (left to right) Wire Mesh, Range Lines

Shading choices for the Surface graph type include Light Source or Range Colors:



Turn on Fill to shade in areas of a graph or to create a solid surface graph.



Choose Contour as the graph type to display an area graph using data range lines. With the Contour graph type, you have the additional options of displaying the graph in lines, in color, or both.

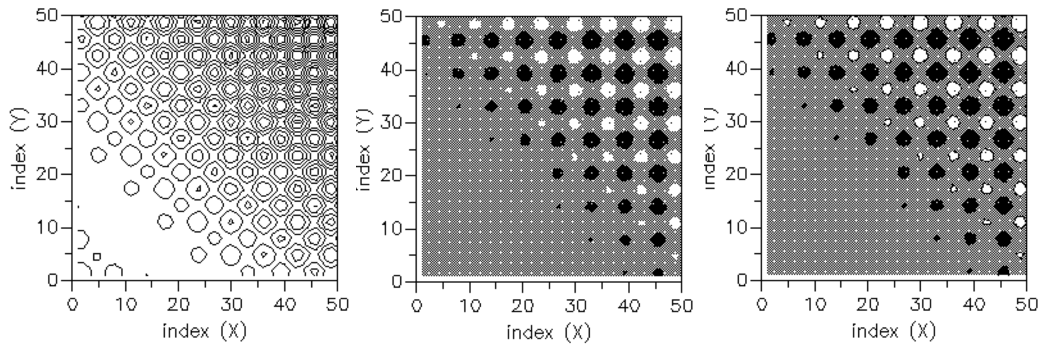


Figure 5-4 Contour graph with (left to right) Range Lines, Range Colors, Both

You can change the number of data value ranges being shown by changing the value in the Ranges: No. of steps.field.

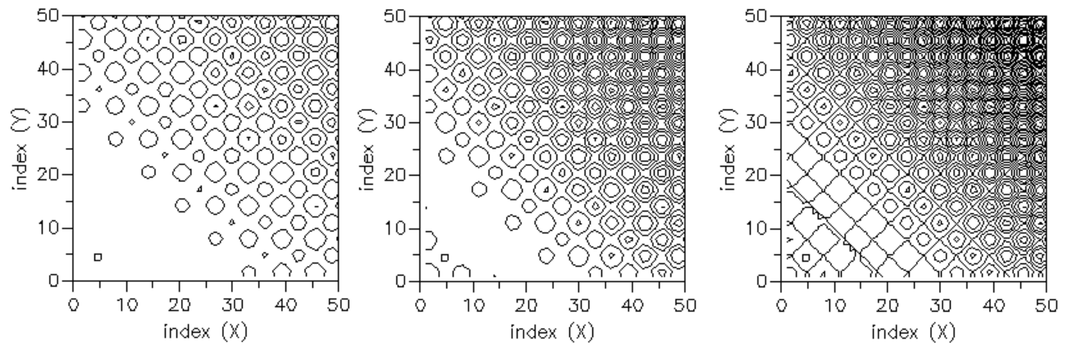


Figure 5-5 Number of steps (left to right): 5, 10, 15

Be aware that if you choose a large number of steps, you can adversely affect the color map.

Click on the Show range legend button if you want a legend to display your range intervals.

Analyzing Data

There are different ways to update the data being visualized, depending on what you are trying to accomplish. For example, you can update upon demand, at breakpoints, or at specified time intervals. You can observe changes or analyze final results. This section provides several scenarios illustrating different situations.

Two sample programs, `dg_fexamp` (Fortran) and `dg_cexamp` (C), are included with WorkShop. These sample programs are used in the following scenarios to illustrate data visualization.

You can also use these programs for practice. They are located in `/etc/opt/SUNWspro/WS4.0/examples`. To use the programs, `change` to this directory and type `make`, the executable programs are created for you.

Scenario 1: Comparing Different Views of the Same Data

The same data can be graphed multiple times which allows different graph types and different segments of data to be compared.

1. **Load the C or Fortran sample program.**
2. **Set a breakpoint at the end of the program.**
3. **Start the program, running the program to that breakpoint.**
4. **Type `bf` into the Expressions text field in the Debugging window.**
5. **Choose Data ➤ Graph Expression, which brings up a Surface graph of both dimensions of the `bf` array.**
6. **Choose Data ➤ Graph Expressions again to bring up a duplicate graph.**
7. **Click Show Options and select Contour for the graph type. Now you can compare different views of the same data (Surface vs. Contour).**
8. **Type `bf(1,:)` for the Fortran or `bf[1][..]` for the C example program into the Expression text field.**
9. **Now Choose Data ➤ Graph Expression to bring up a graph of a section of the data contained in the `bf` array.**

All these different views of the data can now be compared.

Scenario 2: Updating Graphs of Data Automatically.

The updating of a graph can be controlled automatically by turning on the Update at: Program stops option. This feature enables you to make comparisons of data as it changes during the execution of the program.

1. **Load the C or Fortran sample program.**
2. **Set a breakpoint at the end of the outer loop of the `bf` function.**
3. **Start the program, running the program to that breakpoint.**
4. **Type `bf` into the Expressions text field.**
5. **Choose Data ➤ Graph Expression. A graph of the values in the `bf` array after the first loop iteration appear.**
6. **Click Show Options and select Update At: Program stops.**
7. **Choose the Go command to cause the execution of several other loop iterations of the program. Each time the program stops at the breakpoint, the graph is updated with the values set in the previous loop iteration.**
8. **Utilizing the automatic update feature can save time when an up to date view of the data is desired at each breakpoint.**

Scenario 3: Comparing Data Graphs at Different Points in Program

The updating of a graph can also be controlled manually.

1. **Load the C or Fortran example program.**
2. **Set a breakpoint at the end of the outer loop of the `af` function.**
3. **Start the program, running the program to that breakpoint.**
4. **Type `af` into the Expression: text field.**
5. **Choose Data ➤ Graph Expression. A graph of the values in the `af` array after the first loop iteration appears. Make sure automatic updating is turned off on this graph (the default setting).**
6. **Execute another loop iteration of the program using the Go command.**

7. **Bring up another graph of the `af` array choosing Data ➤ Graph Expression. This graph contains the data values set in the second iteration of the outer loop.**
8. **The data contained in the two loop iterations of the `af` array can now be compared. Any graph with automatic updating turned off can be used as a reference graph to a graph that is continually being updated automatically or manually.**

Scenario 4: Comparing Data Graphs from Different Runs of Same Program

Data graphs persist between different runs of the same program. Graphs from previous runs will no be overwritten unless they are manually updated or automatic updating is turned on.

1. **Load the C or Fortran example program.**
2. **Set a breakpoint at the end of the program.**
3. **Start the program, running the program to the breakpoint.**
4. **Type `vec` into the expressions text field, and choose Data ➤ Graph Expression.**
5. **A graph of the `vec` array appears (as a sine curve).**
6. **Now you can edit the program (for example, replace sin with cos). Use fix and continue to recompile the program and continue (click the Fix tool bar button).**
7. **Restart the program.**
8. **Because automatic updating is turned off, the previous graph does not get updated when the program reaches the breakpoint.**
9. **Choose Data ➤ Graph Expression (`vec` is still in the Expressions text field), a graph of the current `vec` values appear alongside the graph of the previous run.**
10. **The data from the two runs can now be compared. The graph of the previous run will only change if it is updated manually using the update button, or if automatic updating is turned on.**

5

Fortran Example Program

```
real vec(100)
real af(50,50)
real bf(-3:3,-10:20)
real cf(50, 100, 200)

do x = 1,100
        ct = ct + 0.1
        vec(x) = sin(ct)
enddo

do x = 1,50
       do y = 1,50
         af(x,y) = (sin(x)+sin(y))*(20-abs(x+y))
       enddo
 enddo

do x = -3,3
       do y = -10,20
        bf(x,y) = y*(y-1)*(y-1.1)-10*x*x*(x*x-1)
       enddo
enddo

do x = 1,50
       do y = 1,100
        do z = 1,200
         cf(x,y,z) = 3*x*y*z - x*x*x - y*y*y - z*z*z
        enddo
       enddo
enddo

end
```

C Example Program

```
#include <math.h>
main()
{
    int x,y,z;
    float ct=0;
    float vec[100];
    float af[50][50];
    float bf[10][20];
    float cf[50][100][200];

    for (x=0; x<100; x++)
    {
        ct = ct + 0.1;
        vec[x] = sin(ct);
    }
    for (x=0; x<50; x++)
    {
        for (y=0; y<50; y++)
        {
        af[x][y] = (sin(x)+sin(y))*(20-abs(x+y));
        }
    }
    for (x=0; x<10; x++)
    {
        for (y=0; y<20; y++)
        {
          bf[x][y] = y*(y-1)*(y-1.1)-10*x*x*(x*x-1);
        }
    }
    for (x=0; x<50; x++)
    {
        for (y=0; y<100; y++)
        {
        for (z=0; z<200; z++)
        {
                cf[x][y][z] = 3*x*y*z - x*x*x - y*y*y - z*z*z ;
        }
        }
    }
}
```

Part 4 — Advanced Debugging

Runtime Checking 6

Runtime Checking (RTC) allows you to automatically detect runtime errors in an application during the development phase. RTC lets you detect runtime errors such as memory access errors and memory leak errors, and monitor memory usage.

The following topics are covered in this chapter:

Basic Concepts	*page 85*
Using RTC	*page 87*
Using Access Checking	*page 87*
Using Memory Use Checking	*page 90*
Setting Options	*page 95*

Basic Concepts

Because RTC is an integral debugging feature, all debugging functions such as setting breakpoints, examining variables and so on, can be used with RTC, except the Collector.

The following list briefly describes the features of RTC:

- Detects memory access errors
- Detects memory leaks
- Collects data on memory use
- Works with all languages

- Works on code that you do not have the source for, such as system libraries
- Works with multithreaded code.
- Requires no recompiling, relinking, or Makefile changes.

Compiling with the `-g` flag provides source line number correlation in the RTC error messages. RTC can also check programs compiled with the optimization `-O` flag. There are some special considerations with programs not compiled with the -g option. See *Sun WorkShop: Command Line Utilities, "Default Suppressions"* on page 127, for more information.

When to Use RTC

One way to avoid seeing a large number of errors at once is to use RTC earlier in the development cycle, as you are developing the individual modules that make up the program. Write a unit test to drive each module and use RTC incrementally to check each module one at a time. That way, you deal with a smaller number of errors at a time. When you integrate all of the modules into the full program, you are likely to encounter few new errors. When you reduce the number of errors to zero, you need to run RTC again only when you make changes to a module.

Requirements

- Programs compiled using a Sun Compiler
- Dynamic linking with `libc`
- Use of the standard `libc` `malloc/free/realloc` functions or allocators based on those functions

  dbx does provide an API to handle other allocators; see *Sun WorkShop: Command Line Utilities*, "*Using Fix & Continue with RTC"* on page 125.
- Programs that are not fully stripped. Programs stripped with `strip -x` are acceptable

Limitations

- Does not handle program text areas and data areas larger than 8Mb.

  A possible solution is to insert special files in the executable image to handle program text areas and data areas larger than 8Mb.

For more detailed information on any aspect of RTC, see the online help.

Using RTC

To use Runtime Checking, you enable the type of checking you want to use.

To turn on the desired checking mode from the Debugging window:

♦ **Choose Checks ➤ Enable Memuse Checking.**

-or-

♦ **Choose Checks ➤ Enable Access Checking.**
In the Debugging window status area, you see an indicator that RTC is enabled.

- For memory use checking, a blue recycling symbol with three arrows pointing in a circle.
- For access checking, a red circle with a white minus sign in the middle (the international Do Not Enter sign.)

♦ **Choose Execute ➤ Go or click Go to start the program.**

When RTC detects an error, it reports the type and location of the error and returns control to the user. You can perform any of the usual debugging activities such as setting breakpoints and examining variables.

Leak errors are reported after the program finishes execution. The Memory Use window opens with the leak and block information listed.

You can selectively suppress reporting of RTC errors using the dbx command `suppress`. For more information, see the `dbx` online help for `suppress`.

Using Access Checking

RTC checks whether your program accesses memory correctly by monitoring each read, write, and memory free operation.

Programs may incorrectly read or write memory in a variety of ways; these are called memory access errors. For example, the program may reference a block of memory which has been de-allocated through a `free()` call for a heap block, or because a function returned a pointer to a local variable. Access errors

may result in wild pointers in the program and can cause incorrect program behavior, including wrong outputs and segmentation violations. Some kinds of memory access errors can be very hard to track down.

RTC maintains a table that tracks the state of each block of memory being used by the program. When the program performs a memory operation, RTC checks the operation against the state of the block of memory it involves, to determine whether the operation is valid. The possible memory states are:

- Unallocated—initial state. Memory has not been allocated. It is illegal to read, write, or free this memory because it is not owned by the program.
- Allocated, but uninitialized. Memory has been allocated to the program but not initialized. It is legal to write to or free this memory, but is illegal to read it because it is uninitialized. For example, upon entering a function, stack memory for local variables is allocated, but uninitialized.
- Read-only. It is legal to read, but not write or free, read-only memory.
- Allocated and initialized. It is legal to read, write, or free allocated and initialized memory.

The program runs normally, except that it runs slower because each memory access is checked for validity just before it actually occurs. If an invalid access is detected, the Access Checking window opens and an error message giving specific information about the error is listed. The program is then suspended and control is returned to you. If the error is not a fatal error, you can continue execution of the program. The program continues to the next error or breakpoint, whichever is detected first.

Using RTC to find memory access errors is not unlike using a compiler to find syntax errors in your program. In both cases a list of errors is produced, with each error message giving the cause of the error and the program location where the error occurred. In both cases you should fix the errors in the program starting at the top of the error list and working your way down. The reason is that one error can cause the other errors in a sort of chain reaction. The first error in the chain is therefore the “first cause,” and fixing that error may also fix some subsequent errors. For example, a read from an uninitialized section of memory can create an incorrect pointer, which when dereferenced can cause another invalid read or write, which can in turn lead to other errors.

Memory Access Errors

RTC detects the following memory access errors:

- Read from uninitialized memory (rui)
- Read from unallocated memory (rua)
- Write to unallocated memory (wua)
- Write to read-only memory (wro)
- Misaligned read (mar)
- Misaligned write (maw)
- Duplicate free (duf)
- Bad free (baf)
- Misaligned free (maf)
- Out of memory (oom)

For a more detailed description of each memory access error checked by RTC, see the manual *Command-Line Utilities*.

Understanding the Memory Access Error Report

RTC prints the following information for memory access errors:

`type`	Type of error.
`access`	Type of access attempted (read or write).
`size`	Address of attempted access.
`addr`	Size of attempted access.
`detail`	More detailed information about addr. For example, if addr is in the vicinity of the stack, then its position relative to the current stack pointer is given. If addr is in the heap, then the address, size, and relative position of the nearest heap block is given.
`stack`	Call stack at time of error (with batch mode).
`allocation`	If *addr* is in the heap, then the allocation trace of the nearest heap block is given.

`location`	`Where the error occurred. If line number information is available, this information includes` *`line number`* `and` *`function`*`. If line numbers are not available, RTC provides` *`function`* `and` *`address`*`.`

The following example shows a typical access error:

```
Read from uninitialized (rui):
Attempting to read 4 bytes at address 0xeffff67c
    which is 1268 bytes above the current stack pointer
Location of error: Basic.c, line 56,
     read_uninited_memory()
```

Using Memory Use Checking

A memory leak is a dynamically allocated block of memory that has no pointers pointing to it anywhere in the data space of the program. Such blocks are orphaned memory. Because there are no pointers to the blocks, the program cannot even reference them, much less free them. RTC finds and reports such blocks.

Memory leaks result in increased virtual memory consumption and generally result in memory fragmentation. This may slow down the performance of your program and the whole system.

Typically, memory leaks occur because allocated memory is not freed and you lose a pointer to the allocated block. Here are some examples of memory leaks:

No free of s. Once foo returns, there is no pointer pointing to the malloc'ed block, so that block is leaked.

```
void
foo()
{
    char *s;
    s = (char *) malloc(32);

    strcpy(s, "hello world");

    return; /* }
}
```

A leak can result from incorrect use of an API:

libc function getcwd() returns a pointer to the malloc'ed area when the first argument is NULL. The program should remember to free this. In this case, the block is not freed and results in a leak.	`void`
	`printcwd()`
	`{`
	
	`    printf("cwd = %s\n", getcwd(NULL, MAXPATHLEN));`
	
	`    return;`
	`}`

Memory leaks can be avoided by following a good programming practice of always freeing memory when it is no longer needed and paying close attention to library functions which return allocated memory. If you use such functions, remember to free up the memory appropriately.

Sometimes, the term "memory leak" is used to refer to *any* block that has not been freed. This is a much less useful definition of a memory leak, because it is a common programming practice not to free memory if the program will terminate shortly anyway. RTC does not report a block as a leak if the program still retains one or more pointers to it.

Possible Leaks

There are two cases where RTC may report a "possible" leak. The first case is when no pointers were found pointing to the beginning of the block, but a pointer found pointing to the *interior* of the block. This case is reported as an Address in Block (aib) error. If it was a stray pointer that happened to point into the block, this would be a real memory leak. However, some programs deliberately move the only pointer to an array back and forth as needed to access its entries. In this case it would not be a memory leak. Because RTC cannot distinguish these two cases, it reports them as possible leaks, allowing the user to make the determination.

The second type of possible leak occurs when no pointers to a block were found in the data space, but a pointer was found in a register. This case is reported as an Address in Register (air) error. If the register happens to point to the block accidentally, or if it is an old copy of a memory pointer that has since been lost, then this is a real leak. However, the compiler can optimize

references and place the only pointer to a block in a register without ever writing the pointer to memory. In such cases, this would not be a real leak. In all other cases, it is likely to be a real leak.

Note – RTC leak checking requires use of the standard `libc` `malloc/free/realloc` functions or allocators based on those functions

Checking for Leaks

If memory leaks checking is turned on, a scan for memory leaks is automatically performed just before the program being tested exits. Any detected leaks are reported. The program should not be killed with the `kill` command. Here is a typical memory leak error message:

```
Memory leak (mel):
Found leaked block of size 6 at address 0x21718
At time of allocation, the call stack was:
    [1] foo() at line 63 in test.c
    [2] main() at line 47 in test.c
```

Clicking on the call stack location hypertext link takes you to that line of the source code in the editor window.

UNIX programs have a `main` procedure (called `MAIN` in f77) which is the top-level user function for the program. Normally, a program terminates either by calling `exit(3)` or by simply returning from `main`. In the latter case, all variables local to `main` go out of scope after the return, and any heap blocks they pointed to are reported as leaks (unless globals point to those same blocks).

It is a common programming practice not to free heap blocks allocated to local variables in `main`, because the program is about to terminate anyway, and then return from `main` without calling (`exit()`). To prevent RTC from reporting such blocks as memory leaks, stop the program just before `main` returns by setting a breakpoint on the last executable source line in `main.` When the program halts there, use the RTC `showleaks` command to report all the true leaks, omitting the leaks that would result merely from `main`'s variables going out of scope.

Detecting Memory Leak Errors

Note – RTC only finds leaks of `malloc` memory. If your program does not use `malloc`, RTC cannot find memory leaks.

RTC detects the following memory leak errors:

- Memory Leak (`mel`)
- Possible leak — Address in Register (`air`)
- Possible leak — Address in Block (`aib`)

For a more detailed description of each memory leak error RTC reports, see the *Command-Line Utilities* manual.

Memory Use Error Reporting

You have two choices for reporting memory blocks, a summary report and a detailed report.

To switch report types:

1. **From the Debugging window, choose Windows ➤ Memory Use Checking.**
2. **From the Leaks menu or the Blocks menu, choose Summary Report or Detailed Report to toggle on the report option you want to use.**

You can also set your default reporting option using the Debugging Options dialog box. See “Setting Options” on page 95 for more information.

Memory Use Error Types

Both reports include the following information for memory leak errors:

location	location where leaked block was allocated
addr	address of leaked block
size	size of leaked block
stack	call stack at time of allocation, as constrained by `check -frames`.

Because the number of individual leaks can be very large, RTC automatically combines leaks that were allocated at the same place into a single combined leak report.

However, the summary report capsulizes the error information into a table, while the detailed report gives you a separate error message for each error. They both contain a hypertext link to the location of the error in the source code.

Detailed Leak Report

A typical detailed report contains the following:

```
Actual leaks report    (actual leaks:   1 total size: 16 bytes)
Memory leak (mel):
Found leaked block of size 16 bytes at address 0x21590
At time of allocation, the call stack was:
            main

Possible leaks report  (possible leaks:  0 total size:   0 bytes)

Blocks in use report   (blocks in use:   0 total size:   0 bytes)
```

Summary Leak Report

When you use a summary report, the error information is summarized, and displayed as follows:

```
Actual leaks report   (actual leaks:  1   total size:   16 bytes)
Total Num of Leaked Allocation call stack
Size  Blocks Block   Address
===== ===== ======= ======================
 16     1    0x21590  main
```

The location on the call stack is a blue hypertext link that takes you to the appropriate place on the stack.

Setting Options

You can set options to control the reporting operation of RTC.

To set Debugging Runtime Checking options:

♦ **Choose Debug ➤ Debugging Options.**

♦ **Choose Category ➤ Runtime Checking.**

When you set an option here, you can choose to apply it to the current debugging session by clicking OK, or you can save it as the new default by clicking Save As Default and then clicking OK.

Access Checking Reporting

Automatic blocks report at exit

Sets whether your blocks report for Access Checking is automatically generated in detailed or summary form at program exit, or if no report is generated.

Automatic leaks report at exit

Sets whether your leaks report for Access Checking is automatically generated in detailed or summary form at program exit, or if no report is generated.

Error Reporting

Max. errors to report

Sets the maximum number of errors that RTC reports. The default is 1000.

The error limit is used separately for access errors and leak errors. For example, if the error limit is set to 5, then a maximum of 5 access errors and 5 memory leaks are shown in the report at the end of the run.

Log errors to file and continue

Causes RTC not to stop upon finding an error, but to continue running. The program stops when breakpoints are encountered or if the program is interrupted.

All errors are redirected to the default file `/tmp/dbx.errlog.<pid>`. You can select a different file to save errors to. To redirect all errors to the terminal, set the filename to `/dev/tty`.

Note – If the filename *filename,* already exists, the contents of that file are erased before output is redirected to that file.

Suppress duplicate error messages

Causes a particular error at a particular location to be reported only the first time it is encountered. This is useful for preventing multiple copies of the same error report when an error occurs in a loop which is executed many times.

Using Fix and Continue 7

Fixing allows you to quickly recompile edited source code without stopping the debugging process.

This chapter is organized into the following sections:

Basic Concepts	*page 97*
Fixing Your Program	*page 99*
Continuing after Fixing	*page 99*
Changing Variables after Fixing	*page 100*

Basic Concepts

The fix and continue feature allows you to modify and recompile a source file and continue executing without rebuilding the entire program. By updating the `.o` files and splicing them into your program, you don't need to relink.

The advantages of fixing and continuing are:

- You do not have to relink the program.
- You do not have to reload the program for debugging.
- You can resume running the program from the fix location.

Note – Do not use `fix` if a build is in process; the output from the two processes will intermingle in the Building window.

How Fix and Continue Operate

The modified files are compiled and shared object (`.so`) files are created. Semantic tests are done by comparing the old and new files. The new object file is linked to your running process using the runtime linker. If the function on top of the stack is being fixed, the new stopped in function is the beginning of the same line in the new function. All the breakpoints in the old file are moved to the new file. The modified source replaces the old source in the editor window, and you can resume debugging from the exact point where you stopped.

Modifying Source Using Fix and Continue

You can modify source code in the following ways when using `fix` and `continue`:

- Add, delete, or change lines of code in functions
- Add or delete functions
- Add or delete global and static variables

Restrictions

WorkShop might have problems when functions are mapped from the old file to the new file. To minimize such problems when editing a source file:

- Do not change the name of a function.
- Do not add, delete, or change the type of arguments to a function.
- Do not add, delete, or change the type of local variables in functions currently active on the stack.
- Do not make changes to the declaration of a template or to template instances. Only the body of a C++ template function definition can be modified.

If you need to make any of the proceeding changes, rebuild your program.

Fixing Your Program

To fix your file:

1. **Save the changes to your source.**

WorkShop automatically saves your changes if you forget this step.

2. **Choose Execute ➤ Fix Changes or click the Fix icon on the tool bar.**

The Build Output window opens and lets you know that your fix is underway. Any compile-time errors are listed and underscored, denoting links to source files.

Although you can do an unlimited number of fixes, if you have done several fixes in a row, consider rebuilding your program. Fixing changes the program image in memory, but not on the disk. As you do more fixes, the memory image gets out of sync with what is on the disk.

`fix` does not make the changes within your executable file, but only changes the `.o` files and the memory image. Once you have finished debugging a program, you need to rebuild your program to merge the changes into the executable.

For more information on customizing the Fix command, see *Using Fix and Continue* in the manual, *Command-Line Utilities*.

Continuing after Fixing

You can continue executing using Go, Step Into, or Start.

Before resuming program execution, you should be aware of the following conditions:

Changing an executed function

If you made changes in a function that has already executed, the changes will have no effect until:

- You run the program again.
- That function is called the next time.

If your modifications involve more than simple changes to variables, use Fix then Go. Using Go is faster than using Build because it does not relink the program.

Changing a function not yet called

If you made changes in a function not yet called, the changes will be in effect when that function is called.

Changing a function currently being executed

If you made changes to the function currently being executed, Fix's impact depends on where the change is relative to the stopped in function:

- If the change is in already executed code, the code is not re-executed. Execute the code by popping the current function off the stack and continuing from where the changed function is called. You need to know your code well enough to figure out whether the function has side effects that can't be undone (for example, opening a file).
- If the change is in code yet to be executed, the new code is run.

Changing a function presently on the stack

If you made changes to a function presently on the stack, but not the stopped in function, the changed code will not be used for the present call of that function. When the stopped in function returns, the old versions of the function on the stack execute.

There are two ways to solve this problem:

- Do a Fix and then Go.
- Pop the stack until all changed functions are removed from the stack. You need to know your code to be sure that there are no ill effects.

If there are breakpoints in modified functions on the stack, the breakpoints are moved to the new versions of the functions. If the old versions are executed, the program does not stop in those functions.

Changing Variables after Fixing

Changes made to global variables are not undone using Fix or by popping the stack. To manually reassign correct values to global variables, use the Data History pane of the Debugging window.

The following example shows how a simple bug can be fixed. The application gets a segmentation violation in line 6 when trying to dereference a NULL pointer:

```
    1   #include <stdio.h>
    2
    3   char *from = "ships";
    4   void copy(char *to)
    5   {
    6       while ((*to++ = *from++) != '\0');
    7       *to = '\0';
    8   }
    9
    10  main()
    11  {
    12      char buf[100];
    13
    14      copy(0);
    15      printf("%s\n", buf);
    16      return 0;
    17  }

signal SEGV (no mapping at the fault address) in copy at line 6
in
    file "testfix.cc"
    6       while ((*to++ = *from++) != '\0');
```

1. **Change line 14 to `copy` to `buf` instead of `0` and save the file**

2. **Choose Execute ➤ Fix Changes.**

modified line

```
    14      copy(buf);

fixing "testfix.cc".....
pc moved to "testfix.cc":6
stopped in copy at line 6 in file "testfix.cc"
    6       while ((*to++ = *from++) != '\0');
```

If the program is continued from here, it still gets a SEGV because the zero-pointer is still pushed on the stack.

3. **Pop one frame of the stack by choosing Execute ➤ Pop.**

```
stopped in main at line 14 in file "testfix.cc"
    14      copy(buf);
```

If the program is continued from here, it runs, but it will not contain the correct value because the global variable `from` has already been incremented by one. The program would print `hips` and not `ships`.

4. **Use the Assign button in the Debugging window to restore the global variable.**
5. **Click Go.**

Now the program prints the correct string:

```
ships
```

Index

Symbols

`.NO_PARALLEL:` special target, 12
`.PARALLEL:` special target, 12
`.WAIT` special target, 9

A

About box, 32
archiving libraries, 10
array
    graphing, 71
array display, 73
arrays
    automatic updating, 72
automatic parallelization
    -parallel compiler switch, 25

B

Build Server Configuration File, 4
build servers, 13

C

C array example program, 81
call-graph profile
    threads, 50
changing array display perspective, 73
changing the array display, 73
compiler switch
    -depend, 33
    -loopinfo, 33
    -O4, 25
    -parallel, 25
    promotion of, 32
    -Zlp, 25
    -Ztha (Thread Analyzer), 43
concurrent file modification, 10
condition variable
    time spent wating for signal, 42
contour array graph, 73

D

-depend (perform data dependency analysis), 33
dependency lists, 9
    explicit ordering, 9
    implicit ordering, 9
distributed make, explanation of, 3
DMake host, 13
Dmake, basic concept, 3
DOALL pragma
    mark loop for parallelization, 32

E

effects of optimizations applied to loops
- inlining, 38
- phantom loops, 38

environment variable
- LD_LIBRARY_PATH, 25
- PARALLEL, 24

exit Thread Analyzer, 46

explicit parallelization, 32

, 32

-explicitpar, 32

F

file
- collision, 12
- concurrent modification, 10

fix and continue, 97
- conditions, 99
- example, 101
- restrictions, 98
- using, 98

Fortran array example program, 80

function glyph (Thread Analyzer), 47

G

gnuemacs editor, help with, 30

graphing an array, 71

graphing an array from dbx, 72

H

hints about optimizations applied to loops, 34
- compiler generated two versions of this loop, 35
- loop contains backward flow of control, 37
- loop contains data dependency, 36
- loop contains I/O, or other function calls, that are not MT safe, 37
- loop contains multiple exits, 37
- loop contains procedure call, 35
- loop marked by user-inserted pragma, DOALL, 37
- loop may have been distributed, 37
- loop may or may not hold enough work to be profitably parallelized, 36
- loop significantly transformed during optimization, 36
- no hint available, 35
- two or more loops may have been fused, 38
- two or more loops may have been interchanged, 38

I

inlining, 38

instrumenting a program with-Ztha, 43

L

LD_LIBRARY_PATH environment variable, 25

library update, concurrent, 10

limitations on makefiles, 9

loop contains backward flow of control, 37

loop contains procedure call, 35

loop may have been distributed, 37

-loopinfo (print hints about loops), 33

LoopReport
- loading timing file, 26
- starting, 26

loops,phantom, 38

LoopTool
- bar chart of loop runtimes, 27
- choosing an editor, 30
- creating a detailed report on loops, 29
- editing source code, 30
- getting help, 32
- gettting hints, 30
- graphical user interface, 26
- LD_LIBRARY_PATH, 25
- loading timing file, 26

looptool command, 27
LVPATH environment variable, 27
online explanation of all compiler hints, 29
opening files, 28
printing the LoopTool graph, 29
sending comments, 32
setting default search paths, 30
specified via command line (looptool command), 27
starting, 26
Version menu in editor, 30
XUSERFILESEARCHPATH, 24
LVPATH environment variable, 27

M

macro
dynamic, 10
makefiles, limitations, 9
metrics collected by Thread Analyzer, 48
multiple targets, 11
multiprocessing, 19
multiprocessors, how many, 24
multithreaded, 19

N

naming convention, thread, 48
ncpus utility, 24
No hint available, 35

O

optimizations applied to loops
inlining, 38
loop transformations
jamming, 39
transposition, 39
unrolling, 39
phantom loops, 38

P

LoopReport
, 27
LoopTool
, 27
-p option, 27
-parallel (automatic parellelization), 32
PARALLEL environment variable, 24
parallel loop nested inside serial loop
wallclock anomaly, 39
parallelism, 20
restricting, 12
parallelize loop marked by DOALL pragma
-explicitpar, 32
phantom loops, 38
pragma
DOALL, 32
processors, how many on your machine, 24
profile data
functions, 51
program, 51
threads, 51
program glyph (Thread Analyzer), 47
promotion of compiler switches, 32

Q

quit Thread Analyzer, 46

R

read system call
number per second, 43
reader/writer read lock
time spent wating to acquire, 43
reader/writer write lock
time spent waiting to acquire, 43
restricting parallelism, 12
restrictions on makefiles, 9
rotating array display, 73
runtime checking
features, 85
limitations, 86

- memory access error checking, 87
- memory access error reporting, 89
- memory access errors, 89
- memory leak error reporting, 93
- memory leak errors, 93
- memory leaks checking, 90
- setting options, 95
- starting, 87

Runtime Configuration File, 4

S

surface array graph, 73

T

targets
- .NO_PARALLEL:, 12
- .PARALLEL:, 12
- .WAIT, 9
- multiple, 11

Thread Analyzer, 41 to 53, 53 to 65
- blocking on I/O or thread synchronization, 61
- bottleneck, narrowing focus, 61 to 63
- call-graph profile for threads, 50
- collapse glyph hierarchy, 48
- collecting metrics, 44
- conventions, 48
- CPU time filter, 63 to 65
- display graph, 49
- display table, 49
- error messages, 48
- exit, 46
- expand glyph hierarchy, 48
- filter, 53
- glyph hierarchy, 48
- gprof table, 50
- gprof table for particular thread, 56
- graph condition variable wait time, 63
- hierarchy navigation, 48
- horizontal scrollbar, 47
- identified via graphical data, 60 to ??
- initial investigation, 54 to 59
- instrumenting program, 43
- interactive error messages, 48
- Load button, 49
- load trace directory, 49
- loading a trace directory via command line, 44
- manipulating menus, 48
- metric
  - condition variable wait time, 42
  - CPU time, 42
  - file reads (ops), 43
  - file write (bytes), 43
  - file writes (ops), 43
  - reader/writer read lock wait, 43
  - reader/writer write lock, 43
  - total sync wait time, 43
  - wall clock time, 42
- metric graph, 49
- metric graph property sheet, 49
- metric scope, entire program, 42
- metric scope, single function, 42
- metric scope, single thread, 42
- metric table, 52
- metric table for particular function, 58
- metrics, 48
  - collected by, 48
- mouse gestures, 48
- multiple metrics for particular thread or function, 52
- multiple metrics, particular function, 52
- multiple metrics, particular thread, 52
- navigating through glyph hierarchy, 48
- particular glyph level, 51
- performance bottlenecks, 60 to 61
- plot CPU time versus wallclock time, 60
- prof table, 51
- prof table for all threads, 55
- profile data for functions, 51
- profile data for program, 51
- profile data for threads, 51
- property sheet, 49, 51
- quit, 46

select metric, 49
sorted metric profile table, 51
sorted metric profile table for all threads, 59
sorted metric profile table property sheet, 51
specifying pathname to executable via command line, 44
start ThA via comand line, 44
tha.pid directory, 44
thread naming convention, 48
threshold CPU time for function, 53
threshold CPU time for thread, 53
trace data file, 46
View menu, 49
write trace data files, 44
-Ztha, 43
thread glyph (Thread Analyzer), 47
thread naming convention, 48
thread synchronization
time spent on, 43
timing file, 26
timing file location
LoopTool
current directory, 26
specified via environment variable, 27
specified via -p option, 27
Two or more loops may have been fused, 38
two or more loops may have been interchanged, 38

U

updating array displays, 72

V

Version menu, in LoopTool, 30
vi editor, help with, 30
vitem, 72

W

wallclock anomaly
parallel loop nested inside serial loop, 39
write system call
number of bytes written, 43
number per second, 43

X

xemacs editor, help with, 30
XUSERFILESEARCHPATH (LoopTool environment variable), 24

Z

-Ztha
instrument program
C, 43
C++, 43
FORTRAN, 43
-Ztha compiler option for Thread Analyzer, 43

Copyright 1996 Sun Microsystems Inc., 2550 Garcia Avenue, Mountain View, Californie 94043-1100, U.S.A. Tous droits réservés.

Ce produit ou document est protégé par un copyright et distribué avec des licences qui en restreignent l'utilisation, la copie, la distribution, et la décompilation. Aucune partie de ce produit ou de sa documentation associée ne peut être reproduite sous aucune forme, par quelque moyen que ce soit, sans l'autorisation préalable et écrite de Sun et de ses bailleurs de licence, s'il y en a.

Des parties de ce produit pourront être dérivées du système UNIX® licencié par Novell, Inc. et du système Berkeley 4.3 BSD licencié par l'Université de Californie. UNIX est une marque enregistrée aux Etats-Unis et dans d'autres pays et licenciée exclusivement par X/Open Company Ltd. Le logiciel détenu par des tiers, et qui comprend la technologie relative aux polices de caractères, est protégé par un copyright et licencié par des fournisseurs de Sun.

Sun, Sun Microsystems, le logo Sun, SunOS, Sun WorkShop, LoopTool, LockLint, Thread Analyzer, Sun C, Sun C++, et Sun FORTRAN sont des marques déposées ou enregistrées de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC, utilisées sous licence, sont des marques déposées ou enregistrées de SPARC International, Inc. aux Etats-Unis et dans d'autres pays. Les produits portant les marques SPARC sont basés sur une architecture développée par Sun Microsystems, Inc.

Les interfaces d'utilisation graphique OPEN LOOK® et Sun™ ont été développées par Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox pour la recherche et le développement du concept des interfaces d'utilisation visuelle ou graphique pour l'industrie de l'informatique. Sun détient une licence non exclusive de Xerox sur l'interface d'utilisation graphique Xerox, cette licence couvrant aussi les licenciés de Sun qui mettent en place l'interface d'utilisation graphique OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

Le système X Window est un produit de X Consortium, Inc.

CETTE PUBLICATION EST FOURNIE "EN L'ETAT" SANS GARANTIE D'AUCUNE SORTE, NI EXPRESSE NI IMPLICITE, Y COMPRIS, ET SANS QUE CETTE LISTE NE SOIT LIMITATIVE, DES GARANTIES CONCERNANT LA VALEUR MARCHANDE, L'APTITUDE DES PRODUITS A RÉPONDRE A UNE UTILISATION PARTICULIERE, OU LE FAIT QU'ILS NE SOIENT PAS CONTREFAISANTS DE PRODUITS DE TIERS.