



---

## Analyzing Program Performance With Sun WorkShop

---

901 San Antonio Road  
Palo Alto, , CA 94303-4900  
USA 650 960-1300 fax 650 969-9131

Part No: 805-4947  
Revision A, February 1999

Copyright Copyright 1999 Sun Microsystems, Inc. 901 San Antonio Road, Palo Alto, California 94303-4900 U.S.A. All rights reserved.

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 in 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, SunDocs, SunExpress, Solaris, Sun Performance Library, Sun Performance WorkShop, Sun Performance WorkShop Fortran, Sun Visual WorkShop, Sun WorkShop, Sun WorkShop Compilers C, Sun WorkShop Compilers C++, Sun WorkShop Compilers Fortran, Sun WorkShop Memory Monitor, Sun WorkShop Professional, Sun WorkShop Professional C, and Sun WorkShop TeamWare are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the U.S. and other countries. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. in the U.S. 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 Corporation in researching and developing the concept of visual or graphical user interfaces for the computer industry. Sun holds a nonexclusive 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.

Copyright 1999 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, Californie 94303-4900 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 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, the Sun logo, SunDocs, SunExpress, Solaris, Sun Performance Library, Sun Performance WorkShop, Sun Performance WorkShop Fortran, Sun Visual WorkShop, Sun WorkShop, Sun WorkShop Compilers C, Sun WorkShop Compilers C++, Sun WorkShop Compilers Fortran, Sun WorkShop Memory Monitor, Sun WorkShop Professional, Sun WorkShop Professional C, et Sun WorkShop TeamWare sont des marques de fabrique ou des marques déposées, ou marques de service, de Sun Microsystems, Inc. aux Etats-Unis et dans d'autres pays. Toutes les marques SPARC sont utilisées sous licence et sont des marques de fabrique ou des marques déposées 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 utilisateurs d'interfaces graphiques OPEN LOOK® et Sun™ ont été développés de Sun Microsystems, Inc. pour ses utilisateurs et licenciés. Sun reconnaît les efforts de pionniers de Xerox Corporation 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, cette licence couvrant aussi les licenciés de Sun qui mettent en place les utilisateurs d'interfaces graphiques OPEN LOOK et qui en outre se conforment aux licences écrites de Sun.

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 REpondre A UNE UTILISATION PARTICULIERE OU LE FAIT QU'ILS NE SOIENT PAS CONTREFAISANTS DE PRODUITS DE TIERS.



# Contents

---

	<b>Preface</b>	<b>ix</b>
<b>1.</b>	<b>Performance Profiling and Analysis Tools</b>	<b>1</b>
	Sampling Analysis Tools	2
	The Sampling Collector	2
	The Sampling Analyzer	2
	Loop Analysis Tools	3
	LoopTool	3
	LoopReport	3
	Lock Analysis Tool	4
	Traditional Profiling Tools	4
<b>2.</b>	<b>The Sampling Analysis Tools</b>	<b>7</b>
	Understanding the Sampling Collector	8
	Using the Sampling Collector	9
	Summary Data	9
	Execution Profile Data	9
	Program Memory Usage: Address Space Data	9
	Collecting Sampling Data	10
	Understanding the Sampling Analyzer	14
	Loading an Experiment	15

Selecting Data Types for Viewing	15
Selecting Display Options	17
Reordering an Application	29
Comparing Runtime Experiment Samples	31
Printing Experiments	31
Exporting Experiment Data	32
<b>3. Loop Analysis Tools</b>	<b>33</b>
Basic Concepts	33
Setting Up Your Environment	34
Creating a Loop Timing File	34
Other Compilation Options	35
Run The Program	36
Starting LoopTool	37
Loading a Timing File	37
Using LoopTool	38
Opening Files	38
Creating a Report on All Loops	39
Printing the LoopTool Graph	39
Choosing an Editor	40
Getting Hints and Editing Source Code	40
Starting LoopReport	41
Timing File	42
Fields in the Loop Report	44
Compiler Hints	45
0. No Hint Available	46
1. Loop contains procedure call	46
2. Compiler generated two versions of this loop	47
3. The variable(s) “ <i>list</i> ” cause a data dependency in this loop	47

4. Loop was significantly transformed during optimization	48
5. Loop may or may not hold enough work to be profitably parallelized	48
6. Loop was marked by user-inserted pragma, <code>DOALL</code>	48
7. Loop contains multiple exits	48
8. Loop contains I/O, or other function calls, that are not MT safe	49
9. Loop contains backward flow of control	49
10. Loop may have been distributed	49
11. Two or more loops may have been fused	49
12. Two or more loops may have been interchanged	49
How Optimization Affects Loops	50
Inlining	50
Loop Transformations—Unrolling, Jamming, Splitting, and Transposing	50
Parallel Loops Nested Inside Serial Loops	51
<b>4. Traditional Profiling Tools</b>	<b>53</b>
Basic Concepts	53
Using <code>prof</code> to Generate a Program Profile	54
Sample <code>prof</code> Output	56
Using <code>gprof</code> to Generate a Call Graph Profile	56
Using <code>tcov</code> for Statement-Level Analysis	59
Compiling for <code>tcov</code>	60
Creating <code>tcov</code> Profiled Shared Libraries	62
Locking Files	62
Errors Reported by <code>tcov</code> Runtime	63
<code>tcov</code> Enhanced—Statement-level Analysis	64
Compiling for <code>tcov</code> Enhanced	64
Creating Profiled Shared Libraries	66
Locking Files	66

<b>5. Lock Analysis Tool</b>	<b>71</b>
Basic Concepts	71
LockLint Overview	72
Collecting Information for LockLint	74
LockLint User Interface	74
How to Use LockLint	75
Managing LockLint's Environment	76
Compiling Code	78
LockLint Subcommands	78
Checking an Application	79
Program Knowledge Management	81
Analysis of Lock Usage	83
Limitations of LockLint	84
Source Code Annotations	86
Reasons to Use Source Code Annotations	87
The Annotations Scheme	87
Using LockLint NOTES	88
Assertions Recognized by LockLint	97
<b>A. LockLint Command Reference</b>	<b>101</b>
Subcommand Summary	101
LockLint Naming Conventions	103
LockLint Subcommands	106
analyze	107
assert	109
assert <i>mutex rwlock</i> protects	110
assert order	111
assert read only	111

assert *rwlock* covers 111  
declare 112  
declare mutex *mutex* declare rwlocks *rwlock* 112  
declare *func\_ptr* targets *func* 113  
declare nonreturning *func* 113  
declare one *tag* 113  
declare readable *var* 114  
declare root *func* 114  
disallow 114  
disallows 114  
exit 115  
files 115  
funcptrs 115  
funcs 116  
help 121  
ignore 121  
load 122  
locks 122  
members 124  
order 125  
pointer calls 126  
reallow 126  
reallows 127  
refresh 127  
restore 127  
save 128  
saves 128  
start 129

Start Examples	129
sym	130
unassert	130
vars	131
Lock Inversions	133
<b>Index</b>	<b>135</b>

# Preface

---

This manual describes the performance and analysis tools available with Sun WorkShop™. Developing high performance applications requires a combination of compiler features, libraries of optimized routines, and tools to analyze and isolate code. Analyzing Program Performance With Sun WorkShop describes the third part of this development strategy, and shows you how to use these tools:

The Sampling Analyzer and Sampling CollectorLoopTool and LoopReportLockLintprof, gprof, and tcov

“Related Books” on page x lists sources for information about Sun WorkShop compilers and performance libraries.

---

## Who Should Use This Book

This manual is intended for programmers with a working knowledge of Sun WorkShop and some understanding of the Solaris™ operating environment and UNIX® commands. Some knowledge of performance analysis is also helpful in understanding how to use the data derived from the tools, but is not required for using them. The traditional profiling tools prof, gprof, and tcov, do not require a working knowledge of Sun WorkShop.

---

## How This Book Is Organized

Chapter 1, "Performance Profiling and Analysis Tools," introduces the performance analysis tools, briefly discussing what they do and when to use them.

Chapter 2, "The Sampling Analysis Tools," describes the Sampling Collector and the Sampling Analyzer. You can use these tools to help you improve your program's use of resources.

Chapter 3, "Loop Analysis Tools," presents LoopReport and LoopTool, which help you analyze program loops that have been parallelized by your compiler.

Chapter 4, "Traditional Profiling Tools," covers the traditional profiling tools `prof`, `gprof`, and `tcov`, which help you find the parts of your program that are most heavily used, and determine how much of your program is being tested.

Chapter 5, "Lock Analysis Tool," documents LockLint, which assists you in analyzing locks and potential race conditions in multi-threaded programs.

Appendix A, "LockLint Command Reference," provides reference-style information on the LockLint commands.

---

## Multiplatform Release

---

**Note** - The name of the latest Solaris operating environment release is Solaris 7 but code and path or package path names may use Solaris 2.7 or SunOS 5.7.

---

The Sun<sup>™</sup> WorkShop<sup>™</sup> documentation applies to Solaris 2.5.1, Solaris 2.6, and Solaris 7 operating environments on:

- The SPARC<sup>™</sup> platform
- The x86 platform, where x86 refers to the Intel implementation of one of the following: Intel 80386, Intel 80486, Pentium, or the equivalent

---

**Note** - The term "x86" refers to the Intel 8086 family of microprocessor chips, including the Pentium, Pentium Pro, and Pentium II processors and compatible microprocessor chips made by AMD and Cyrix. In this document, the term "x86" refers to the overall platform architecture. Features described in this book that are particular to a specific platform are differentiated by the terms "SPARC" and "x86" in the text.

---

---

## Related Books

This document describes performance analysis tools you can use with programs written in a variety of languages. The following lists name manuals containing related information on compilers, usage data, and other aspects of Sun WorkShop.

## Other Programming Books

- *C User's Guide* describes compiler options, pragmas, and more.
- *Sun Visual WorkShop C++ Overview* gives a high-level outline of the C++ package suite.
- *C++ User's Guide* provides information on command-line options and how to use the compiler.
- *C++ Programming Guide* discusses issues relating to the use of templates, exception handling, and interfacing with FORTRAN 77.
- *C++ Migration Guide* describes migrations between compiler releases.
- *C++ Library Reference* explains the `iostream` libraries.
- *Tools.h++ User's Guide* provides details on the `Tools.h++` class library.
- *Tools.h++ Class Library Reference* discusses use of the C++ classes for enhancing the efficiency of your programs.
- *Sun Performance WorkShop Fortran Overview* gives a high-level outline of the Fortran package suite.
- *Fortran User's Guide* provides information on command-line options and how to use the compilers.
- *Fortran Programming Guide* discusses issues relating to input/output, libraries, program analysis, debugging, and performance.
- *Fortran Library Reference* gives detail on the language and routines.
- *FORTRAN 77 Language Reference Manual* provides a complete language reference.
- *Numerical Computation Guide* details floating-point computation numerical accuracy issues.
- *Standard C++ Library User's Guide* describes how to use the Standard C++ Library.
- *Standard C++ Class Library Reference* provides detail on the Standard C++ Library.

## Other Sun WorkShop Books

- *Sun WorkShop Quick Install* provides installation instructions.
- *Sun WorkShop Installation and Licensing Reference* provides supporting installation and licensing information.
- *Using Sun WorkShop* gives information on performing development operations through Sun WorkShop.
- *Debugging a Program With dbx* provides information on using `dbx` commands to debug a program.
- *Sun WorkShop TeamWare User's Guide* describes how to use the Sun WorkShop TeamWare code management tools.

- *Sun WorkShop Performance Library Reference Manual* discusses the library of subroutines and functions to perform useful operations in computational linear algebra and Fourier transforms.
- *Sun WorkShop Visual User's Guide* describes how to use Visual to create C++ and Java™ graphical user interfaces.
- *Sun WorkShop Memory Monitor User's Manual* describes how to use the Sun WorkShop Memory Monitor garbage collection and memory management tools.

## Solaris Books

The following Solaris manuals and guides provide additional useful information:

- The *Solaris Linker and Libraries Guide* gives information on linking and libraries.
- The *Solaris Programming Utilities Guide* provides information for developers about the special built-in programming tools available in the SunOS™ system.

---

## Ordering Sun Documents

The SunDocsSM program provides more than 250 manuals from Sun Microsystems, Inc. If you live in the United States, Canada, Europe, or Japan, you can purchase documentation sets or individual manuals using this program.

For a list of documents and how to order them, see the catalog section of the SunExpress™ Internet site at <http://www.sun.com/sunexpress>.

---

## Accessing Sun Documents Online

Sun WorkShop documentation is available online from several sources:

- The `docs.sun.com` Web site
- AnswerBook2™ collections
- HTML documents
- Online help and release notes

## Using the docs.sun.com Web site

The docs.sun.com Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

## Accessing AnswerBook2 Collections

The Sun WorkShop documentation is also available using AnswerBook2 software. To access the AnswerBook2 collections, your system administrator must have installed the AnswerBook2 documents during the installation process (if the documents are not installed, see your system administrator or Chapter 3 of *Sun WorkShop Quick Install* for installation instructions). For information about accessing AnswerBook2 documents, see Chapter 6 of *Sun WorkShop Quick Install*, Solaris installation documentation, or your system administrator.

---

**Note** - To access AnswerBook2 documents, Solaris 2.5.1 users must first download AnswerBook2 documentation server software from a Sun Web page. For more information, see Chapter 6 of *Sun WorkShop Quick Install*.

---

## Accessing HTML Documents

The following Sun Workshop documents are available online only in HTML format:

- Tools.h++ Class Library Reference
- Tools.h++ User's Guide
- *Numerical Computation Guide*
- Standard C++ Library User's Guide
- *Standard C++ Class Library Reference*
- *Sun WorkShop Performance Library Reference Manual*
- *Sun WorkShop Visual User's Guide*
- Sun WorkShop Memory Monitor User's Manual

To access these HTML documents:

**1. Open the following file through your HTML browser:**

`install-directory/SUNWspr0/DOC5.0/lib/locale/C/html/index.html`

Replace *install-directory* with the name of the directory where your Sun WorkShop software is installed (the default is `/opt`).

The browser displays an index of the HTML documents for the Sun WorkShop products that are installed.

2. Open a document in the index by clicking the document's title.

## Accessing Sun WorkShop Online Help and Release Notes

This release of Sun WorkShop includes an online help system as well as online manuals. To find out more see:

- Online Help. A help system containing extensive task-oriented, context-sensitive help. To access the help, choose Help > Help Contents. Help menus are available in all Sun WorkShop windows.
- Release Notes. The Release Notes contain general information about Sun WorkShop and specific information about software limitations and bugs. To access the Release Notes, choose Help > Release Notes.

---

## 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
<code>AaBbCc123</code>	The names of commands, files, and directories; on-screen computer output	Edit your <code>.login</code> file. Use <code>ls -a</code> to list all files. <code>machine_name% You have mail.</code>
<b>AaBbCc123</b>	What you type, contrasted with on-screen computer output	<code>machine_name% su</code> <code>Password:</code>
<i>AaBbCc123</i>	Command-line placeholder: replace with a real name or value	To delete a file, type <code>rm filename</code> .
<i>AaBbCc123</i>	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in <i>User's Guide</i> . These are called <i>class</i> options. You <i>must</i> 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 System 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	#



# Performance Profiling and Analysis Tools

---

Developing high performance applications requires a combination of compiler features, libraries of optimized routines, and tools to analyze and isolate code. *Analyzing Program Performance With Sun WorkShop* describes these tools that are available to help you achieve the third part of this development strategy:

- The Sampling Collector and Sampling Analyzer

The Sampling Collector collects performance behavior data about a program during runtime and writes that data to a file. The Sampling Analyzer examines the data and displays it graphically.

The Sampling Collector and Sampling Analyzer are included in the following Sun Workshop products:

- Sun Professional WorkShop C
- Sun Visual WorkShop C++
- Sun Performance WorkShop Fortran
- Sun WorkShop University Edition
- LoopTool and LoopReport

LoopTool supports performance tuning of automatically parallelized programs. LoopReport is the command line version of LoopTool.

LoopTool and LoopReport are included in the following Sun Workshop products:

- Sun Visual WorkShop C++
- Sun Performance WorkShop Fortran
- Sun WorkShop University Edition
- LockLint

LockLint extends Sun WorkShop development tools and compilers with support for development of multiprocessing/multithreaded applications.

LockLint performs a static analysis of the use of mutex and multiple readers-single writer locks, and looks for inconsistent use of these locking techniques. LockLint is included in the following Sun Workshop products:

- Sun Visual WorkShop C++
- Sun Performance WorkShop Fortran
- Sun WorkShop University Edition
- `prof`, `gprof`, and `tcov`

`prof` and `gprof` are traditional tools for generating profile data and are included with versions 2.5.1, 2.6, and Solaris 7 of the Solaris *SPARC Platform Edition* and Solaris *Intel Platform Edition*.

`tcov` is a code coverage tool. It is included in Sun Workshop.

---

## Sampling Analysis Tools

Sun WorkShop provides a pair of tools that you use together to collect performance data on your application and analyze that data.

### The Sampling Collector

The Sampling Collector is a graphical tool that is available from the Sun WorkShop debugger. When you execute your application, the Sampling Collector collects performance data from the kernel; this is called an *experiment*. The Sampling Collector writes the data it collects to a file that is called an *experiment record*. The experiment record includes resource usage data, program memory usage data, and execution profile data with or without called-function times.

### The Sampling Analyzer

The Sampling Analyzer examines the experiment record written by the Sampling Collector and displays it graphically. Two additional utilities enable you to convert the experiment record into ASCII format (`er_export`) and to print the data to a file or printer (`er_print`).

The sampling analysis tools are described in Chapter 2.

---

# Loop Analysis Tools

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 or LoopReport to examine loops parallelized by the FORTRAN 77, Fortran 90, or C compiler.

## LoopTool

LoopTool is a graphical analysis tool that reads the loop timing files created by programs compiled for loop analysis with Fortran and C compilers. LoopTool enables you to:

- Time all loops, whether serial or parallel
- Produce a table of loop timings
- Collect hints during compilation that can help you parallelize loops

LoopTool displays a graph of loop run times and identifies which loops were parallelized. You can go directly from the display to the source code for any loop.

To use LoopTool, you compile your program with the `-zlp` option. When you run a program that has been compiled with the `-zlp` option, Sun WorkShop creates a loop timing file. You run LoopTool to read the timing file and display its results. Sun WorkShop provides a demonstration program named `vgam` and a demonstration timing file named `vgam.looptimes`. When you start LoopTool from the Sun WorkShop Tools menu, it displays results from these files. You can use these demos to become familiar with LoopTool or specify your own files if you have already compiled a program with the `-zlp` option. LoopTool is described in Chapter 3.

## LoopReport

LoopReport is the command line version of LoopTool. LoopReport produces an ASCII file of loop times instead of a graphical display. LoopReport is described in Chapter 3.

---

## Lock Analysis Tool

Many C developers thread their applications by programming directly against Solaris user-level threads via the `libthread` library. With Sun WorkShop applications, you can run the multithreaded 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.

In multithreaded applications, threads must acquire and release locks associated with the data they share. When threads fail to acquire and release locks appropriately, they can cause your program to produce different results over different executions with the same input. This condition is known as a *data race*. Data races are easy problems to introduce and difficult ones to find.

A program can also *deadlock* when various threads are waiting for a lock that will not be released. For a definition of deadlock, see “Basic Concepts” on page 71.

Sun WorkShop provides a command-line utility for analyzing locks and how they are used. LockLint looks for inconsistent use of mutex and multiple readers-single writer locks. It detects a common cause of data races: failure to hold the appropriate lock while accessing a variable. LockLint also detects common causes for deadlocks.

To gather the information used by LockLint you specify the `-Zll` option to the C compiler, which then generates an `.ll` file for each `.c` source code file. The `.ll` file contains information about the flow of control in each function and about each access to a variable or operation on a mutex or readers-writer lock.

LockLint is described in Chapter 5.

---

## Traditional Profiling Tools

Sun WorkShop includes three command-line utilities for reporting data about your program:

- `prof` and `gprof` for performance profiling
- `tcov` for code coverage

Use `prof` and `gprof` to gather and display information for threaded (including auto-parallelized) programs. These tools provide a variety of levels of feedback about your program for you to analyze. They work in combination with other utilities and compiler options to collect and use performance data. `prof` generates a program profile in a flat file. `gprof` generates a call graph profile. `tcov` generates statement-level information in a copy of the source file, annotated to show which lines are used and how often.

These tools are described in Chapter 4.



## The Sampling Analysis Tools

---

Sun WorkShop provides a pair of tools that you use together to collect performance data on your application and analyze that data: the Sampling Collector and the Sampling Analyzer. The Sampling Collector collects performance data and the Sampling Analyzer displays the data graphically. These tools are designed for use by any software developer, even if performance tuning is not the developer's main responsibility.

This chapter is organized as follows:

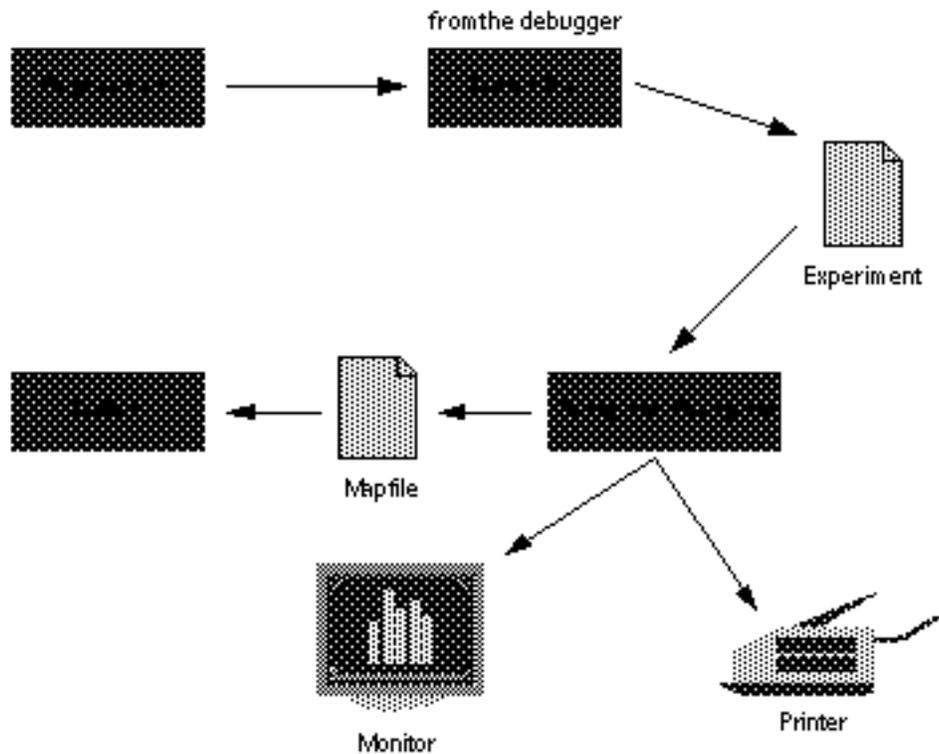
- “Understanding the Sampling Collector” on page 8
- “Using the Sampling Collector” on page 9
- “Understanding the Sampling Analyzer” on page 14

The Sampling Collector collects timing summary information for the operating system, address space data, and statistical samples of either program counters or call stacks, and shows inclusive and exclusive times in various states on a per-function basis.

The Sampling Analyzer lets you browse an experiment recorded with the Sampling Collector. In certain cases, the Sampling Analyzer helps rebuild a tuned application by creating a mapfile. You can use the information in the mapfile to improve the order of loading functions into the application address space, thus reducing the memory footprint of your application. When you link the application with the new loading order, you get an executable with a smaller text working set size.

The Analyzer identifies a way to improve the order of loading functions into the application address space. You can link the application again to create an executable that runs with a smaller text working set size.

Figure 2-1 illustrates the basic performance-tuning architecture.



*Figure 2-1* Performance Tuning Using the Sampling Collector and Sampling Analyzer

---

## Understanding the Sampling Collector

The Sampling Collector, a GUI tool within the Sun WorkShop debugger, collects performance behavior data from the kernel under which the application is running and writes that data to an experiment record file. The execution of the application during which data is collected is called the experiment.

Before collecting behavior data, you can define the following:

- The directory and file name to which you want experiment data written
- The period of time during which to sample data
- The type of data to collect

---

# Using the Sampling Collector

The Sampling Collector can gather:

- Summary data for the application
- Execution profile data with or without called function times
- Program memory usage data

The Sampling Collector records data samples, each sample representing a portion of a run. To display the data for any set of such samples, you use the Sampling Analyzer.

## Summary Data

Summary statistics are always collected, and include two kinds of data. The first kind is shown on the initial screen of the Sampling Analyzer, and is a summary of the amount of time the program spent in various states. States include user time, system time, system wait time, text page fault time, and data page fault time. The second kind of statistics is shown in the “Execution Statistics” screen of the Analyzer, and includes page fault and I/O statistics, context switches, and a variety of page residency (working-set and paging) statistics.

## Execution Profile Data

Execution profile data shows how much time is consumed by functions, modules, and segments while the application is running. It can be displayed in the Sampling Analyzer as a histogram or as a cumulative histogram (if called-function times are included).

Execution profile data helps answer the following kinds of questions:

- How much time does the application spend executing various functions, object modules, or segments?
- What functions, modules, or segments are consuming the greatest amount of time?
- If called-function times are included in the sample, how much time is spent by functions that call other functions?

## Program Memory Usage: Address Space Data

Address space data represents the process address space as a series of segments, each of which contains a number of pages. It allows the Sampling Analyzer to describe

the status of each page and whether it was referenced or modified. Program memory-usage data can be displayed in the Sampling Analyzer in the Address Space display.

Address space data helps answer the following kinds of questions:

- How does memory usage change over time?
- Is the application using pages efficiently?
- Which pages are accessed during a particular operation?

## Collecting Sampling Data

Before you can collect data:

- Your program must be loaded in the Sun WorkShop Debugging window.
- Runtime checking must be turned off.

To collect data:

- 1. In the Debugging window, choose Windows > Sampling Collector to open the Sampling Collector Window (see Figure 2-1)..**

- 2. Select whether you want to collect data for only one run or for all runs.**

If you run the Sampling Collector for only one run, the Sampling Collector shuts off after the first experiment is created. If you leave the Sampling Collector on for all runs, the Sampling Collector remains on, and subsequent executions record additional experiments.

Turning the Sampling Collector on does not start data collection. Performance data is collected only when you execute the program in the Debugging window.

- 3. Enter the complete path name for your experiment file in the Experiment File text box.**

The Sampling Collector provides the default experiment-record name `test.1.er`. If you use the `.1.er` suffix for your experiment-record filename, the Sampling Collector automatically increments the names of subsequent experiments by one—for example, `test.1.er` is followed by `test.2.er`.

- 4. Select the data to be collected.**

Summary data is always collected, but most users want to collect information about functions that are executed as well; to do so, select Execution Profile data. Execution profile data can include or exclude time spent in called functions. Most users want to include such time; to do so, select Include called function time.

If you are concerned about your application's memory usage, you should select Address Space data.

Data is collected in terms of samples, each representing a sample of the program's execution. By default, samples are collected periodically. If you select Manually, on 'New Sample' command, samples are marked in response to the new sample command or button. Whether you collect samples manually or periodically, additional samples are always marked when the program encounters a breakpoint.

**5. Start the program running in the Debugging window by clicking either Start or Go.**

Start begins sampling the program from the beginning of the code. Go begins sampling it from the current location in the code.

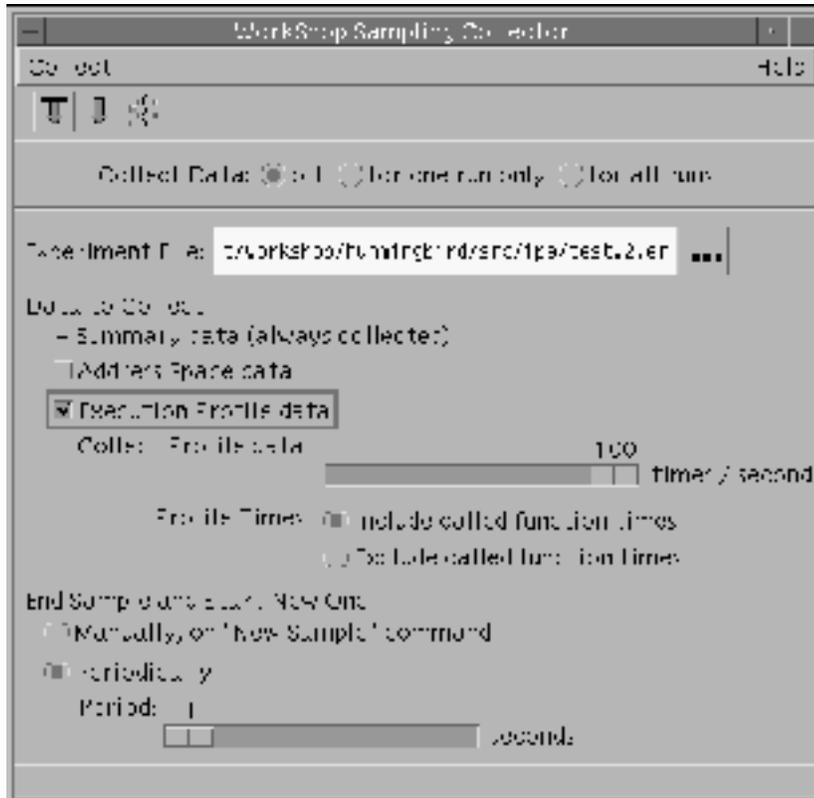


Figure 2-2 Sampling Collector Window

Collect menu	Provides commands to start program sampling, break program sampling, start the Sampling Analyzer, and exit the Sampling Collector.
Collect Data radio buttons	Turn the Sampling Collector off, on for one run, or on for all runs. If you select for one run only, the Sampling Collector turns off after an experiment is created. If you select for all runs, the Sampling Collector remains turned on even after the experiment is created. Turning the Sampling Collector on does not start data collection. Performance data is collected only when you execute the program in the Debugging window.
Experiment File text box	Accepts the complete path name of your experiment. You can either type in the path yourself or select it through the file chooser, which can be accessed by clicking the ellipsis button (...) to the right of the text box.
Address Space data checkbox	Collects process state address space data represented as a series of segments, each of which contains a number of pages. Such data allows the Sampling Analyzer to describe the status of each page and whether it was referenced or modified.
Execute Profile data checkbox	Collects information about the time consumed by functions, modules, and segments during the execution of the application.
Collect Profile data slider	Controls how many samples per second the Sampling Collector gathers.
Profile Times radio buttons	Determine whether called function times are included in or excluded from the sample data.
Manually, on "New Sample" command radio button	Summarizes data and starts a new sample whenever you choose Collect New Sample.
Periodically radio button	Collects data periodically at the interval set by the Period slider (default is one second).
Period slider	Determines the interval at which period data is collected.

## Controlling Profile Frequency

You can specify, in intervals of seconds, how often the Sampling Collector records profile PCs and call stacks. The valid range for the interval is 1 to 100 samples per second.

- Decrease the frequency if you want to decrease your data-collection overhead and don't mind having less precisely accurate data.
- Increase the frequency if you want more accurate data and don't mind higher data-collection overhead.

To specify an interval for gathering profiles, move the Collect Profile data slider to the number of samples per second you want.

## Marking Samples

The Sampling Collector interrupts data gathering to end one sample and begin another at these three points:

- At breakpoints
- When you choose Collect New Sample
- Periodically, at intervals you set with the slider

### *Breakpoints*

Because data is always summarized at breakpoints in the code, you can set breakpoints at any location at which you want to summarize collected data.

- If your application is compiled using the `-g` option, you can set breakpoints in those specific areas of the code pertaining to the information you want to collect. The breakpoints can be set at the beginning or end of a function, or on any line of code.
- If the application is compiled with the `-O` option, you can set breakpoints on functions, but not on specific source-code lines.

### *New Sample Command*

If you select the radio button labeled “Manually, on ‘New Sample’ command”, you can use the New Sample command on the Collect menu to check data at whatever points in the application you wish without setting a breakpoint in the code. This is useful if you are interested in measuring human interaction with the application—for example, the time it takes to choose a command from a menu or to type in a keyboard command.

### *Periodic Sampling*

If you select the radio button labeled “Periodically”, the Sampling Collector takes behavior data samples as you observe the running application, to give you a uniform view of the application's behavior. Use the Period slider at the bottom of the Sampling Collector window to define the intervals at which the Sampling Collector summarizes samples. The interval can be from 1 to 60 seconds.

# Understanding the Sampling Analyzer

The Sampling Analyzer measures, records, and analyzes the performance of an application. It can also compute an improved load order for functions in your application's address space and help you rebuild a tuned application.

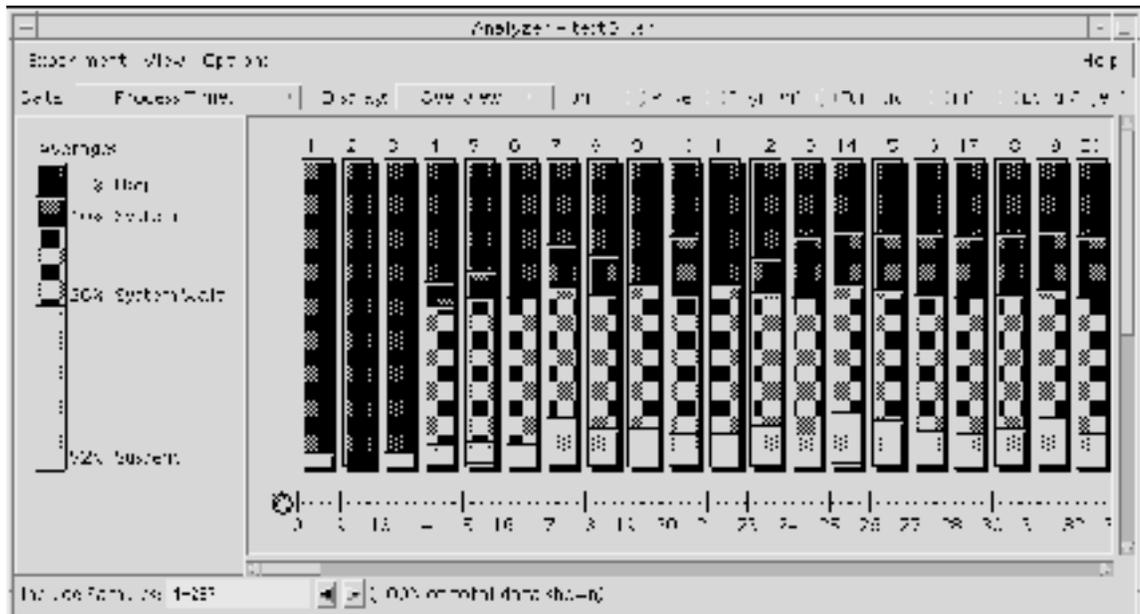


Figure 2-3 The Sampling Analyzer's main window

Experiment menu	Provides commands for loading, exporting, printing, and deleting experiments and for creating mapfiles.
View menu	Provides commands for selecting, sorting, finding, and showing data.
Options menu	Provides commands for altering column widths and histogram names
Help menu	Provides online help
Data list box	Determines the kind of performance data to be analyzed
Display list box	Sets the display method for the data being analyzed
Unit radio buttons	Select the type of unit to view in the display pane.
Average legend	Displays the average percentage of time spent in performance problem areas contained in experiment samples.

Sample display pane	Contains graphical analyses of collected data.
Include Samples text field	Displays samples, sample ranges, and/or numbers of displayed samples. The text box is editable.
Arrow buttons	Let you step through an experiment, incrementing or decrementing the sample number by one per click, and view program behavior at each sample.
Message area	Displays information about current actions.

The Sampling Analyzer examines an experiment record written by the Sampling Collector and displays it graphically on screen. The `er_export` utility converts the data of the experiment record to ASCII format, and the `er_print` utility prints the data of the current display to a file or printer. These two utilities are invoked from the Export and Print entries in the Experiment menu. They are not normally run from the command line.

## Loading an Experiment

You can load an experiment both as the Sampling Analyzer is opening and after the Sampling Analyzer is already open. By default, experiment data is shown using Overview display, but you can change the view to a Histogram, Cumulative, Address Space, or Statistics display, depending on the nature of the data.

To load an experiment as the Sampling Analyzer opens, double-click the experiment name in the Load Experiment dialog box that is displayed when the Sampling Analyzer window opens.

Or, navigate to the experiment name you want, and double-click it.

To load an experiment after the Sampling Analyzer is already open:

- 1. Choose Experiment > Load.**
- 2. Type the name of the experiment in the Name text box or double-click its entry in the file filter.**

Or, navigate to the experiment name you want, and double-click it.

## Selecting Data Types for Viewing

The Sampling Analyzer allows you to view different types of collected data. You can specify the kind of data that would help you improve your application's performance.

Select one of the following data types from the Data list box:

Process Times	Summary of process state transitions
User Time	Time spent in the user process state from the execution of instructions
System Wait Time	Time the process is sleeping in the kernel but is not in the suspend, idle, lock wait, text fault, or data fault state
System Time	Time the operating system spends executing system calls
Text Page Fault Time	Time spent faulting in text pages
Data Page Fault Time	Time spent faulting in data pages
Program Sizes	Sizes in bytes of the functions, modules, and segments of your application. Used in conjunction with Address Space data, this lets you examine the size of your application and helps you establish specific memory requirements
Address Space	Reference behavior of both text pages and data pages. Used in conjunction with Program Sizes data, it lets you examine the size of your application and helps you establish specific memory requirements
Execution Statistics	Overall statistics on the execution of the application

## Data Types and Display Options

Each data type can be viewed only in displays appropriate to its nature. Table 2-1 lists the display options associated with each data type:

**TABLE 2-1** Data Types and Corresponding Display Options

<b>Data Type</b>	<b>Display Option(s)</b>
Process Times	Overview
User Time	Histogram; Cumulative
System Wait Time	Histogram; Cumulative
System Time	Histogram; Cumulative
Text Page Fault Time	Histogram; Cumulative

**TABLE 2-1** Data Types and Corresponding Display Options *(continued)*

Data Type	Display Option(s)
Data Page Fault Time	Histogram; Cumulative
Program Sizes	Histogram
Address Space	Address Space
Execution Statistics	Statistics

## Selecting Display Options

The Sampling Analyzer associates each data type with one or two display options, depending on the nature of the actual data.

Select one of the display options shown in Table 2-2 from the Display list box.

**TABLE 2-2** Display Options for Specific Data Types

Display Option	Information Presented
Overview	The default display gives a high-level overview of performance behavior
Histogram	Summary of the amount of time spent executing functions, files, and load objects
Cumulative	Cumulative amount of time spent by a function, file, or load object, including the time spent in called functions, files, or segments
Address Space	Information about memory usage
Statistics	Aggregate data about performance and system resource usage

## The Overview Display

For each sample, the Overview display (see Figure 2-4) shows the amount of time the application spends in different process states. The Sampling Collector always gathers this data during the data collection process, so the Overview display appears by default whenever an experiment is loaded into the Sampling Analyzer.

The Overview display option:

- Provides a high-level overview of the performance behavior of an application
- Provides data about how your application's execution time breaks down into different performance areas, helping you identify CPU bottlenecks, I/O bottlenecks, and paging bottlenecks
- Shows how application performance changes during execution (for example, early parts of the execution might be I/O-bound, while later parts might be CPU-bound)

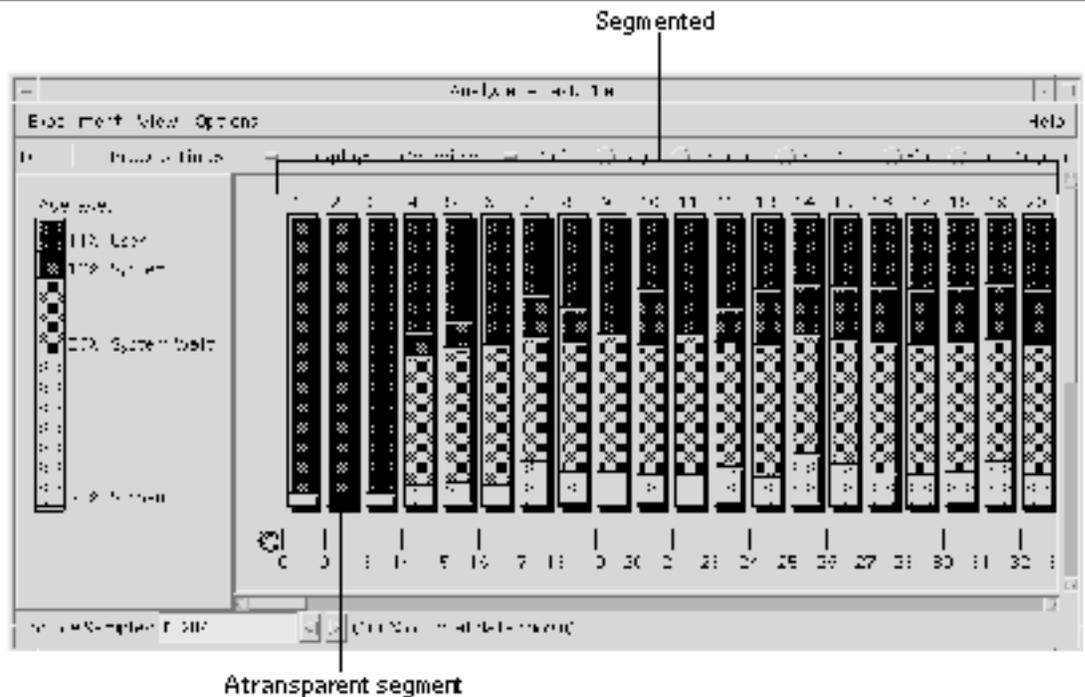


Figure 2-4 Overview Display

The Overview display contains numbered sample columns made up of segmented bars. Each column represents individual samples collected during an experiment.

The segments inside each column represent different performance areas. The height of each segment is proportional to the time spent in each performance area.

The shade that represents a specific performance area is consistent across all the sample columns in the experiment and across other experiments as well.

A transparent segment is a segment the same color as the foreground of the display pane. It represents performance areas too small to display individually. To see exactly which performance areas are contained in a transparent segment, click the segment's column and choose View Show Details to open the Sample Details dialog box (see Figure 2-5).

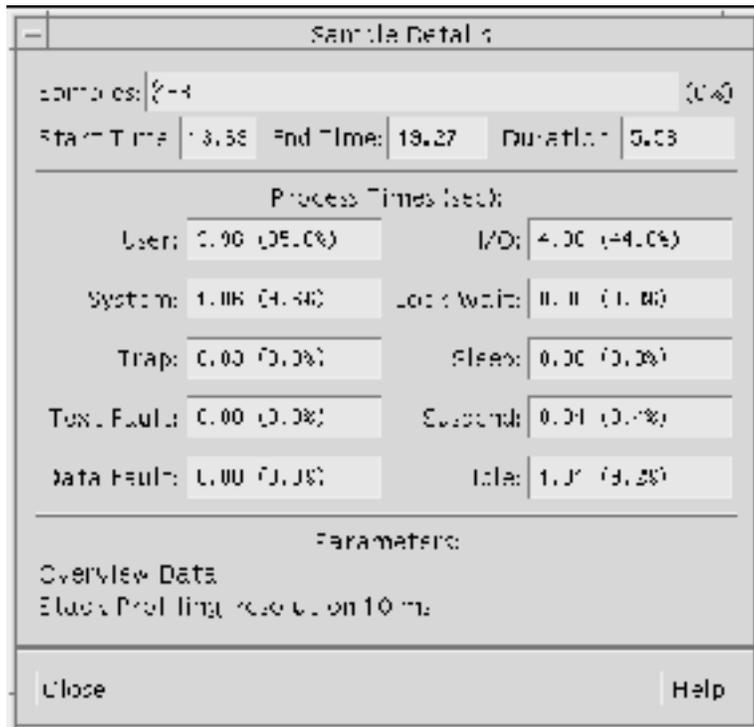


Figure 2-5 Sample Details Dialog Box

The fields in the dialog box contain the following information about the selected samples.

Samples	Samples currently selected and the percentage of the experiment they represent
Start Time	Start time of the sample
End Time	End time of the sample
Duration	Duration of the sample
User	Time spent executing application instructions
System	Time the operating system spent executing system calls
Trap	Time spent executing traps (automatic exceptions or memory faults)
Text Fault	The time spent faulting in text pages
Data Fault	The time spent faulting in data pages
I/O	Time spent in program I/O
Lock Wait	Time spent waiting for lightweight process locks to be released

Sleep	Time the program spent sleeping (due to any cause other than Text Fault, Data Fault, System Wait, or Lock Wait)
Suspend	Time spent suspended (including time spent in the debugger when it encounters breakpoints)
Idle	Time spent idle
Parameters	List of the data parameters collected for each sample (set in the Sampling Collector before beginning the experiment)

## The Histogram Display

The Histogram display (see Figure 2-6) shows how much time an application spends executing functions, files, or load objects.

The Histogram display option is available for the following data types:

- User Time
- System Wait Time
- System Time
- Text Page Fault Time
- Data Page Fault Time



To select which segments to include in the Histogram display, choose View Segments Included from Files to open the Segments Included from Files dialog. Click any segments and click Apply, or click the Select All button to select all segments.

To sort the Histogram display, choose View > Sort by and select either Values (descending by time value) or Names(alphabetically).

To search for specific names, choose View > Find to open the Find dialog box. Enter the search string in the text field and click Apply.

## The Cumulative Display

The Cumulative display (see Figure 2-7) shows the total execution time spent by a function, file, or load object, including time spent in called functions, files, or segments. All execution time accumulated in a descendant function is attributed to the parent function.

The Cumulative display is available for the following data types:

- User Time
- System Wait Time
- System Time
- Text Page Fault Time
- Data Page Fault Time

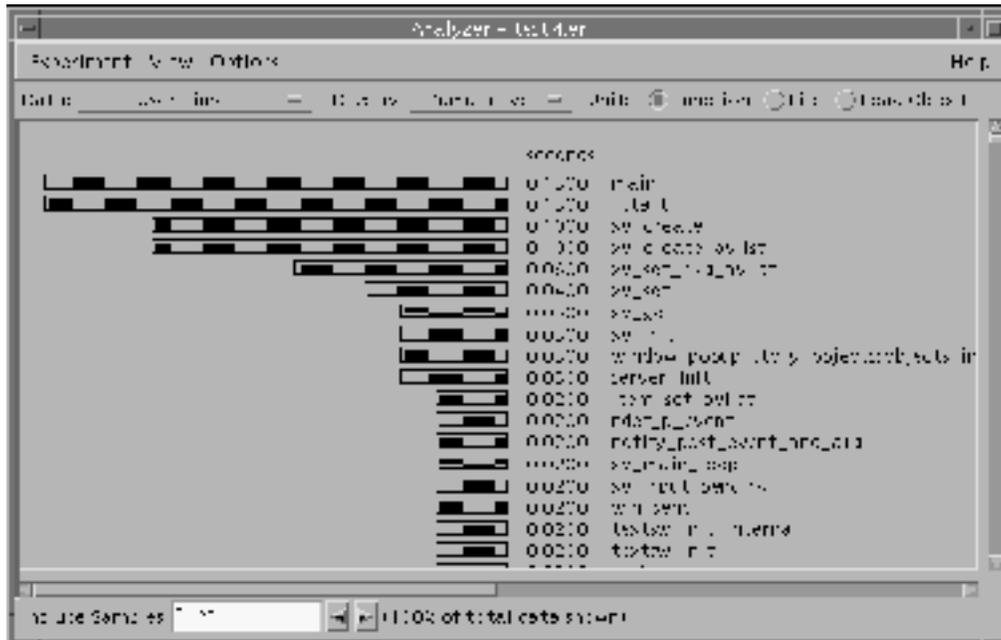


Figure 2-7 Cumulative Display

To view data at various levels of compilation granularity, choose one of the following unit types:

Function	Time your application spent executing functions
File	Time spent executing file-level units. This view is useful if your application has a large number of functions. All data for a single source file is displayed together. Note: If any part of the executable (including shared libraries) is not compiled with the <code>-g</code> option, the Sampling Collector may not have enough information to associate functions with their containing files.
Load Object	Time spent executing text segments.

You can select which samples to include in the Cumulative display in three ways:

- Type sample numbers directly into the Includes Samples text field: separate numbers with commas (*1,3,6*), and define ranges using a hyphen (*1-6*).
- Select the columns containing those samples while still in the Overview display.
- Choose either View > Select or View > Select None while still in the Overview display to include or exclude all samples in the experiment.

To select which segments to include in the Cumulative display, choose View > Segments Included from Files to open the Segments Included from Files

dialog. Click any segments and click Apply, or click the Select All button to select all segments.

To sort the Cumulative display, choose View > Sort by and select either Values (descending by time value) or Names (alphabetically).

To search for specific names, choose View > Find to open the Find dialog box. Enter the search string in the text field and click Apply.

## The Address Space Display

The Address Space display (see Figure 2-8) helps you identify memory that is most heavily used by your application (modified and referenced pages). This display option also identifies memory that is unused because the experiment did not exercise all of your application's functionality, or because your application has dead code or memory allocation problems.

The Address Space display option shows data only if you collect address-space data. If no address-space data was collected, a message to that effect will appear at the bottom of the Sampling Analyzer screen.

### *Memory Categories*

The Address Space display divides memory used by your application into the following categories:

Modified	A page written on during the execution of the application; may or may not be referenced
Referenced	A page read by your application or containing instructions that have been executed by your application
Unreferenced	A page neither modified nor referenced by the application

### *Address Space Display Layout*

The Address Space display (see Figure 2-8) is laid out in rows and columns that are made up of individual squares (pages) or rectangles (segments). The rows and columns are numbered to describe their address in memory. Gaps (shown as white space) represent a region of the address space that was not used by the application.

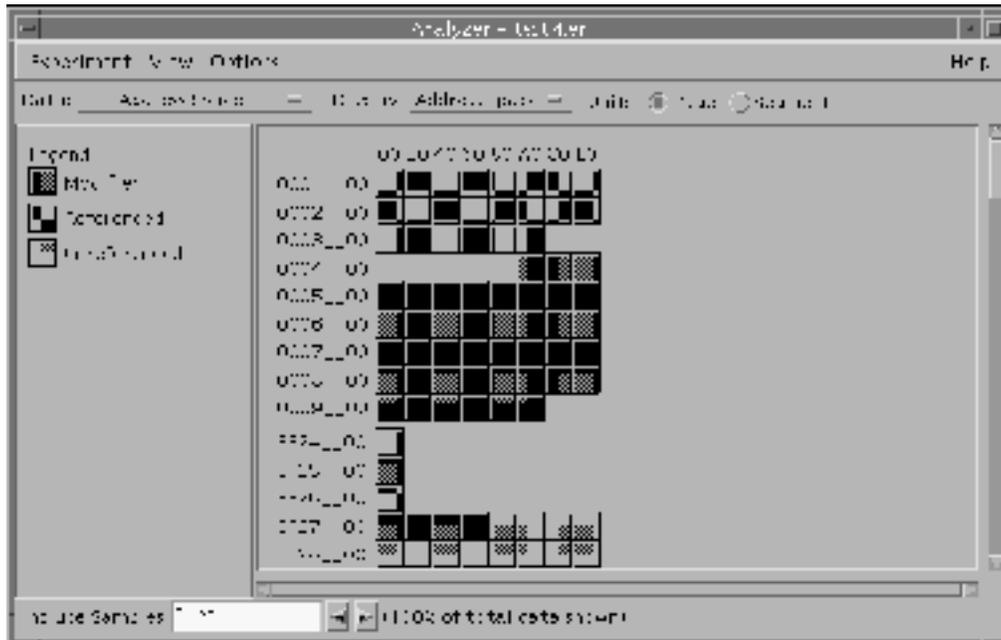


Figure 2-8 Address Space Display

Sun systems use either 4-Kbyte or 8-Kbyte pages. The address of a page is a multiple of 0x1000 (4 Kbytes in hexadecimal) or 0x2000 (8 Kbytes in hexadecimal).

To verify the page size of your system, go to a prompt and type:

```
% pagesize
```

The `pagesize` command returns the page size in bytes:

- 4096 (4-Kbyte pages)
- 8192 (8-Kbyte pages)

If the page size is 4 Kbytes, the number of pages per row is 16. If the page size is 8 Kbytes, the number of pages per row is 8.

You can determine the address of a page by combining the hexadecimal values of the row and column that contains the page. For example, if the page you are examining is in the fourth row (0004\_00) and the third column (20), then the address of that page is 00042000.

To view memory units at various levels of granularity in the Address Space display, select Page or Segment in the Unit type area.

Selected pages and segments are shadowed and raised to the left. If you keep the right mouse button pressed down over a selected page, the segment containing that page is also displayed and shadowed; likewise, if you keep the right mouse button pressed down over a selected segment, the pages contained within that segment are also displayed and selected.

To view information about the properties of a selected page or segment, choose View Show Details to open either the Page Properties or Segment Properties dialog, which displays the following information:

- Address
- Size of the page or size range of the segment in bytes
- Functions contained in the page or segment
- Name of the segment

To select which samples to include in the Address Space display, you can:

- Type sample numbers directly into the Includes Samples text field: separate numbers with commas (1,3,6), and define ranges using a hyphen (1-6).
- Select the columns containing those samples while still in the Overview display.
- Choose either View Select or View Select None while still in the Overview display to include or exclude all samples in the experiment.

## The Statistics Display

The Statistics display (see Figure 2-9) provides data about your application's overall performance and system resource usage (as opposed to the Histogram, Cumulative, and Address Space display options, which show data broken down by program components such as functions and pages). The information provided by the Statistics display is useful when you want to compare actual numerical values against any previous estimates you may have made.

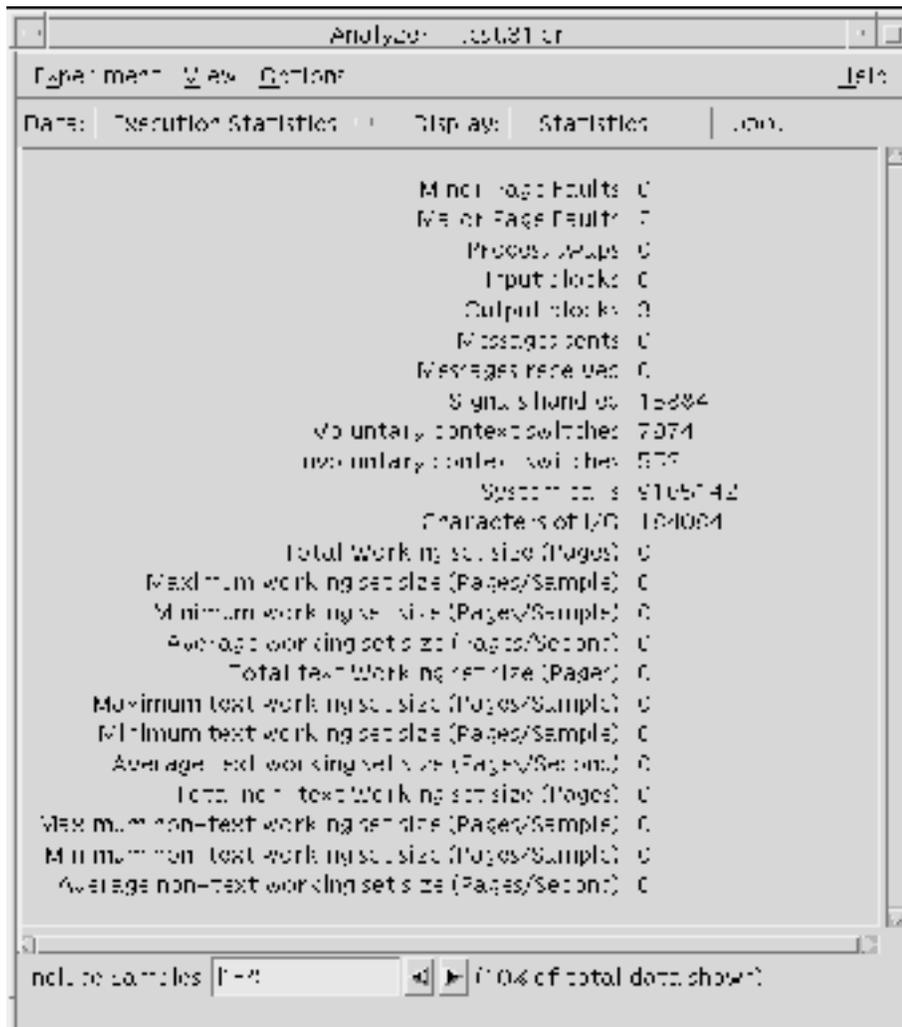


Figure 2-9 Display of Program Execution Statistics

The information needed to produce the Statistics display is always collected by the Sampling Collector during the data collection process, so you do not need to specify any particular data type to view information in this display. The Statistics display shows:

Minor Page Faults	The number of page faults serviced that do not require any physical I/O activity
Major Page Faults	The number of page faults serviced that require physical I/O activity (if non-zero, the Overview display shows text page or data page fault wait time)
Process swaps	The number of times a process is swapped out of main memory

Input blocks	The number of times a read() system call is performed on a non-character or special file
Output blocks	The number of times a write() system call is performed on a non-character or special file
Messages sent	The number of messages sent over sockets
Messages received	The number of messages received from sockets
Signals handled	The number of signals delivered or received
Voluntary context switches	The number of times a context switch occurred because a process voluntarily gave up the processor before its allotted time was completed, to wait for availability of a resource
Involuntary context switches	The number of times a context switch occurred because a higher-priority process became runnable, or because the current process exceeded its allotted time
System calls	The total number of system calls
Characters of I/O	The number of characters transferred in or out to a character device or file by read and write calls
Total address space size	Total size of the address space (in pages)
Maximum address space size	Maximum size of the address space (pages per sample)
Minimum address space size	Minimum size of the address space (pages per sample)
Average address space size	Average size of the address space (pages per sample)
Total text address space size	Total size of the text address space (pages)
Maximum text address space size	Maximum size of the text address space (pages per sample)
Minimum text address space size	Minimum size of the text address space (pages per sample)
Average text address space size	Average size of the text address space (pages per sample)

Total non-text address space size	Total size of the non-text address space (pages)
Maximum non-text address space size	Maximum size of the non-text address space (pages per sample)
Minimum non-text address space size	Minimum size of the non-text address space (pages per sample)
Average non-text address space size	Average size of the non-text address space (pages per sample)

---

**Note** - Workset sizes will be non-zero only if address-space data was collected.

---

You can select which samples to include in the Statistics display in three ways:

- Type sample numbers directly into the Includes Samples text field: separate numbers with commas (*1,3,6*), and define ranges using a hyphen (*1-6*).
- Select the columns containing those samples while still in the Overview display.
- Choose either View Select or View Select None while still in the Overview display to include or exclude all samples in the experiment.

## Reordering an Application

You might wish to reorder your application if (and only if) text page faults are consuming a large percentage of its time.

After the behavior data is collected, you can use the Sampling Analyzer to generate a mapfile containing an improved ordering of functions. The `-M` option passes the mapfile to the linker, which then relinks your application and produces a new executable application with a smaller text address space size.

After you have reordered your application, you can run a new experiment and compare the original version with the reordered one.

To reorder an application:

### 1. Compile the application using the `-xF` option.

The `-xF` option is required for reordering. This option causes the compiler to generate functions that can be relocated independently.

For C applications, type:

```
cc -xF -c a.c b.c
```

```
cc -o application_name a.o b.o
```

For C++ applications, type:

```
CC -xF -c a.cc b.cc
```

```
CC -o application_name a.o b.o
```

For Fortran applications, type:

```
f77 -xF -c a.f b.f
```

```
f77 -o application_name a.o b.o
```

If you see the following warning message, check any files that are statically linked, such as unshared object and library files, because these files may not have been compiled with the -xF option:

```
ld: warning: mapfile: text: .text% :function_name
```

```
object_file_name:
```

```
Entrance criteria not met, the named file,  
function_name, has not been compiled with the -xF option
```

2. **Load the application in Sun WorkShop for debugging.**
3. **Activate the Sampling Collector to collect performance data by choosing Windows > Sampling Collector from the Debugging window. Be sure to enable Address Space data collection.**
4. **Run the application in Sun WorkShop.**
5. **Load the specified experiment into the Sampling Analyzer.**
6. **Create a reordered map in the Sampling Analyzer by choosing Experiment > Create Mapfile. In the file chooser, enter the samples to be used, the mapfile directory, and the name of the mapfile to be created; and click OK.**  
The mapfile contains names of functions that have user CPU time associated with them. It specifies a function ordering that reduces the size of the text address space by sorting profiling data and function sizes in descending order. All functions not listed in the mapfile are placed after the listed functions.
7. **Link the application using the new mapfile.**

For C applications, type:

```
cc -Wl -M mapfile_name a.o b.o
```

For C++ applications, type:

```
CC -M omapfile_name a.o b.
```

For C applications, the `-M` option causes the compiler to pass `-M mapfile_name` to the linker.

For Fortran applications, type:

```
f77 -M mapfile_name a.o b.o
```

## Comparing Runtime Experiment Samples

The Sampling Analyzer lets you simultaneously view data in multiple displays, so you can compare samples in an experiment. With multiple displays, you can:

- View different sets of samples in the same display option. For example, you can compare Histogram displays of sample 8 and sample 11.
- View one set of samples in different display options. For example, you can view samples 1-6 in the Histogram display and, in a second window, view the same samples in the Cumulative display.
- Compare samples from different experiments.

To view multiple displays:

1. **Choose View > New Window to open a second Sampling Analyzer window.**
2. **In the new Sampling Analyzer window, choose data types, displays, and samples to examine, or load a second experiment if you wish.**

The new window does not inherit the settings of the first Sampling Analyzer window; it is set to the defaults with which the original Sampling Analyzer window started. Also, if you close or quit the original Sampling Analyzer window, all windows opened from that window close as well.

## Printing Experiments

If you want to save a record of an experiment, you can print experiment data to either a printer or a file. The Sampling Analyzer allows you to print:

- A plain-text version of the current display
- A text summary of the experiment that gives average sample times for each data type and shows how frequently functions, modules, and segments are used

To print a plain-text version of the current display:

1. **Choose Experiment > Print.**

2. **Select whether the data should be printed to a printer or a file, and indicate the printer name and number of copies, if applicable.**
3. **Click OK.**

To print a plain text summary of the experiment:

1. **Choose Experiment > Print Summary.**
2. **Select whether the summary data should be printed to a printer or a file, and indicate the printer name and number of copies, if applicable.**
3. **Click OK.**

## Exporting Experiment Data

The Sampling Analyzer allows you to export experiment data to an ASCII file to be used later by other programs.

To export experiment data to an ASCII file:

1. **Choose Experiment > Export to open the Export dialog box.**
2. **Enter the directory and the name of the experiment data file to be exported.**
3. **Click OK to save the experiment data under the given file name.**

## Loop Analysis Tools

---

The Fortran and 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 uses a graphical user interface (GUI). LoopReport is the command-line version of LoopTool.

This chapter is organized as follows:

- “Basic Concepts” on page 33
- “Setting Up Your Environment” on page 34
- “Creating a Loop Timing File” on page 34
- “Starting LoopTool” on page 37
- “Using LoopTool” on page 38
- “Starting LoopReport” on page 41
- “Compiler Hints” on page 45
- “How Optimization Affects Loops” on page 50

---

### Basic Concepts

LoopTool and LoopReport enable you to:

- Time all loops, whether serial or parallel.
- Produce a table of loop timings.
- Collect hints from the compiler during compilation.
- 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 reports loop runtimes in an ASCII file instead of a graphical display.

There are four basic steps for using LoopTool and LoopReport:

1. Setting up environment variables
2. Compiling the program with the options required to create a timing file for loop analysis
3. Running the program to generate a timing file
4. Invoke LoopTool or LoopReport on the timing file

---

**Note** - The examples in this section use the Fortran (f77 and f90) compilers. The options shown (such as `-xparallel`, `-Zlp`) also work for C.

---

---

## Setting Up Your Environment

Before running an executable compiled with `-Zlp`, 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`
```

You may want to put this command in a shell startup file (such as `.cshrc` or `.profile`).

---

## Creating a Loop Timing File

To create a loop timing file, you compile your program with compiler options that automatically parallelize and optimize your code (`-xparallel` and `-xO4`). You also add the `-Zlp` option to compile for LoopTool or LoopReport. When you run the program compiled with these options, Sun WorkShop creates a timing file for LoopTool or LoopReport to process.

The three compiler options are illustrated in this example:

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

---

**Note** - All examples apply to FORTRAN 77, Fortran 90, and C programs.

---

There are a number of other useful options for looking at and parallelizing loops:

---

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

---

## Other Compilation Options

Many combinations of compiler options work for LoopTool and LoopReport.

To compile for automatic parallelization, typical compilation switches are `-xparallel` and `-x04`. To compile for LoopTool and LoopReport, add `-Zlp`.

```
% f77 -x04 -xparallel -Zlp source_file
```

You can use either `-x03` or `-x04` with `-xparallel`. If you don't specify `-x03` or `-x04` but you do use `-xparallel`, then the compiler uses `-x03`. Table 3-1 summarizes how optimization level options are added for specific options.

**TABLE 3-1** Optimization Level Options and What They Imply

---

You type:	Bumped Up To:
<code>-xparallel</code>	<code>-xparallel -x03</code>
<code>-xparallel -Zlp</code>	<code>-xparallel -x03 -Zlp</code>
<code>-xexplicitpar</code>	<code>-xexplicitpar -x03</code>
<code>-xexplicitpar -Zlp</code>	<code>-xexplicitpar -x03 -Zlp</code>
<code>-Zlp</code>	<code>-xdepend -x03 -Zlp</code>

---

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

The Fortran compiler option `-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 parallelizes the 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 `-xdepend` option instructs the compiler to perform the data dependency analysis that it needs to do to identify loops. The option `-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> gamtab.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`.

## Run The Program

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

---

# Starting LoopTool

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

```
% looptool program &
```

If you start LoopTool without specifying a file, the Open File dialog box is displayed, allowing you to select a file to examine:

```
% looptool &
```

## 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 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 LoopTool window.

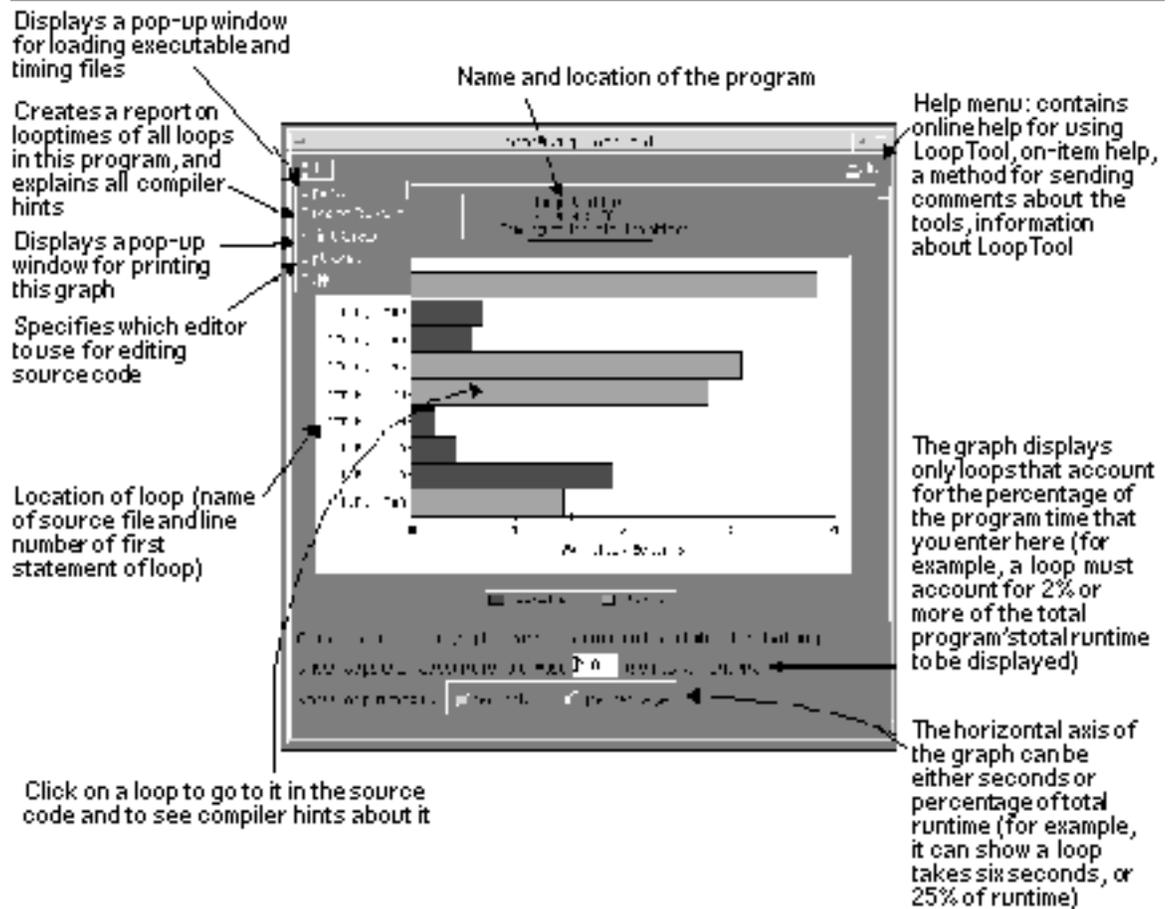


Figure 3-1 LoopTool Main Window

## Opening Files

To open executable and timing files, choose File Open in the main window.



For more information about printing see the Sun WorkShop online help.

## Choosing an Editor

Choose File Options in the main window to open the Options dialog box, where you can choose an editor for editing source code. The available editors are `vi`, `gnuemacs`, and `xemacs`.

---

**Note** - `vi` and `xemacs` are installed with LoopTool into your install directory (usually `/opt/SUNWsprow/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 in your search path in order for LoopTool to find it. For example, your `PATH` environment variable should include `/usr/local` if that's where `gnuemacs` 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 (see Figure 3-1) does two things:

- It brings up a window in which you can edit your source code (see Figure 3-3). The available editors are `vi`, `xemacs`, and `gnuemacs`.

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

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

- 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 Sun WorkShop online help compiler hints section. See also "Compiler Hints" on page 45, which explains the hints in detail.

Figure 3-3 shows an `xemacs` editor window with a loop selected, and a hint window with an explanation of a compiler hint.

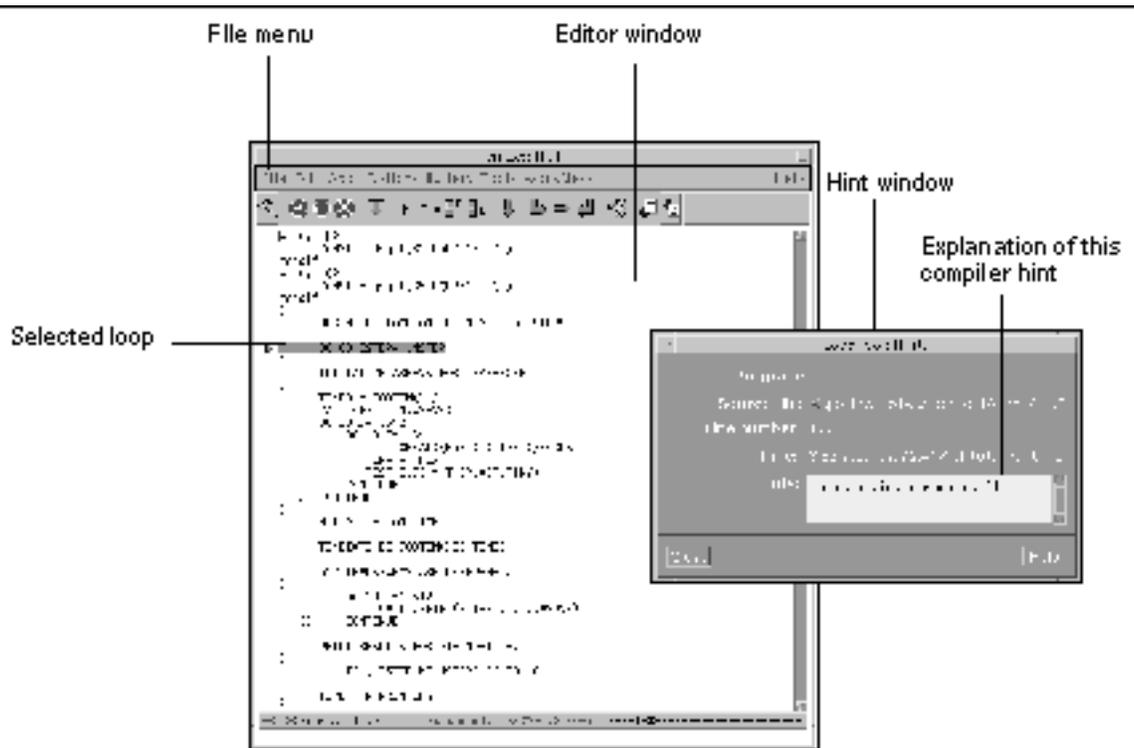


Figure 3-3 The Text Editor and Hints Windows



**Caution** - 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.

## Starting LoopReport

When it starts up, LoopReport expects to be given the name of your program. Type `loopreport` and the name of the program (an executable) you want examined.

```
% loopreport program
```

You can also start LoopReport with no file specified. However, if you invoke LoopReport without giving it the name of a program, it looks for a file named `a.out` in the current working directory.

```
% loopreport > a.out.loopreport
```

You can also direct the output into a file, or pipe it into another command:

```
% loopreport program > program.loopreport
% loopreport program | more
```

## Timing File

LoopReport also reads the *timing file* associated with your program. The timing file is created when you use the `-zlp` option, and contains information about loops. Typically, this file has a name of the format `program.looptimes`, and is found in the same directory as your program.

However, there are four ways to specify the location of a timing file. LoopReport chooses a timing file according to the rules listed below.

- If a timing file is named on the command line, LoopReport uses that file.

```
% loopreport program newtimes > program.loopreport
```

- If the command-line option `-p` is used, LoopReport looks in the directory named by `-p` for a timing file.

```
% loopreport program -p /home/timingfiles > program.loopreport
```

- If the environment variable `LVPATH` is set, LoopReport looks in that directory for a timing file.

```
% setenv LVPATH /home/timingfiles
% loopreport program > program.loopreport
```

- LoopReport writes the table of loop statistics to `stdout`—the standard output. You can also redirect the output to a file, or pipe it into another command:

```
% loopreport program > program.loopreport
% loopreport program | more
```

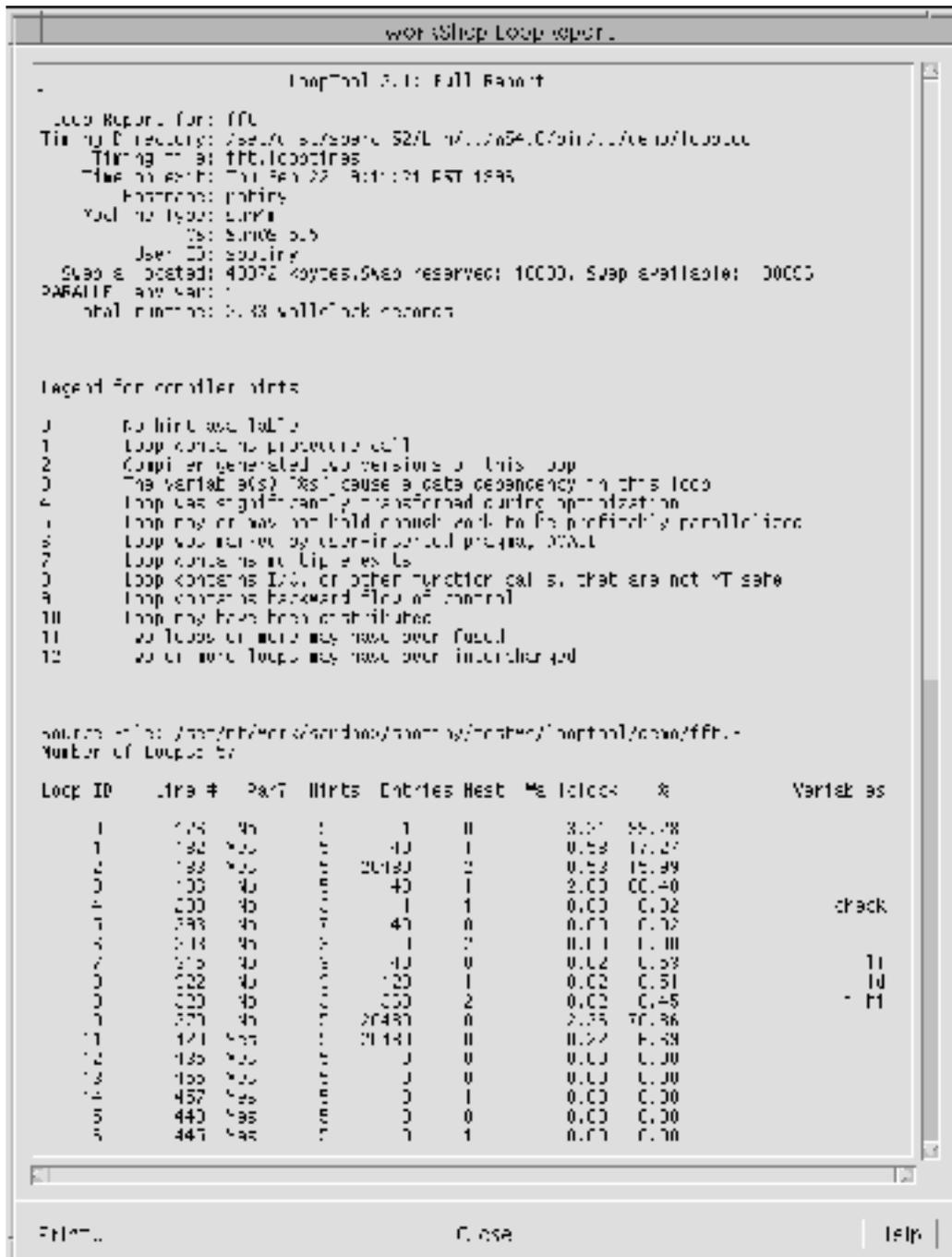


Figure 3-4 Sample Loop Report

# Fields in the Loop Report

The descriptions below apply equally to LoopTool's "Create Report" output and LoopReport's output.

The loop report contains the following information:

- **LoopID**

An arbitrary number, assigned by the compiler during compile time. This is just an internal loopID, useful for talking about loops, but not really related in any way to your program.

- **Line #**

The line number of the first statement of the loop in the source file.

- **Par?**

Par is short for "Parallelized by the compiler?" *Y* means that this loop was marked for parallelization; *N* means that the loop was not.

- **Hints**

Number corresponding to hint text in the "Legend for compiler hints" list.

- **Entries**

Number of times this loop was entered from above. This is distinct from the number of loop iterations, which is the total number of times a loop executes. For example, these are two loops in Fortran.

```
do 10 i=1,17
do 10 j=1,50
...some code...
10 continue
```

The first loop is entered once, and it iterates 17 times. The second loop is entered 17 times, and it iterates  $17 \cdot 50 = 850$  times.

- **Nest**

Nesting level of the loop. If a loop is a top-level loop, its nesting level is 0. If the loop is the child of another loop, its nesting level is 1.

For example, in this C code, the *i* loop has a nesting level of 0, the *j* loop has a nesting level of 1, and the *k* loop has a nesting level of 2.

```
for (i=0; i<17; i++)
for (j=0; j<42; j++)
for (k=0; k<1000; k++)
do something;
```

- **Wallclock**

The total amount of elapsed wallclock time spent executing this loop for the whole program. The elapsed time for an outer loop includes the elapsed time for an inner loop. For example:

```
for (i=1; i<10; i++)
  for (j=1; j<10; j++)
    do something;
```

The time assigned to the outer loop (the *i* loop) might be 10 seconds, and the time assigned to the inner loop (the *j* loop) might be 9.9 seconds.

- Percentage

The percentage of total program runtime measured as wallclock time spent executing this loop. As with wallclock time, outer loops are credited with time spent in loops they contain.

- Variables

The names of the variables that cause a data dependency in this loop. This field only appears when the compiler hint indicates that this loop suffers from a data dependency. A data dependency occurs when parallelization of a loop can not be done safely because the values computed in one iteration of a loop are used in another. The following illustrates a data dependency:

```
do i = 1, N
  a(i) = b(i) + c(i)
  b(i) = 2 * a(i + 1)
end do
```

If the example loop above is run in parallel, iteration 1 which recomputes  $b(1)$  based on the value of  $a(2)$ , may run after iteration 2 which has recomputed  $a(2)$ . The value of  $b(1)$  is determined by the new value of  $a(2)$  rather than the original value as would happen if the loop is not parallelized.

---

## Compiler Hints

LoopTool and LoopReport present hints about the optimizations applied to a particular loop, and about why a loop might not have been parallelized. The hints are heuristics gathered by the compiler during optimization. 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 3-2 lists the hints about optimizations applied to loops.

TABLE 3-2 Loop Optimization Hints

Hint #	Hint Definition
0	No hint available
1	Loop contains procedure call
2	Compiler generated two versions of this loop
3	The variable(s) " <i>list</i> " cause a data dependency in this loop
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, <code>DOALL</code>
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 could not 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. The variable(s) “*list*” cause a data dependency in this loop

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, “loop-carried data dependency.”

The compiler can often tell you the names of the variables that cause the loop-carried data dependency. If you rearrange your program to remove (or minimize) such dependencies, then the compiler can 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.

## 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 multithread-safe, whereas Hint 1 can refer to any sort of multithread-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.

---

# How Optimization Affects 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 within 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 transposing.

LoopTool and LoopReport attempt 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.

## 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.



## Traditional Profiling Tools

---

The tools discussed in this chapter are standard utilities for timing programs and obtaining performance data to analyze. `prof` and `gprof` are profiling tools that are provided with versions 2.5.1, 2.6, and Solaris 7 of the Solaris *SPARC Platform Edition* and Solaris *Intel Platform Edition*. `tcov` is a code coverage tool.

Some tools work on modules written in any programming language, while others work only with the C programming language. See the sections on each tool for more information about languages.

This chapter is organized into the following sections:

- “Basic Concepts” on page 53
- “Using `prof` to Generate a Program Profile” on page 54
- “Using `gprof` to Generate a Call Graph Profile” on page 56
- “Using `tcov` for Statement-Level Analysis” on page 59
- “`tcov` Enhanced—Statement-level Analysis” on page 64

---

### Basic Concepts

`prof`, `gprof`, and `tcov` extend the Sun WorkShop development environment to enable you to collect and use performance data. `prof` generates a program profile in a flat file. `gprof` generates a call graph profile. `tcov` generates statement-level information in a copy of the source file, annotated to show which lines are used and how often.

Table 4-1 lists these standard performance profiling tools.

TABLE 4-1 Performance Profiling Tools

Command	Action
<code>prof</code>	Generates a statistical profile of the CPU time used by a program, along with an exact count of the number of times each function is entered. This tool is included with the Solaris operating environment.
<code>gprof</code>	Generates a statistical profile of the CPU time used by a program, along with an exact count of the number of times each function is entered and the number of times each arc (caller-callee pair) in the program's call graph is traversed. This tool is included with the Solaris operating environment.
<code>tcov</code>	Generates exact counts of the number of times each statement in a program is executed. There are two versions of <code>tcov</code> : the original <code>tcov</code> and an enhanced <code>tcov</code> . They differ in the runtime support that generates the raw data used in the output. The original <code>tcov</code> obtains data from programs that are compiled with the <code>-xa</code> compiler option or the <code>-a</code> option (C++).

## Using `prof` to Generate a Program Profile

`prof` generates a statistical profile of the CPU time used by a program and counts the number of times each function in a program is entered. Different, or more detailed data, is provided by the call-graph profile and the code coverage tools.

Using `prof` involves three basic steps:

1. Compiling the program for `prof`
2. Running the program to produce a profile data file
3. Using `prof` to generate a report that summarizes the data

To compile a program for profiling with `prof`, use the `-p` option to the compiler. For example, to compile a C source file named `index.assist.c` for profiling, you would use this compiler command:

```
% cc -p -o index.assist index.assist.c
```

The compiler would produce a program called `index.assist`.

Now you could run the `index.assist` program. Each time you run the program, profiling data would be sent to a profile file called `mon.out`. Every time you run the program a new `mon.out` file would be created, overwriting the old version.

Finally, you would use the `prof` command to generate a report:

```
% index.assist
% ls mon.outmon.out
%prof index.assist
```

The `prof` output would resemble this example.

---

%Time	Seconds	Cumsecs	#Calls	msecs/call	Name
19.4	3.28	3.28	11962	0.27	compare_strings
15.6	2.64	5.92	32731	0.08	_strlen
12.6	2.14	8.06	4579	0.47	__doprint
10.5	1.78	9.84			mcount
9.9	1.68	11.52	6849	0.25	_get_field
5.3	0.90	12.42	762	1.18	_fgets
4.7	0.80	13.22	19715	0.04	_strcmp
4.0	0.67	13.89	5329	0.13	_malloc
3.4	0.57	14.46	11152	0.05	_insert_index_entry
3.1	0.53	14.99	11152	0.05	_compare_entry
2.5	0.42	15.41	1289	0.33	lmodt
0.9	0.16	15.57	761	0.21	_get_index_terms
0.9	0.16	15.73	3805	0.04	_strcpy
0.8	0.14	15.87	6849	0.02	_skip_space
0.7	0.12	15.99	13	9.23	_read
0.7	0.12	16.11	1289	0.09	ldivt
0.6	0.10	16.21	1405	0.07	_print_index

.. (The rest of the output is insignificant)

---

## Sample `prof` Output

This display points out that most of the program running time is spent in the `compare_strings` routine; after that, most of the time is spent in the `_strlen` library routine. To make improvements to this program, concentrate on the `compare_strings` function.

Let's interpret the results of the profiling run-through. The results are listed under these column headings:

`%Time`—The percentage of the total CPU time consumed by this routine of the program.

`Seconds`—The total CPU time accounted for by this function.

`Cumsecs`—A running sum of the number of seconds accounted for by this function and those listed above it.

`#Calls`—The number of times this routine is called.

`msecs/call`—The average number of milliseconds this routine consumes each time it is called.

`Name`—The name of the routine.

What results can be derived from the profile data? The `compare_strings` function consumes nearly 20% of the total time. To improve the runtime of `index.assist`, you could either improve the algorithm that `compare_strings` uses, or cut down the number of calls to `compare_strings`.

It is not obvious from the flat call graph that `compare_strings` is heavily recursive, but you can deduce this by using the call graph profile described in “Using `gprof` to Generate a Call Graph Profile” on page 56. In this particular case, improving the algorithm also reduces the number of calls.

---

**Note** - For Solaris 2.6 and Solaris 7, the profile of CPU time is accurate for programs that use multiple CPUs, but the fact that the counts are not locked may affect the accuracy of the counts for functions.

---

---

## Using `gprof` to Generate a Call Graph Profile

While the flat profile from `prof` can provide valuable data for performance improvements, a more detailed analysis can be obtained by using a call graph profile to display a list identifying which modules are called by other modules, and which modules call other modules. Sometimes removing calls altogether can result in performance improvements.

---

**Note** - `gprof` allocates the time within a function to the callers in proportion to the number of times each arc is traversed. Because all calls are not equivalent in performance, this behavior might lead to incorrect assumptions.

---

Like `prof`, `gprof` generates a statistical profile of the CPU time used by a program and it counts the number of times each function is entered. `gprof` also counts the number of times each *arc* in the program's call graph is traversed. An arc is a caller-callee pair.

---

**Note** - For Solaris 2.6 and Solaris 7, the profile of CPU time is accurate for programs that use multiple CPUs, but the fact that the counts are not locked may affect the accuracy of the counts for functions.

---

Using `gprof` involves three basic steps:

1. Compiling the program for `gprof`
2. Running the program to produce a profile data file
3. Using `gprof` to generate a report that summarizes the data

To compile the program for call graph profiling, use the `-xpg` option for the C compiler or the `-pg` option for the Fortran compiler. For example:

```
% cc -xpg -o index.assist index.assist.c
```

Now you could run the `index.assist` program. Each time that you run a program compiled for `gprof`, call-graph profile data is sent to a file called `gmon.out`. This file is recreated each time that you run the program.

Use the `gprof` command to generate a report of the results of the profile. The output from `gprof` can be large. You might find the report easier to read if you redirect it to a file. To redirect the output from `gprof` to a file named `/tmp/g.output`, you would use this sequence of commands:

```
% index.assist % ls gmon.outgmon.out % gprof index.assist > /tmp/g.output
```

The output from `gprof` consists of two major items:

- The full call graph profile, which shows fragments of output from a profiling run.
- The “flat” profile, similar to the summary the `prof` command supplies.

The output from `gprof` contains an explanation of what the various parts of the summary mean. `gprof` also identifies the granularity of the sampling:

```
granularity: each sample hit covers 4 byte(s) for 0.07% of 14.74 seconds
```

The “4 bytes” means resolution to a single instruction. The “0.07% of 14.74 seconds” means that each sample, representing ten milliseconds of CPU time, accounts for 0.07% of the run.

This is part of the call graph profile.

seconds	called/total parents		name	index
	called	total		
-----				
0.07	1	1	start	[1]
0.07	1	1	_main	[2]
0.70	760	760	_insert_index_entry	[3]
0.08	1	1	_print_index	[6]
0.90	761	761	_get_index_terms	[11]
0.06	762	762	_fgets	[13]
0.08	761	761	_get_page_number	[18]
0.40	761	761	_get_page_type	[22]
0.09	761	761	_skip_start	[24]
0.08	761	761	_get_index_type	[26]
0.00	761	820	_insert_page_entry	[34]
-----				
		10392	_insert_index_entry	[3]
0.70	760	760	_main	[2]
0.70	760	10392	_insert_index_entry	[3]
0.53	11152	11152	_compare_entry	[4]
0.02	59	112	_free	[38]
0.00	59	820	_insert_page_entry	[34]

---

Assuming there are 761 lines of data in the input file to the `index.assist` program, the following conclusions can be drawn:

- `fgets` is called 762 times. The last call to `fgets` returns an end-of-file.
- The `insert_index_entry` function is called 760 times from `main`.
- In addition to the 760 times `insert_index_entry` is called from `main`, `insert_index_entry` also calls itself 10,392 times. `insert_index_entry` is heavily recursive.
- `compare_entry` (which is called from `insert_index_entry`) is called 11,152 times, which is equal to 760+10,392 times. There is one call to `compare_entry` for every time `insert_index_entry` is called. This is as it should be. If there were a discrepancy in the number of calls, you could suspect some problem in the program logic.
- `insert_page_entry` is called 820 times in total: 761 times from `main` while the program is building index nodes, and 59 times from `insert_index_entry`. This frequency indicates there are 59 duplicated index entries, so their page number entries are linked into a chain with the index nodes. The duplicate index entries are then freed; hence the 59 calls to `free()`.

---

## Using `tcov` for Statement-Level Analysis

`tcov` gives line-by-line information on how a program executes. It produces a copy of the source file, annotated to show which lines are used and how often. It also summarizes information about basic blocks. `tcov` does not product any time-based data.

Using `tcov` involves three basic steps:

1. Compiling the program to produce a `tcov` experiment
2. Running the experiment
3. Using `tcov` to create summaries of execution counts for each statement in the program

## Compiling for tcov

To compile a program for code coverage, use the `-xa` option to the C compiler. Using a program named `index.assist` as an example, you compile for use with `tcov` with this command:

```
% cc -xa -o index.assist index.assist.c
```

You use the `-a` compiler option with the C++ or `f77` compilers.

The C compiler generates an `index.assist.d` file, containing database entries for the basic blocks present in `index.assist.c`. When the program `index.assist` is run until completion, the compiler updates the `index.assist.d` file.

---

**Note** - `tcov` works with both C and C++ programs, but `tcov` does not support files that contain `#line` or `#file` directives. `tcov` does not enable test coverage analysis of the code in the `#include` header files. Applications compiled with `-xa` (C), `-a` (other compilers), and `+d` (C++) run slower than normal. The `+d` option inhibits expansion of C++ inline functions, and updating the `.d` file for each execution takes considerable time.

---

The `index.assist.d` file is created in the directory specified by the environment variable `TCOVDIR`. If `TCOVDIR` is not set, `index.assist.d` is created in the current directory.

Having compiled `index.assist.c`, you could run `index.assist`:

```
% index.assist
% ls *.d
index.assist.d
```

Now you could run `tcov` to produce a file containing the summaries of execution counts for each statement in the program. `tcov` uses the `index.assist.d` file to generate an `index.assist.tcov` file containing an annotated list of your code. The output shows the number of times each source statement is executed. At the end of the file, there is a short summary.

```
% tcov index.assist.c
% ls *.tcov
index.assist.tcov
```

This small fragment of the C code from one of the modules of `index.assist` shows the `insert_index_entry` function, which is called so recursively.

```
    struct index_entry *
11152 -> insert_index_entry(node, entry)
    struct index_entry *node;
    struct index_entry *entry;
    {
        int result;
```

```

int level;

result = compare_entry(node, entry);
if (result == 0) {          /* exact match */
    /* Place the page entry for the duplicate */
    /* into the list of pages for this node */
59 ->    insert_page_entry(node, entry->page_entry);
        free(entry);
        return(node);
    }

11093 -> if (result > 0)      /* node greater than new entry -- */
        /* move to lesser nodes */
3956 ->    if (node->lesser != NULL)
3626 ->        insert_index_entry(node->lesser, entry);
        else {
330 ->        node->lesser = entry;
            return (node->lesser);
        }
    else /* node less than new entry -- */
        /* move to greater nodes */
7137 ->    if (node->greater != NULL)
6766 ->        insert_index_entry(node->greater, entry);
        else {
371 ->        node->greater = entry;
            return (node->greater);
        }
    }
}

```

The numbers to the side of the C code show how many times each statement was executed. The `insert_index_entry` function is called 11,152 times.

`tcov` places a summary like this at the end of the annotated program listing for `index.assist.tcov`:

Top 10 Blocks	
Line	Count
240	21563
241	21563
245	21563
251	21563
250	21400
244	21299
255	20612
257	16805
123	12021

	124	11962
77	Basic blocks in this file	
55	Basic blocks executed	
71.43	Percent of the file executed	
	439144	Total basic block executions
	5703.17	Average executions per basic block

A program compiled for code coverage analysis can be run multiple times (with potentially varying input); `tcov` can be used on the program after each run to compare behavior.

## Creating `tcov` Profiled Shared Libraries

It is possible to create a `tcov` profiled shareable library and use it in place of one where binaries have already been linked. Include the `-xa (C)` or `-a` (other compilers) option when creating the shareable libraries. For example:

```
%cc -G -xa -o foo.so.1 foo.o
```

This command includes a copy of the `tcov` profiling subroutines in the shareable libraries, so that clients of the library do not need to relink. If a client of the library is also linked for profiling, then the version of the `tcov` subroutines used by the client is used to profile the shareable library.

## Locking Files

`tcov` uses a simple file-locking mechanism for updating the block coverage database in the `.d` files. It employs a single file, `/tmp/tcov.lock`, for this purpose. Consequently, only one executable compiled with `-xa (C)` or `-a` (other compilers) should be running on the system. If the execution of the program compiled with the `-xa` (or `-a`) option is manually terminated, then the `/tmp/tcov.lock` file must be deleted manually.

Files compiled with the `-xa` or `-a` option call the profiling tools subroutines automatically when a program is linked for `tcov` profiling. At program exit, these subroutines combine the information collected at runtime for file `xyz.f` with the existing profiling information stored in file `xyz.d`. To ensure this information is not

corrupted by several people simultaneously running a profiled binary, a `xyz.d.lock` lock file is created for `xyz.d` for the duration of the update. If there are any errors in opening or reading `xyz.d` or its lock file, or if there are inconsistencies between the runtime information and the stored information, then the information stored in `xyz.d` is not changed.

An edit and recompile of `xyz.d` may change the number of counters in `xyz.d`. This is detected if an old profiled binary is run.

If too many people are running a profiled binary, the lock cannot be obtained. An error message similar to the following is displayed after a delay of several seconds:

```
tcov_exit: Failed to create lock file "/tmp_mnt/net/rbbb/export/home/src/newpattern/foo.d.lock" for coverage data file "/tmp_mnt/net/rbbb/export/home/src/newpattern/foo.d" after 5 tries. Is somebody else running this binary?
```

The stored information is not updated. This locking is safe across a network. Since locking is performed on a file-by-file basis, other files may be correctly updated.

The profiling subroutines attempt to deal with automounted file systems that have become inaccessible. They still fail if the file system containing a coverage data file is mounted with different names on different machines, or if the user running the profiled binary does not have permission to write to either the coverage data file or the directory containing it. Be sure all the directories are uniformly named and writable by anyone expected to run the binary.

## Errors Reported by `tcov` Runtime

The following error messages may be reported by the `tcov` runtime routines:

```
tcov_exit: Could not open coverage data file 'coverage data file name' because 'system error message'
```

The user running the binary lacks permission to read or write to the coverage data file. The problem also occurs if the coverage data file has been deleted.

```
tcov_exit: Could not write coverage data file 'coverage data file name' because 'system error message'
```

The user running the binary lacks permission to write to the directory containing the coverage data file. The problem also occurs if the directory containing the coverage data file is not mounted on the machine where the binary is being run.

```
tcov_exit: Failed to create lock file 'lock file name' for coverage data file 'coverage data file name'
```

Too many users are trying to update a coverage data file at the same time. The problem also occurs if a machine has crashed while a coverage data file is being updated, leaving behind a lock file. In the event of a crash, the longer of the two files should be used as the post-crash coverage data file. Manually remove the lock file.

tcov\_exit: Stdio failure, probably no memory left.

No memory is available, and the standard I/O package will not work. You cannot update the coverage data file at this point.

tcov\_exit: Coverage data file path name too long (*length* characters) '*coverage data file name*'

The lock file name contains six more characters than the coverage data file name; therefore, the derived lock file name may not be legal.

tcov\_exit: Coverage data file '*coverage data file name*' is too short. Is it out of date?

A library or binary with tcov profiling enabled is simultaneously being run, edited, and recompiled. The old binary expects a coverage data file of a certain size, but the editing often changes that. If the compiler creates a new coverage data file at the same time the old binary is trying to update the old coverage data file, the binary may see an apparently empty or corrupt coverage file.

---

## tcov Enhanced—Statement-level Analysis

Like the original tcov, tcov Enhanced gives line-by-line information on how a program executes. It produces a copy of the source file, annotated to show which lines are used and how often. It also gives a summary of information about basic blocks. tcov Enhanced works with both C and C++ source files.

tcov Enhanced overcomes some of the shortcomings of the original tcov:

- It provides more complete support for C++.
- It supports code found in `#include` header files and corrects a flaw that obscured coverage numbers for template classes and functions.
- Its runtime is more efficient than the original tcov runtime.
- It is supported for all the platforms the compilers support.

## Compiling for tcov Enhanced

To use tcov Enhanced, follow the same basic steps as the original tcov:

1. Compile a program for a tcov Enhanced experiment.
2. Run the experiment.
3. Analyze the results using tcov.

For the original `tcov`, you compile with the `-xa` option. To compile a program for code coverage with `tcov Enhanced`, you use the `-xprofile=tcov` option for all compilers. Using a program named `index.assist` as an example, you would compile for use with `tcov Enhanced` with this command:

```
% cc -xprofile=tcov -o index.assist index.assist.c
```

`tcov Enhanced`, unlike `tcov`, does not produce a `.d` file. The coverage data file is not created until the program is run. Then one coverage data file is produced as opposed to one file for each module compiled for coverage analysis.

After you compiled `index.assist.c`, you would run `index.assist` to create the profile file:

```
% index.assist

% ls -dF *.profile

index.assist.profile/

% ls *.profile

tcovd
```

By default, the name of the directory where the `tcovd` file is stored is derived from the name of the executable. Furthermore, that directory is created in the directory the executable was run in (the original `tcov` created the `.d` files in the directory where the modules were compiled).

The directory where the `tcovd` file is stored is also known as the “profile bucket.” The profile bucket can be overridden by using the `SUN_PROFDATA` environment variable. This may be useful if the name of the executable is not the same as the value in `argv[0]` (for example, the invocation of the executable was through a symbolic link with a different name).

You can also override the directory where the profile bucket is created. To specify a location different from the run directory, specify the path using the `SUN_PROFDATA_DIR` environment variable. Absolute or relative pathnames can be specified in this variable. Relative pathnames are relative to the program’s current working directory at program completion.

`TCOVDIR` is supported as a synonym for `SUN_PROFDATA_DIR` for backward compatibility. Any setting of `SUN_PROFDATA_DIR` causes `TCOVDIR` to be ignored. If both `SUN_PROFDATA_DIR` and `TCOVDIR` are set, a warning is displayed when the profile bucket is generated. `SUN_PROFDATA_DIR` takes precedence over `TCOVDIR`. The variables are used at runtime by a program compiled with `-xprofile=tcov`, and are used by the `tcov` command.

---

**Note** - This scheme is also used by the profile feedback mechanism.

---

Now that some coverage data has been produced, you could generate a report that relates the raw data back to the source files:

```
% tcov -x index.profile index.assist.c
% ls *.tcov
index.assist.c.tcov
```

The output of this report is identical to the one from the previous example (for the original `tcov`).

## Creating Profiled Shared Libraries

Creating shared libraries for use with `tcov` Enhanced is accomplished by using the analogous compiler options:

```
% cc -G -xprofile=tcov -o foo.so.1 doo.o
```

## Locking Files

`tcov` Enhanced uses a simple file-locking mechanism for updating the block coverage data file. It employs a single file created in the same directory as the `tcovd` file. The file name is `tcovd.temp.lock`. If execution of the program compiled for coverage analysis is manually terminated, then the lock file must be deleted manually.

The locking scheme does an exponential back-off if there is a contention for the lock. If, after five tries, the `tcov` runtime cannot acquire the lock, it gives up and the data is lost for that run. In this case, the following message is displayed:

```
tcov_exit: temp file exists, is someone else running this executable?
```

## `tcov` Directories and Environment Variables

When you compile a program for `tcov` and run the program, the running program generates a profile bucket. If a previous profile bucket exists, the program uses that profile bucket. If a profile bucket does not exist, it creates the profile bucket.

The profile bucket specifies the directory where the profile output is generated. The name and location of the profile output are controlled by defaults that you can modify with environment variables.

---

**Note** - `tcov` uses the same defaults and environment variables that are used by the compiler options that you use to gather profile feedback: `-xprofile=collect` and `-xprofile=use`. For more information about these compiler options, see the documentation for your compiler.

---

The default profile bucket the program creates is named after the executable with a `.profile` extension and is created in the directory where the executable is run. Therefore, if you are in `/home/joe`, and run a program called `/usr/bin/xyz`, the default behavior is to create a profile bucket called `xyz.profile` in `/home/joe`.

The environment variables that you set to modify the defaults are:

- `SUN_PROFDATA`

Can be used to specify the name of the profile bucket at runtime. The value of this variable is always appended to the value of `SUN_PROFDATA_DIR` if both variables are set.

- `SUN_PROFDATA_DIR`

Can be used to specify the name of the directory containing the profile bucket. It is used at runtime and in the `tcov` command.

- `TCOVDIR`

`TCOVDIR` is supported as a synonym for `SUN_PROFDATA_DIR` to maintain backward compatibility. Any setting of `SUN_PROFDATA_DIR` causes `TCOVDIR` to be ignored. If both `SUN_PROFDATA_DIR` and `TCOVDIR` are set, a warning is displayed when the profile bucket is generated.

`TCOVDIR` is used at runtime by a program compiled with `-xprofile=tcov` and it is used by the `tcov` command.

## Overriding the Default Definitions

To override the default, use the environment variables to change the profile bucket:

1. **Change the name of the profile bucket by using the `SUN_PROFDATA` environment variable.**
2. **Change the directory where the profile-bucket is placed by using the `SUN_PROFDATA_DIR` environment variable.**

The environment variables override the default location and name of the profile bucket. Both can be overridden independently. For example, if you only choose to set

SUN\_PROFDATA\_DIR, the profile bucket will go into the directory where you set SUN\_PROFDATA\_DIR. The default name (which is the executable name followed by a “.profile”) will still be the name used for the profile bucket.

## Absolute and Relative Pathnames

There are two forms of directories you can specify by using SUN\_PROFDATA\_DIR on the Profile Feedback compile line: absolute pathnames (which start with a '/'), and relative pathnames. If you use an absolute pathname, the profile bucket is dropped into that directory. If you specify a relative pathname, then it is relative to the current working directory where the executable is being run.

For example, if you are in /home/joe and run a program called /usr/bin/xyz with SUN\_PROFDATA\_DIR set to . . . , then the profile bucket is called /home/joe/./xyz.profile. The value specified in the environment variable was relative, and therefore, it was relative to /home/joe. Also, the default profile bucket name is used, which is named after the executable.

## TCOVDIR and SUN\_PROFDATA\_DIR

The previous version of tcov (enabled by compiling with the -xa or -a flag) used an environment variable called TCOVDIR. TCOVDIR specified the directory where the tcov counter files go to instead of next to the source files. To retain compatibility with this environment variable, the new SUN\_PROFDATA\_DIR environment variable behaves like the TCOVDIR environment variable. If both variables are set, a warning is output and SUN\_PROFDATA\_DIR takes precedence over TCOVDIR.

## For -xprofile=tcov

By default the profile bucket is called <argv[0]>.profile in the current directory.

If you set SUN\_PROFDATA, the profile bucket is called \$SUN\_PROFDATA, wherever it is located.

If you set SUN\_PROFDATA\_DIR, the profile bucket is placed in the specified directory.

SUN\_PROFDATA and SUN\_PROFDATA\_DIR are independent. If both are specified, the profile bucket name is generated by using SUN\_PROFDATA\_DIR to find the profile bucket and, SUN\_PROFDATA is used to name the profile bucket in that directory.

A UNIX process can change its current working directory during the execution of a program. The current working directory used to generate the profile bucket is the current working directory of the program at exit. In the rare case where a program actually does change its current working directory during execution, you can use the environment variables to control where the profile bucket is generated.

## For the `tcov` Program

The `-xprofile=bucket` option specifies the name of the profile bucket to use for the `tcov` profile. `SUN_PROFDATA_DIR` or `TCOVDIR` are prepended to this argument, if they are set.



## Lock Analysis Tool

---

LockLint is a command line utility that analyzes the use of mutex and multiple readers/single writer locks, and looks for inconsistent use of these locking techniques.

This chapter is organized as follows:

- “Basic Concepts” on page 71
- “LockLint Overview” on page 72
- “Collecting Information for LockLint ” on page 74
- “LockLint User Interface” on page 74
- “How to Use LockLint” on page 75
- “Source Code Annotations” on page 86

---

### Basic Concepts

In the multithreading model, a *process* consists of one or more threads of control that share a common address space and most other process resources. Threads must acquire and release locks associated with the data they share. If they fail to do so, a *data race* may ensue—a situation in which a program may produce different results when run repeatedly with the same input.

Data races are easy problems to introduce. Simply accessing a variable without first acquiring the appropriate lock can cause one. Data races are generally very difficult to find. Symptoms generally manifest themselves only if two threads access the improperly protected data at nearly the same time; hence a data race may easily run correctly for months without showing any signs of a problem. It is extremely difficult to exhaustively test all concurrent states of a program for even a simple

multithreaded program, so conventional testing and debugging are not an adequate defense against data races.

Most processes share several resources. Operations within the application may require access to more than one of those resources. This means that the operation needs to grab a lock for each of the resources before performing the operation. If different operations use a common set of resources, but the order in which they acquire the locks is inconsistent, there is a potential for *deadlock*. For example, the simplest case of deadlock occurs when two threads hold locks for different resources and each thread tries to acquire the lock for the resource held by the other thread.

---

## LockLint Overview

When analyzing locks and how they are used, LockLint detects a common cause of data races: failure to hold the appropriate lock while accessing a variable.

Table 5-1, Table 5-2, and Table 5-3 list the routines of the Solaris and POSIX libthread APIs recognized by LockLint.

**TABLE 5-1** Reader -Writer Locks

<b>Solaris</b>	<b>Kernel (Solaris only)</b>
<code>rw_rdlock, rw_wrlock</code>	<code>rw_enter</code>
<code>rw_unlock rw_tryrdlock,</code> <code>rw_trywrlock</code>	<code>rw_exit</code>
	<code>rw_tryenter</code>
	<code>rw_downgrade</code>
	<code>rw_tryupgrade</code>

**TABLE 5-2** Condition Variables

<b>Solaris</b>	<b>POSIX</b>	<b>Kernel (Solaris only)</b>
cond_broadcast	pthread_cond_broadcast	cv_broadcast
cond_wait	pthread_cond_wait	cv_wait
cond_timedwait	pthread_cond_timedwait	cv_wait_sig
cond_signal	pthread_cond_signal	cv_wait_sig_swap
		cv_timedwait
		cv_timedwait_sig
		cv_signal

**TABLE 5-3** Mutex (Mutual Exclusion) Locks

<b>Solaris</b>	<b>POSIX</b>	<b>Kernel (Solaris only)</b>
mutex_lock	pthread_mutex_lock	mutex_enter
mutex_unlock	pthread_mutex_unlock	mutex_exit
mutex_trylock	pthread_mutex_trylock	mutex_tryenter

Additionally, LockLint recognizes the structure types shown in Table 5-4.

**TABLE 5-4** Lock Structures

<b>Solaris</b>	<b>POSIX</b>	<b>Kernel (Solaris only)</b>
mutex_t	pthread_mutex_t	kmutex_t
rwlock_t		krwlock_t

LockLint reports several kinds of basic information about the modules it analyzes, including:

- Locking side effects of functions. Unknown side effects can lead to data races or deadlocks.

- Accesses to variables that are not consistently protected by at least one lock, and accesses that violate assertions about which locks protect them. This information can point to a potential data race.
- Cycles and inconsistent lock-order acquisitions. This information can point to potential deadlocks.
- Variables that were protected by a given lock. This can assist in judging the appropriateness of the chosen *granularity*, that is, which variables are protected by which locks.

LockLint provides subcommands for specifying assertions about the application. During the analysis phase, LockLint reports any violation of the assertions.

---

**Note** - Add assertions liberally, and use the analysis phase to refine assertions and to make sure that new code does not violate the established locking conventions of the program.

---

---

## Collecting Information for LockLint

The compiler gathers the information used by LockLint. More specifically, you specify a command-line option, `-zll`, to the C compiler to generate a `.ll` file for each `.c` source code file. The `.ll` file contains information about the flow of control in each function and about each access to a variable or operation on a mutex or readers-writer lock.

---

**Note** - No `.o` file is produced when you compile with the `-zll` flag.

---

---

## LockLint User Interface

There are two ways for you to interact with LockLint: source code annotations and the command-line interface.

- Source code annotations are assertions and `NOTES` that you place in your source code to pass information to LockLint. LockLint can verify certain assertions about the states of locks at specific points in your code, and annotations can be used to verify that locking behavior is correct or avoid unnecessary error warnings.

See “Source Code Annotations” on page 86 for more information.

- Alternatively, you can use LockLint subcommands to load the relevant `.ll` files and make assertions. This interface to LockLint consists of a `lock_lint`

command and a set of subcommands that you specify on the `lock_lint` command line.

The important features of the `lock_lint` subcommands are:

- You can exercise a few additional controls that have no corresponding annotations.
- You can make a number of useful queries about the functions, variables, function pointers, and locks in your program.

LockLint subcommands help you analyze your code and discover which variables are not consistently protected by locks. You may make assertions about which variables are supposed to be protected by a lock and which locks are supposed to be held whenever a function is called. Running the analysis with such assertions in place will show you where the assertions are violated.

See Appendix A.

Most programmers report that they find source code annotations preferable to command-line subcommands. However, there is not always a one-to-one correspondence between the two.

---

## How to Use LockLint

Using LockLint consists of three steps:

1. Setting up the environment for using LockLint
2. Compiling the source code to be analyzed, producing the LockLint database files (.ll files)
3. Using the `lock_lint` command to run a LockLint session

These steps are described in the rest of this section.

Figure 5-1 shows the flow control of tasks involved in using LockLint:

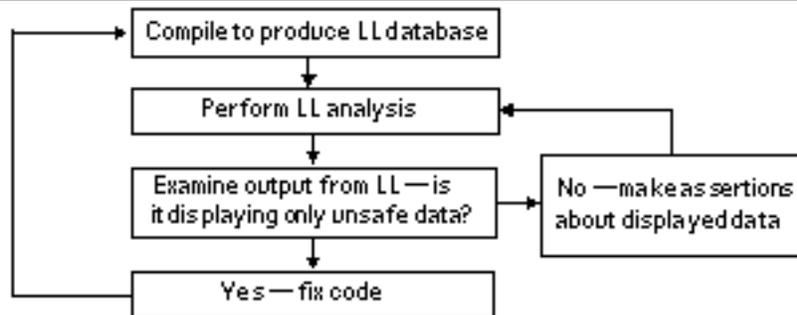


Figure 5-1 LockLint Control Flow

Use LockLint to refine the set of assertions you maintain for the implementation of your system. A rich set of assertions enables LockLint to validate existing and new source code as you work.

## Managing LockLint's Environment

The LockLint interface consists of the `lock_lint` command, which is executed in a shell, and the `lock_lint` subcommands. By default, LockLint uses the shell given by the environment variable `$SHELL`. Alternatively, LockLint can execute any shell by specifying the shell to use on the `lock_lint start` command. This example starts a LockLint session in the Korn shell:

```
% lock_lint start /bin/ksh
```

LockLint creates an environment variable called `LL_CONTEXT`, which is visible in the child shell. If you are using a shell that provides for initialization, you can arrange to have the `lock_lint` command source a `.ll_init` file in your home directory, and then execute a `.ll_init` file in the current directory if it exists. If you use `csh`, you can do this by inserting the following code into your `.cshrc` file:

```
if (?LL_CONTEXT) then
  if ( -x $(HOME)/.ll_init ) source $(HOME)/.ll_init
endif
```

It is better *not* to have your `.cshrc` source the file in your current working directory, since others may want to run LockLint on those same files, and they may not use the same shell you do. Since you are the only one who is going to use your `$(HOME)/.ll_init`, you *should* source that one, so that you can change the prompt and define aliases for use during your LockLint session. The following version of `~/ll_init` does this for `csh`:

```
# Cause analyze subcommand to save state before analysis.
alias analyze `lock_lint save before analyze;\
  lock_lint analyze``
# Change prompt to show we are in lock_lint.
set prompt=`lock_lint~$prompt``
```

Also see “start” on page 129.

When executing subcommands, remember that you can use pipes, redirection, backward quotes ```, and so on to accomplish your aims. For example, the following command asserts that `lock foo` protects all global variables (the formal name for a global variable begins with a colon):

```
% lock_lint assert foo protects `lock_lint vars | grep ^:`
```

In general, the subcommands are set up for easy use with filters such as `grep` and `sed`. This is particularly true for `vars` and `funcs`, which put out a single line of information for each variable or function. Each line contains the attributes (defined and derived) for that variable or function. The following example shows which members of struct `bar` are supposed to be protected by member `lock`:

```
% lock_lint vars -a 'lock_lint members bar' | grep =bar::lock
```

Since you are using a shell interface, a log of user commands can be obtained by using the shell's history function (the history level may need to be made large in the `.ll_init` file).

## Temporary Files

LockLint puts temporary files in `/var/tmp` unless `$TMPDIR` is set.

## Makefile Rules

To modify your makefile to produce `.ll` files, first use the rule for creating a `.o` from a `.c` to write a rule to create a `.ll` from a `.c`. For example, from:

```
# Rule for making .o from .c in ../src.
%.o: ../src/%.c
    $(COMPILE.c) -o $@ $<
```

you might write:

```
# Rule for making .ll from .c in ../src.
%.ll: ../src/%.c
    cc $(CFLAGS) $(CPPFLAGS) $(FOO) $<
```

In the above example, the `-zll` flag would have to be specified in the make macros for compiler options (`CFLAGS` and `CPPFLAGS`).

If you use a suffix rule, you will need to define `.ll` as a suffix. For that reason some prefer to use `%` rules.

If the appropriate `.o` files are contained in a make variable `FOO_OBJS`, you can create `FOO_LLS` with the line:

```
FOO_LLS = ${FOO_OBJS:%.o=%.ll}
```

or, if they are in a subdirectory `ll`:

```
FOO_LLS = ${FOO_OBJS:%.o=ll/%.ll}
```

If you want to keep the `.ll` files in subdirectory `ll/`, you can have the makefile automatically create this file with the label:

```
.INIT:  
@if [ ! -d ll ]; then mkdir ll; fi
```

## Compiling Code

For LockLint to analyze your source code, you must first compile it using the `-Zll` option of the Sun WorkShop ANSI C compiler. The compiler then produces the LockLint database files (`.ll` files), one for each `.c` file compiled. Later you load the `.ll` files into LockLint with the `load` subcommand.

LockLint sometimes needs a simpler view of the code to return meaningful results during analysis. To allow you to provide this simpler view, the `-Zll` option automatically defines the preprocessor symbol `__lock_lint`; further discussions of the likely uses of `__lock_lint` can be found in “Limitations of LockLint” on page 84.

## LockLint Subcommands

The interface to LockLint consists of a set of subcommands that can be specified with the `lock_lint` command:

```
lock_lint [subcommand  
]
```

In this example *subcommand* is one of a set of subcommands used to direct the analysis of the source code for data races and deadlocks. More information about subcommands can be found in Appendix A.

## Starting and Exiting LockLint

The first subcommand of any LockLint session must be `start`, which starts a subshell of your choice with the appropriate LockLint context. Since a LockLint

session is started within a subshell, you exit by exiting that subshell. For example, to exit LockLint when using the C shell, use the command `exit`.

## Setting the Tool State

LockLint's *state* consists of the set of databases loaded and the specified assertions. Iteratively modifying that state and rerunning the analysis can provide optimal information on potential data races and deadlocks. Since the analysis can be done only once for any particular state, the `save`, `restore`, and `refresh` subcommands are provided as a means to reestablish a state, modify that state, and retry the analysis.

## Checking an Application

1. **Annotate your source code and compile it to create `.ll` files.**

See “Source Code Annotations” on page 86.

2. **Load the `.ll` files using the `load` subcommand.**

3. **Make assertions about locks protecting functions and variables using the `assert` subcommand.**

---

**Note** - These specifications may also be conveyed using source code annotations. See “Source Code Annotations” on page 86.

---

4. **Make assertions about the order in which locks should be acquired in order to avoid deadlocks, using the `assert order` subcommand.**

---

**Note** - These specifications may also be conveyed using source code annotations. See “Source Code Annotations” on page 86.

---

5. **Check that LockLint has the right idea about which functions are roots.**

If the `funcs -o` subcommand does not show a root function as root, use the `declare root` subcommand to fix it. If `funcs -o` shows a non-root function as root, it's likely that the function should be listed as a function target using the `declare ... targets` subcommand. See “declare root *func*” on page 114 for a discussion of root functions.

6. **Describe any hierarchical lock relationships (if you have any—they are rare) using the `assert rlock` subcommand.**

---

**Note** - These specifications may also be conveyed using source code annotations. See “Source Code Annotations” on page 86.

---

**7. Tell LockLint to ignore any functions or variables you want to exclude from the analysis using the `ignore` subcommand.**

Be conservative in your use of the `ignore` command. Make sure you should not be using one of the source code annotations instead (for example, `NO_COMPETING_THREADS_NOW`).

**8. Run the analysis using the `analyze` subcommand.**

**9. Deal with the errors.**

This may involve modifying the source using `#ifdef __lock_lint` (see “Limitations of LockLint” on page 84) or adding source code annotations to accomplish steps 3, 4, 6, and 7 (see “Source Code Annotations” on page 86).

Restore LockLint to the state it was in before the analysis and rerun the analysis as necessary.

---

**Note** - It is best to handle the errors in order. Otherwise, problems with locks not being held on entry to a function, or locks being released while not held, can cause lots of misleading messages about variables not being properly protected.

---

**10. Run the analysis using the `analyze -v` subcommand and repeat the above step.**

**11. When the errors from the `analyze` subcommand are gone, check for variables that are not properly protected by any lock.**

Use the command: `lock_lint vars -h | fgrep \*`

Rerun the analysis using appropriate assertions to find out where the variables are being accessed without holding the proper locks.

Remember that you cannot run `analyze` twice for a given state, so it will probably help to save the state of LockLint using the `save` subcommand before running `analyze`. Then restore that state using `refresh` or `restore` before adding more assertions. You may want to set up an alias for `analyze` that automatically does a `save` before analyzing.

# Program Knowledge Management

LockLint acquires its information on the sources to be analyzed with a set of databases produced by the C compiler. The LockLint database for each source file is stored in a separate file. To analyze a set of source files, use the `load` subcommand to load their associated database files. The `files` subcommand can be used to display a list of the source files represented by the loaded database files. Once a file is loaded, LockLint knows about all the functions, global data, and external functions referenced in the associated source files.

## Function Management

As part of the analysis phase, LockLint builds a *call graph* for all the loaded sources. Information about the functions defined is available via the `funcs` subcommand. It is extremely important for a meaningful analysis that LockLint have the correct call graph for the code to be analyzed.

All functions that are not called by any of the loaded files are called *root* functions. You may want to treat certain functions as root functions even though they are called within the loaded modules (for example, the function is an entry point for a library that is also called from within the library). Do this by using the `declare root` subcommand. You may also remove functions from the call graph by issuing the `ignore` subcommand.

LockLint knows about all the references to *function pointers* and most of the assignments made to them. Information about the function pointers in the currently loaded files is available through the `funcptrs` subcommand. Information about the calls made via function pointers is available via the `pointer calls` subcommand. If there are function pointer assignments that LockLint could not discover, they may be specified with the `declare ... targets` subcommand.

By default, LockLint tries to examine all possible execution paths. If the code uses function pointers, it's possible that many of the execution paths are not actually followed in normal operation of the code. This can result in the reporting of deadlocks that do not really occur. To prevent this, use the `disallow` and `reallow` subcommands to inform LockLint of execution paths that never occur. To print out existing constraints, use the `reallows` and `disallows` subcommands.

## Variable Management

The LockLint database also contains information about all global variables accessed in the source code. Information about these variables is available via the `vars` subcommands.

One of LockLint's jobs is to determine if variable accesses are consistently protected. If you are unconcerned about accesses to a particular variable, you can remove it from consideration by using the `ignore` subcommand.

You may also consider using one of the following source code annotations, as appropriate.

```
SCHEME_PROTECTS_DATA
READ_ONLY_DATA
DATA_READABLE_WITHOUT_LOCK
NOW_INVISIBLE_TO_OTHER_THREADS
NOW_VISIBLE_TO_OTHER_THREADS
```

For more information, see “Source Code Annotations” on page 86.

## Lock Management

Source code annotations are an efficient way to refine the assertions you make about the locks in your code. There are three types of assertions: *protection*, *order*, and *side effects*.

Protection assertions state what is protected by a given lock. For example, the following source code annotations can be used to assert how data is protected.

```
MUTEX_PROTECTS_DATA
RWLOCK_PROTECTS_DATA
SCHEME_PROTECTS_DATA
DATA_READABLE_WITHOUT_LOCK
RWLOCK_COVERS_LOCK
```

A variation of the `assert` subcommand is used to assert that a given lock protects some piece of data or a function. Another variation, `assert ... covers`, asserts that a given lock protects another lock; this is used for hierarchical locking schemes.

*Order* assertions specify the order in which the given locks must be acquired. The source code annotation `LOCK_ORDER` or the `assert order` subcommand can be used to specify lock ordering.

Side effect assertions state that a function has the side effect of releasing or acquiring a given lock. Use the following source code annotations:

```
MUTEX_ACQUIRED_AS_SIDE_EFFECT
READ_LOCK_ACQUIRED_AS_SIDE_EFFECT
WRITE_LOCK_ACQUIRED_AS_SIDE_EFFECT
LOCK_RELEASED_AS_SIDE_EFFECT
LOCK_UPGRADED_AS_SIDE_EFFECT
LOCK_DOWNGRADED_AS_SIDE_EFFECT
```

NO\_COMPETING\_THREADS\_AS\_SIDE\_EFFECT

COMPETING\_THREADS\_AS\_SIDE\_EFFECT

You can also use the `assert side effect` subcommand to specify side effects. In some cases you may want to make side effect assertions about an external function and the lock is not visible from the loaded module (for example, it is static to the module of the external function). In such a case, you can “create” a lock by using a form of the `declare` subcommand.

## Analysis of Lock Usage

LockLint’s primary role is to report on lock usage inconsistencies that may lead to data races and deadlocks. The *analysis* of lock usage occurs when you use the `analyze` subcommand. The result is a report on the following problems:

- Functions that produce side effects on locks or violate assertions made about side effects on locks (for example, a function that changes the state of a mutex lock from locked to unlocked). The most common unintentional side effect occurs when a function acquires a lock on entry, and then fails to release it at some return point. That path through the function is said to acquire the lock as a side effect. This type of problem may lead to both data races and deadlocks.
- Functions that have inconsistent side effects on locks (that is, different paths through the function) yield different side effects. This may be a limitation of LockLint (see “Limitations of LockLint” on page 84) and a common cause of errors. LockLint cannot handle such functions. It always reports them as errors and does not correctly interpret them. For example, one of the returns from a function may forget to unlock a lock acquired in the function.
- Violations of assertions about which locks should be held upon entry to a function. This problem may lead to a data race.
- Violations of assertions that a lock should be held when a variable is accessed. This problem may lead to a data race.
- Violations of assertions that specify the order in which locks are to be acquired. This problem may lead to a deadlock.
- Failure to use the same, or asserted, mutex lock for all waits on a particular condition variable.
- Miscellaneous problems related to analysis of the source code in relation to assertions and locks.

## Post-analysis Queries

After analysis, you can use LockLint subcommands for:

- Finding additional locking inconsistencies.

- Forming appropriate `declare`, `assert`, and `ignore` subcommands. These can be specified after you've restored LockLint's state, prior to rerunning the analysis.

One such subcommand is `order`, which you can use to make inquiries about the order in which locks have been acquired. This information is particularly useful in understanding lock ordering problems and making assertions about those orders so that LockLint can more accurately diagnose potential deadlocks.

Another such subcommand is `vars`. The `vars` subcommand reports which locks are consistently held when a variable is read or written (if any). This information can be useful in determining the protection conventions in code where the original conventions were never documented, or the documentation has become outdated.

## Limitations of LockLint

There are limitations to LockLint's powers of analysis. At the root of many of its difficulties is the fact that LockLint doesn't know the values of your variables.

LockLint solves some of these problems by ignoring the likely cause or making simplifying assumptions. You can avoid some other problems by using conditionally compiled code in the application. Towards this end, the compiler always defines the preprocessor macro `__lock_lint` when you compile with the `-zll` option. You can use this macro to make your code less ambiguous.

LockLint has trouble deducing:

- Which functions your function pointers point to. There are some assignments LockLint cannot deduce (see "declare" on page 112). The `declare` subcommand can be used to add new possible assignments to the function pointer.

When LockLint sees a call through a function pointer, it tests that call path for every possible value of that function pointer. If you know or suspect that some calling sequences are never executed, use the `disallow` and `reallow` subcommands to specify which sequences are executed.

- Whether or not you locked a lock in code like this:

```
if (x) pthread_mutex_lock(&lock1);
```

In this case, two execution paths are created, one holding the lock, and one not holding the lock, which will probably cause the generation of a side effect message at the `unlock` call. You may be able to work around this problem by using the `__lock_lint` macro to force LockLint to treat a lock as unconditionally taken. For example:

```
#ifdef __lock_lint
pthread_mutex_lock(&lock1);
#else
if (x) pthread_mutex_lock(&lock1);
#endif
```

LockLint has no problem analyzing code like this:

```
if (x) {
    pthread_mutex_lock(&lock1);
    foo();
    pthread_mutex_unlock(&lock1);
}
```

In this case, there is only one execution path, along which the lock is acquired and released, causing no side effects.

- Whether or not a lock was acquired in code like this:

```
rc = pthread_mutex_trylock(&lock1);
if (rc) ...
```

- Which lock is being locked in code like this:

```
pthread_mutex_t* lockp;
pthread_mutex_lock(lockp);
```

In such cases, the lock call is ignored.

- Which variables and locks are being used in code where elements of a structure are used (see “Lock Inversions” on page 133):

```
struct foo* p;
pthread_mutex_lock(p->lock);
p->bar = 0;
```

- Which element of an array is being accessed. This is treated analogously to the previous case; the index is ignored.
- Anything about `longjmps`.
- When you would exit a loop or break out of a recursion (so it just stops proceeding down a path as soon as it finds itself looping or after one recursion).

Some other LockLint difficulties:

- LockLint only analyzes the use of mutex locks and readers-writer locks. LockLint performs limited consistency checks of mutex locks as used with condition variables. However, semaphores and condition variables are not recognized as locks by LockLint. Even with this analysis, there are limits to what LockLint can make sense of.
- There are situations where LockLint thinks two different variables are the same variable, or that a single variable is two different variables. (See “Lock Inversions” on page 133.)
- It is possible to share automatic variables between threads (via pointers), but LockLint assumes that automatics are unshared, and generally ignores them (the

only situation in which they are of interest to LockLint is when they are function pointers).

- LockLint complains about any functions that are not consistent in their side effects on locks. `#ifdef`'s and assertions must be used to give LockLint a simpler view of functions that may or may not have such a side effect.

During analysis, LockLint may produce messages about a lock operation called `rw_upgrade`. Such a call does not really exist, but LockLint rewrites code like

```
if (rw_tryupgrade(&lock1)) { ... }
```

as

```
if () { rw_tryupgrade(&lock1); ... }
```

such that, wherever `rw_tryupgrade()` occurs, LockLint always assumes it succeeds.

One of the errors LockLint flags is an attempt to acquire a lock that is already held. However, if the lock is unnamed (for example, `foo::lock`), this error is suppressed, since the name refers not to a single lock but to a set of locks. However, if the unnamed lock always refers to the same lock, use the `declare one` subcommand so that LockLint can report this type of potential deadlock.

If you have constructed your own locks out of these locks (for example, recursive mutexes are sometimes built from ordinary mutexes), LockLint will not know about them. Generally you can use `#ifdef` to make it appear to LockLint as though an ordinary mutex is being manipulated. For recursive locks, use an unnamed lock for this deception, since errors won't be generated when it is recursively locked. For example:

```
void get_lock() {
    #ifdef __lock_lint
        struct bogus *p;
        pthread_mutex_lock(p->lock);
    #else
        <the real recursive locking code>
    #endif
}
```

---

## Source Code Annotations

An annotation is some piece of text inserted into your source code. You use annotations to tell LockLint things about your program that it cannot deduce for

itself, either to keep it from excessively flagging problems or to have LockLint test for certain conditions. Annotations also serve to document code, in much the same way that comments do. There are two types of source code annotations: *assertions* and *NOTES*.

Annotations are similar to some of the LockLint subcommands described in Appendix A. In general, it's preferable to use source code annotations over these subcommands, as explained in "Reasons to Use Source Code Annotations" on page 87.

## Reasons to Use Source Code Annotations

There are several reasons to use source code annotations. In many cases, such annotations are preferable to using a script of LockLint subcommands.

- Annotations, being mixed in with the code that they describe, are generally better maintained than a script of LockLint subcommands.
- With annotations, you can make assertions about lock state at any point within a function—wherever you put the assertion is where the check occurs. With subcommands, the finest granularity you can achieve is to check an assertion on entry to a function.
- Functions mentioned in subcommands can change. If someone changes the name of a function from `func1` to `func2`, a subcommand mentioning `func1` fails (or worse, might work but do the wrong thing, if a different function is given the name `func1`).
- Some annotations, such as `NOTE(NO_COMPETING_THREADS_NOW)`, have no subcommand equivalents.
- Annotations provide a good way to document your program. In fact, even if you are not using LockLint often, annotations are worthwhile just for this purpose. For example, a header file declaring a variable can document what lock or convention protects the variable, or a function that acquires a lock and deliberately returns without releasing it can have that behavior clearly declared in an annotation.

## The Annotations Scheme

LockLint shares the source code annotations scheme with several other tools. When you install the Sun WorkShop ANSI C Compiler, you automatically install the file `SUNW_SPRO-cc-ssbd`, which contains the names of all the annotations that LockLint understands. The file is located in `<installation_directory>/SUNWspro/SC5.0/lib/note`.

You may specify a location other than the default by setting the environment variable `NOTEPATH`, as in

```
setenv NOTEPATH other_location:$NOTEPATH
```

:

The default value for `NOTEPATH` is

```
installation_directory</SUNWSPRO/SC5.0/lib/note:/usr/lib/note.>
```

To use source code annotations, include the file `note.h` in your source or header files:

```
#include <note.h>
```

## Using LockLint NOTES

Many of the note-style annotations accept names—of locks or variables—as arguments. Names are specified using the syntax shown in Table 5-5.

TABLE 5-5 Specifying Names With LockLint NOTES

Syntax	Meaning
<i>Var</i>	Named variable
<i>Var.Mbr.Mbr...</i>	Member of a named struct/union variable
<i>Tag</i>	Unnamed struct/union (with this tag)
<i>Tag::Mbr.Mbr...</i>	Member of an unnamed struct/union
<i>Type</i>	Unnamed struct/union (with this typedef)
<i>Type::Mbr.Mbr...</i>	Member of an unnamed struct/union

In C, structure tags and types are kept in separate namespaces, making it possible to have two different `structs` by the same name as far as LockLint is concerned. When LockLint sees `foo::bar`, it first looks for a `struct` with tag `foo`; if it does not find one, it looks for a `type` `foo` and checks that it represents a `struct`.

However, the proper operation of LockLint requires that a given variable or lock be known by exactly one name. Therefore `type` will be used only when no `tag` is provided for the `struct`, and even then only when the `struct` is defined as part of a typedef.

For example, `FOO` would serve as the type name in this example:

```
typedef struct { int a, b; } Foo;
```

These restrictions ensure that there is only one name by which the struct is known.

Name arguments do not accept general expressions. It is not valid, for example, to write:

```
NOTE(MUTEX_PROTECTS_DATA(p->lock, p->a p->b))
```

However, some of the annotations do accept expressions (rather than names); they are clearly marked.

In many cases an annotation accepts a list of names as an argument. Members of a list should be separated by white space. To simplify the specification of lists, a generator mechanism similar to that of many shells is understood by all annotations taking such lists. The notation for this is:

*Prefix*{*A B ...*}*Suffix*

where *Prefix*, *Suffix*, *A*, *B*, ... are nothing at all, or any text containing no white space. The above notation is equivalent to:

*PrefixASuffix PrefixBSuffix ...*

For example, the notation:

```
struct_tag::{a b c d}
```

is equivalent to the far more cumbersome text:

```
struct_tag::a struct_tag::b struct_tag::c struct_tag::d
```

This construct may be nested, as in:

```
foo::{a b.{c d} e}
```

which is equivalent to:

```
foo::a
```

```
foo::b.c
```

```
foo::b.d
```

```
foo::ae
```

Where an annotation refers to a lock or another variable, a declaration or definition for that lock or variable should already have been seen.

If a name for data represents a structure, it refers to all non-lock (mutex or readers-writer) members of the structure. If one of those members is itself a structure, then all of its non-lock members are implied, and so on. However, LockLint understands the abstraction of a condition variable and therefore does not break it down into its constituent members.

## NOTE and \_NOTE

The `NOTE` interface enables you to insert information for LockLint into your source code without affecting the compiled object code. The basic syntax of a note-style annotation is either:

```
NOTE(NoteInfo)
```

or:

```
_NOTE(NoteInfo)
```

The preferred use is `NOTE` rather than `_NOTE`. Header files that are to be used in multiple, unrelated projects, should use `_NOTE` to avoid conflicts. If `NOTE` has already been used, and you do not want to change, you should define some other macro (such as `ANNOTATION`) using `_NOTE`. For example, you might define an include file (say, `annotation.h`) that contains the following:

```
#define ANNOTATION _NOTE
#include <sys/note.h>
```

The *NoteInfo* that gets passed to the `NOTE` interface must syntactically fit one of the following:

*NoteName*

*NoteName*(*Args*)

*NoteName* is simply an identifier indicating the type of annotation. *Args* can be anything, so long as it can be tokenized properly and any parenthesis tokens are matched (so that the closing parenthesis can be found). Each distinct *NoteName* will have its own requirements regarding arguments.

This text uses `NOTE` to mean both `NOTE` and `_NOTE`, unless explicitly stated otherwise.

### Where NOTE May Be Used

`NOTE` may be invoked only at certain well-defined places in source code:

- At the top level; that is, outside of all function definitions, type and `struct` definitions, variable declarations, and other constructs. For example:

```
struct foo { int a, b; mutex_t lock; };
NOTE(MUTEX_PROTECTS_DATA(foo::lock, foo))
bar() {...}
```

- At the top level within a block, among declarations or statements. Here too, the annotation must be outside of all type and `struct` definitions, variable declarations, and other constructs. For example:

```
foo() { ...; NOTE(...) ...; ... }
```

- At the top level within a `struct` or `union` definition, among the declarations. For example:

```
struct foo { int a; NOTE(...) int b; };
```

### *Where NOTE May Not Be Used*

`NOTE()` may be used only in the locations described above. For example, the following are invalid:

```
a = b NOTE(...) + 1;
typedef NOTE(...) struct foo Foo;
for (i=0; NOTE(...) i<10; i++) ...
```

A note-style annotation is not a statement; `NOTE()` may not be used inside an `if/else/for/while` body unless braces are used to make a block. For example, the following causes a syntax error:

```
if (x) NOTE(...)
```

## How Data Is Protected

The following annotations are allowed both outside and inside a function definition. Remember that any name mentioned in an annotation must already have been declared.

```
NOTE(MUTEX_PROTECTS_DATA(Mutex, DataNameList))
NOTE(RWLOCK_PROTECTS_DATA(Rwlock, DataNameList))
NOTE(SCHEME_PROTECTS_DATA("description", DataNameList))
```

The first two annotations tell LockLint that the lock should be held whenever the specified data is accessed.

The third annotation, `SCHEME_PROTECTS_DATA`, describes how data are protected if it does not have a mutex or readers-writer lock. The *description* supplied for the scheme is simply text and is not semantically significant; LockLint responds by ignoring the specified data altogether. You may make *description* anything you like.

Some examples help show how these annotations are used. The first example is very simple, showing a lock that protects two variables:

```
mutex_t lock1;
int a,b;
NOTE(MUTEX_PROTECTS_DATA(lock1, a b))
```

In the next example, a number of different possibilities are shown. Some members of `struct foo` are protected by a static lock, while others are protected by the lock on `foo`. Another member of `foo` is protected by some convention regarding its use.

```

mutex_t lock1;
struct foo {
    mutex_t lock;
    int mbr1, mbr2;
    struct {
        int mbr1, mbr2;
        char* mbr3;
    } inner;
    int mbr4;
};
NOTE(MUTEX_PROTECTS_DATA(lock1, foo::{mbr1 inner.mbr1}))
NOTE(MUTEX_PROTECTS_DATA(foo::lock, foo::{mbr2 inner.mbr2}))
NOTE(SCHEME_PROTECTS_DATA("convention XYZ", inner.mbr3))

```

A datum can only be protected in one way. If multiple annotations about protection (not only these three but also `READ_ONLY_DATA`) are used for a single datum, later annotations silently override earlier annotations. This allows for easy description of a structure in which all but one or two members are protected in the same way. For example, most of the members of `struct BAR` below are protected by the lock on `struct foo`, but one is protected by a global lock.

```

mutex_t lock1;
typedef struct {
    int mbr1, mbr2, mbr3, mbr4;
} BAR;
NOTE(MUTEX_PROTECTS_DATA(foo::lock, BAR))
NOTE(MUTEX_PROTECTS_DATA(lock1, BAR::mbr3))

```

## Read-Only Variables

```
NOTE(READ_ONLY_DATA(DataNameList))
```

This annotation is allowed both outside and inside a function definition. It tells LockLint how data should be protected. In this case, it tells LockLint that the data should only be read, and not written.

---

**Note** - No error is signaled if read-only data is written while it is considered invisible. Data is considered *invisible* when other threads cannot access it; for example, if other threads do not know about it.

---

This annotation is often used with data that is initialized and never changed thereafter. If the initialization is done at runtime before the data is visible to other threads, use annotations to let LockLint know that the data is invisible during that time.

LockLint knows that `const` data is read-only.

## Allowing Unprotected Reads

```
NOTE(DATA_READABLE_WITHOUT_LOCK(DataNameList))
```

This annotation is allowed both outside and inside a function definition. It informs LockLint that the specified data may be read without holding the protecting locks. This is useful with an atomically readable datum that stands alone (as opposed to a set of data whose values are used together), since it is valid to peek at the unprotected data if you do not intend to modify it.

## Hierarchical Lock Relationships

```
NOTE(RWLOCK_COVERS_LOCKS(RwlockName, LockNameList))
```

This annotation is allowed both outside and inside a function definition. It tells LockLint that a hierarchical relationship exists between a readers-writer lock and a set of other locks. Under these rules, holding the cover lock for write access affords a thread access to all data protected by the covered locks. Also, a thread must hold the cover lock for read access whenever holding any of the covered locks.

Using a readers-writer lock to cover another lock in this way is simply a convention; there is no special lock type. However, if LockLint is not told about this coverage relationship, it assumes that the locks are being used according to the usual conventions and generates error messages.

The following example specifies that member `lock` of unnamed `foo` structures covers member `lock` of unnamed `bar` and `zot` structures:

```
NOTE(RWLOCK_COVERS_LOCKS(foo::lock, {bar zot}::lock))
```

## Functions With Locking Side Effects

```
NOTE(MUTEX_ACQUIRED_AS_SIDE_EFFECT(MutexExpr))
```

```
NOTE(READ_LOCK_ACQUIRED_AS_SIDE_EFFECT(RwlockExpr))
```

```
NOTE(WRITE_LOCK_ACQUIRED_AS_SIDE_EFFECT(RwlockExpr))
```

```
NOTE(LOCK_RELEASED_AS_SIDE_EFFECT(LockExpr))
```

```
NOTE(LOCK_UPGRADED_AS_SIDE_EFFECT(RwlockExpr))
```

```
NOTE(LOCK_DOWNGRADED_AS_SIDE_EFFECT(RwlockExpr))
```

```
NOTE(NO_COMPETING_THREADS_AS_SIDE_EFFECT)
```

```
NOTE(COMPETING_THREADS_AS_SIDE_EFFECT)
```

These annotations are allowed only inside a function definition. Each tells LockLint that the function has the specified side effect on the specified lock—that is, that the function deliberately leaves the lock in a different state on exit than it was in when

the function was entered. In the case of the last two of these annotations, the side effect is not about a lock but rather about the state of concurrency.

When stating that a readers-writer lock is acquired as a side effect, you must specify whether the lock was acquired for read or write access.

A lock is said to be *upgraded* if it changes from being acquired for read-only access to being acquired for read/write access. *Downgraded* means a transformation in the opposite direction.

LockLint analyzes each function for its side effects on locks (and concurrency). Ordinarily, LockLint expects that a function will have no such effects; if the code has such effects intentionally, you must inform LockLint of that intent using annotations. If it finds that a function has different side effects from those expressed in the annotations, an error message results.

The annotations described in this section refer generally to the function's characteristics and not to a particular point in the code. Thus, these annotations are probably best written at the top of the function. There is, for example, no difference (other than readability) between this:

```
foo() {
  NOTE(MUTEX_ACQUIRED_AS_SIDE_EFFECT(lock_foo))
  ...
  if (x && y) {
    ...
  }
}
```

and this:

```
foo() {
  ...
  if (x && y) {
    NOTE(MUTEX_ACQUIRED_AS_SIDE_EFFECT(lock_foo))
    ...
  }
}
```

If a function has such a side effect, the effect should be the same on every path through the function. LockLint complains about and refuses to analyze paths through the function that have side effects other than those specified.

## Single-Threaded Code

```
NOTE (COMPETING_THREADS_NOW)

NOTE (NO_COMPETING_THREADS_NOW)
```

These two annotations are allowed only inside a function definition. The first annotation tells LockLint that after this point in the code, other threads exist that might try to access the same data that this thread will access. The second function specifies that this is no longer the case; either no other threads are running or whatever threads are running will not be accessing data that this thread will access. While there are no competing threads, LockLint does not complain if the code accesses data without holding the locks that ordinarily protect that data.

These annotations are useful in functions that initialize data without holding locks before starting up any additional threads. Such functions may access data without holding locks, after waiting for all other threads to exit. So one might see something like this:

```
main() {
  <initialize data structures>
  NOTE(COMPETING_THREADS_NOW)
  <create several threads>
  <wait for all of those threads to exit>
  NOTE(NO_COMPETING_THREADS_NOW)
  <look at data structures and print results>
}
```

---

**Note** - If a NOTE is present in main(), LockLint assumes that when main() starts, no other threads are running. If main() does not include a NOTE, LockLint does not assume that no other threads are running.

---

LockLint does not issue a warning if, during analysis, it encounters a COMPETING\_THREADS\_NOW annotation when it already thinks competing threads are present. The condition simply nests. No warning is issued because the annotation may mean different things in each use (that is the notion of which threads compete may differ from one piece of code to the next). On the other hand, a NO\_COMPETING\_THREADS\_NOW annotation that does not match a prior COMPETING\_THREADS\_NOW (explicit or implicit) causes a warning.

## Unreachable Code

```
NOTE(NOT_REACHED)
```

This annotation is allowed only inside a function definition. It tells LockLint that a particular point in the code cannot be reached, and therefore LockLint should ignore the condition of locks held at that point. This annotation need not be used after every call to exit(), for example, as the lint annotation /\* NOTREACHED \*/ is used. Simply use it in definitions for exit() and the like (primarily in LockLint libraries), and LockLint will determine that code following calls to such functions is not reached. This annotation should seldom appear outside LockLint libraries. An example of its use (in a LockLint library) would be:

```
exit(int code) { NOTE(NOT_REACHED) }
```

## Lock Order

`NOTE(LOCK_ORDER(LockNameList))`

This annotation, which is allowed either outside or inside a function definition, specifies the order in which locks should be acquired. It is similar to the `assert_order` and `order` subcommands. See Appendix A.

To avoid deadlocks, LockLint assumes that whenever multiple locks must be held at once they are always acquired in a well-known order. If LockLint has been informed of such ordering using this annotation, an informative message is produced whenever the order is violated.

This annotation may be used multiple times, and the semantics will be combined appropriately. For example, given the annotations

```
NOTE(LOCK_ORDER(a b c))
```

```
NOTE(LOCK_ORDER(b d))
```

LockLint will deduce the ordering:

```
NOTE(LOCK_ORDER(a d))
```

It is not possible to deduce anything about the order of `c` with respect to `d` in this example.

If a cycle exists in the ordering, an appropriate error message will be generated.

## Variables Invisible to Other Threads

```
NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(DataExpr, ...))
```

```
NOTE(NOW_VISIBLE_TO_OTHER_THREADS(DataExpr, ...))
```

These annotations, which are allowed only within a function definition, tell LockLint whether or not the variables represented by the specified expressions are *visible* to other threads; that is, whether or not other threads could access the variables.

Another common use of these annotations is to inform LockLint that variables it would ordinarily assume are visible are in fact not visible, because no other thread has a pointer to them. This frequently occurs when allocating data off the heap—you can safely initialize the structure without holding a lock, since no other thread can yet see the structure.

```
Foo* p = (Foo*) malloc(sizeof(*p));
NOTE(NOW_INVISIBLE_TO_OTHER_THREADS(*p))
p->a = bar;
p->b = zot;
NOTE(NOW_VISIBLE_TO_OTHER_THREADS(*p))
add_entry(&global_foo_list, p);
```

Calling a function never has the side effect of making variables visible or invisible. Upon return from the function, all changes in visibility caused by the function are reversed.

## Assuming Variables Are Protected

```
NOTE(ASSUMING_PROTECTED(DataExpr, ...))
```

This annotation, which is allowed only within a function definition, tells LockLint that this function assumes that the variables represented by the specified expressions are protected in one of the following ways:

- The appropriate lock is held for each variable
- The variables are invisible to other threads
- There are no competing threads when the call is made

LockLint issues an error if none of these conditions is true.

```
f(Foo* p, Bar* q) {  
    NOTE(ASSUMING_PROTECTED(*p, *q))  
    p->a++;  
    ...  
}
```

## Assertions Recognized by LockLint

LockLint recognizes some assertions as relevant to the state of threads and locks. (For more information, see the *assert* man page.)

Assertions may be made only within a function definition, where a statement is allowed.

---

**Note** - `ASSERT()` is used in kernel and driver code, whereas `assert()` is used in user (application) code. For simplicity's sake, this document uses `assert()` to refer to either one, unless explicitly stated otherwise.

---

## Making Sure All Locks Are Released

```
assert(NO_LOCKS_HELD);
```

LockLint recognizes this assertion to mean that, when this point in the code is reached, no locks should be held by the thread executing this test. Violations are

reported during analysis. A routine that blocks might want to use such an assertion to ensure that no locks are held when a thread blocks or exits.

The assertion also clearly serves as a reminder to someone modifying the code that any locks acquired must be released at that point.

It is really only necessary to use this assertion in leaf-level functions that block. If a function blocks only inasmuch as it calls another function that blocks, the caller need not contain this assertion as long as the callee does. Therefore this assertion probably sees its heaviest use in versions of libraries (for example, `libc`) written specifically for LockLint (like lint libraries).

The file `synch.h` defines `NO_LOCKS_HELD` as 1 if it has not already been otherwise defined, causing the assertion to succeed; that is, the assertion is effectively ignored at runtime. You can override this default runtime meaning by defining `NO_LOCKS_HELD` before you include either `note.h` or `synch.h` (which may be included in either order). For example, if a body of code uses only two locks called `a` and `b`, the following definition would probably suffice:

```
#define NO_LOCKS_HELD (!MUTEX_HELD(&a) && !MUTEX_HELD(&b))
#include <note.h>
#include <synch.h>
```

Doing so does not affect LockLint's testing of the assertion; that is, LockLint still complains if *any* locks are held (not just `a` or `b`).

## Making Sure No Other Threads Are Running

```
assert(NO_COMPETING_THREADS);
```

LockLint recognizes this assertion to mean that, when this point in the code is reached, no other threads should be competing with the one running this code. Violations (based on information provided by certain NOTE-style assertions) are reported during analysis. Any function that accesses variables without holding their protecting locks (operating under the assumption that no other relevant threads are out there touching the same data), should be so marked.

By default, this assertion is ignored at runtime—that is, it always succeeds. No generic runtime meaning for `NO_COMPETING_THREADS` is possible, since the notion of which threads compete involves knowledge of the application. For example, a driver might make such an assertion to say that no other threads are running in this driver for the same device. Because no generic meaning is possible, `synch.h` defines `NO_COMPETING_THREADS` as 1 if it has not already been otherwise defined.

However, you can override the default meaning for `NO_COMPETING_THREADS` by defining it before including either `note.h` or `synch.h` (which may be included in either order). For example, if the program keeps a count of the number of running threads in a variable called `num_threads`, the following definition might suffice:

```
#define NO_COMPETING_THREADS (num_threads == 1)
#include <note.h>
#include <synch.h>
```

Doing so does not affect LockLint's testing of the assertion.

## Asserting Lock State

```
assert(MUTEX_HELD(lock_expr) && ...);
```

This assertion is widely used within the kernel. It performs runtime checking if assertions are enabled. The same capability exists in user code.

This code does roughly the same thing during LockLint analysis as it does when the code is actually run with assertions enabled; that is, it reports an error if the executing thread does not hold the lock as described.

---

**Note** - The thread library performs a weaker test, only checking that *some* thread holds the lock. LockLint performs the stronger test.

---

LockLint recognizes the use of `MUTEX_HELD()`, `RW_READ_HELD()`, `RW_WRITE_HELD()`, and `RW_LOCK_HELD()` macros, and negations thereof. Such macro calls may be combined using the `&&` operators. For example, the following assertion causes LockLint to check that a mutex is not held and that a readers-writer lock is write-held:

```
assert(p && !MUTEX_HELD(&p->mtx) && RW_WRITE_HELD(&p->rwlock));
```

LockLint also recognizes expressions like:

```
MUTEX_HELD(&foo) == 0
```



---

## LockLint Command Reference

---

This appendix is organized as follows:

- “Subcommand Summary” on page 101
- “LockLint Naming Conventions” on page 103
- “LockLint Subcommands” on page 106
- “Lock Inversions” on page 133

---

### Subcommand Summary

Table A-1 contains a summary of LockLint subcommands.

**TABLE A-1** LockLint Subcommands

<b>Subcommand</b>	<b>Effect</b>
analyze	Tests the loaded files for lock inconsistencies; also validates against assertions
assert	Specifies what LockLint should expect to see regarding accesses and modifications to locks and variables
declare	Passes information to LockLint that it cannot deduce
disallow	Excludes the specified calling sequence in the analysis
disallows	Lists the calling sequences that are excluded from the analysis

**TABLE A-1** LockLint Subcommands *(continued)*

<b>Subcommand</b>	<b>Effect</b>
<code>files</code>	Lists the source code files loaded via the <code>load</code> subcommand
<code>funcptrs</code>	Lists information about function pointers
<code>funcs</code>	Lists information about specific functions
<code>help</code>	Provides information about the specified keyword
<code>ignore</code>	Excludes the specified functions and variables from analysis
<code>load</code>	Specifies the <code>.ll</code> files to be loaded
<code>locks</code>	Lists information about locks
<code>members</code>	Lists members of the specified struct
<code>order</code>	Shows information about the order in which locks are acquired
<code>pointer calls</code>	Lists calls made through function pointers
<code>reallow</code>	Allows exceptions to the <code>disallow</code> subcommand
<code>reallows</code>	Lists the calling sequences reallowed through the <code>reallow</code> subcommand
<code>refresh</code>	Restores and then saves the latest saved state again
<code>restore</code>	Restores the latest saved state
<code>save</code>	Saves the current state on a stack
<code>saves</code>	Lists the states saved on the stack through the <code>save</code> subcommand
<code>start</code>	Starts a LockLint session
<code>sym</code>	Lists the fully qualified names of functions and variables associated with the specified name

TABLE A-1 LockLint Subcommands (continued)

Subcommand	Effect
<code>unassert</code>	Removes some assertions specified through the <code>assert</code> subcommand
<code>vars</code>	Lists information about variables

Many LockLint subcommands require you to specify names of locks, variables, pointers, and functions. In C, it is possible for names to be ambiguous. See “LockLint Naming Conventions” on page 103 for details on specifying names to LockLint subcommands.

Table A-2 lists the exit status values of LockLint subcommands.

TABLE A-2 Exit Status Values of LockLint Subcommands

Value	Meaning
0	Normal
1	System error
2	User error, such as incorrect options or undefined name
3	Multiple errors
5	LockLint detected error: violation of an assertion, potential data race or deadlock may have been found, unprotected data references, and so on.
10	Licensing error

## LockLint Naming Conventions

Many LockLint subcommands require you to specify names of locks, variables, pointers, and functions. In C, it is possible for names to be ambiguous; for example, there may be several variables named `foo`, one of them `extern` and others `static`.

The C language does not provide a way of referring to ambiguously named variables that are hidden by the scoping rules. In LockLint, however, a way of referring to such variables is needed. Therefore, every symbol in the code being analyzed is

given a formal name, a name that LockLint uses when referring to the symbol. Table A-3 lists some examples of formal names for a function.

**TABLE A-3** Sample Formal Function Names

<b>Formal Name</b>	<b>Definition</b>
<code>:func</code>	extern function
<code>file:func</code>	static function

Table A-4 lists the formal names for a variable, depending on its use as a lock, a pointer, or an actual variable.

**TABLE A-4** Sample Formal Variable Names

<b>Formal Name</b>	<b>Definition</b>
<code>:var</code>	extern variable
<code>file:var</code>	static variable with file scope
<code>:func/var</code>	Variable defined in an extern function
<code>file:func/ var</code>	Variable defined in a static function
<code>tag::mbr</code>	Member of an unnamed struct
<code>file@line::mbr</code>	Member of an unnamed, untagged struct

In addition, any of these may be followed by an arbitrary number of `.mbr` specifications to denote members of a structure.

Table A-5 contains some examples of the LockLint naming scheme.

TABLE A-5 LockLint Naming Scheme Examples

Example	Meaning
<code>:bar</code>	External variable or function <code>bar</code>
<code>:main/bar</code>	static variable <code>bar</code> that is defined within extern function <code>main</code>
<code>zot.c:foo/ bar.zot</code>	Member <code>zot</code> of static variable <code>bar</code> , which is defined within static function <code>foo</code> in file <code>zot.c</code>
<code>foo::bar.zot.bim</code>	Member <code>bim</code> of member <code>zot</code> of member <code>bar</code> of a struct with tag <code>foo</code> , where no name is associated with that instance of the struct (it was accessed through a pointer)

While LockLint refers to symbols in this way, you are not required to. You may use as little of the name as is required to unambiguously identify it. For example, you could refer to `zot.c:foo/bar` as `foo/bar` as long as there is only one function `foo` defining a variable `bar`. You can even refer to it simply as `bar` as long as there is no other variable by that name.

C allows the programmer to declare a structure without assigning it a tag. When you use a pointer to such a structure, LockLint must make up a tag by which to refer to the structure. It generates a tag of the format `filename@line_number`. For example, if you declare a structure without a tag at line 42 of file `foo.c`, and then refer to member `bar` of an instance of that structure using a pointer, as in:

```
typedef struct { ... } foo;
foo *p;
func1() { p->bar = 0; }
```

LockLint sees that as a reference to `foo.c@42::bar`.

Because members of a union share the same memory location, LockLint treats all members of a union as the same variable. This is accomplished by using a member name of `%` regardless of which member is accessed. Since bit fields typically involve sharing of memory between variables, they are handled similarly: `%` is used in place of the bit field member name.

When you list locks and variables, you are only seeing those locks and variables that are actually used within the code represented by the `.ll` files. No information is available from LockLint on locks, variables, pointers, and functions that are declared but not used. Likewise, no information is available for accesses through pointers to simple types, such as this one:

```
int *ip = &i;
*ip = 0;
```

When simple names (for example, `foo`) are used, there is the possibility of conflict with keywords in the subcommand language. Such conflicts can be resolved by surrounding the word with double quotes, but remember that you are typing commands to a shell, and shells typically consume the outermost layer of quotes. Therefore you have to escape the quotes, as in this example:

```
% lock_lint ignore foo in func \"func\"
```

If two files with the same base name are included in an analysis, and these two files contain `static` variables by the same name, confusion can result. LockLint thinks the two variables are the same.

If you duplicate the definition for a `struct` with no tag, LockLint does not recognize the definitions as the same `struct`. The problem is that LockLint makes up a tag based on the file and line number where the `struct` is defined (such as `x.c@24`), and that tag differs for the two copies of the definition.

If a function contains multiple automatic variables of the same name, LockLint cannot tell them apart. Because LockLint ignores automatic variables except when they are used as function pointers, this does not come up often. In the following code, for example, LockLint uses the name `:foo/fp` for both function pointers:

```
int foo(void (*fp)()) {
    (*fp)();
    {
        void (*fp)() = get_func();
        (*fp)();
        ...
    }
}
```

---

## LockLint Subcommands

Some of these are equivalent to subcommands such as `assert`. Source code annotations are often preferable to subcommands, because they

- Have finer granularity
- Are easy to maintain
- Serve as comments on the code in question

# analyze

analyze has the following syntax:

```
analyze [-hv]
```

Analyzes the loaded files for lock inconsistencies that may lead to data races and deadlocks. This subcommand may produce a great deal of output, so you may want to redirect the output to a file. This subcommand can be run only once for each saved state. (See “save” on page 128).

-h (history) produces detailed information for each phase of the analysis. No additional errors are issued.

-v (verbose) generates additional messages during analysis:

- Writable variable read while no locks held!
- Variable written while no locks held!
- No lock consistently held while accessing variable!

Output from the analyze subcommand can be particularly abundant if:

- The code has not been analyzed before
- The `assert read only` subcommand was not used to identify read-only variables
- No assertions were made about the protection of writable variables

The output messages are likely to reflect situations that are not real problems; therefore, it is often helpful to first analyze the code without the `-v` option, to show only the messages that are likely to represent real problems.

## LockLint analyze Phases

Each problem encountered during analysis is reported on one or more lines, the first of which begins with an asterisk. Where possible, LockLint provides a complete traceback of the calls taken to arrive at the point of the problem. The analysis goes through the following phases:

### 1. Checking for functions with variable side effects on locks

If a `disallow` sequence specifies that a function with locking side effects should not be analyzed, LockLint produces incorrect results. If such `disallow` sequences are found, they are reported and analysis does not proceed.

### 2. Preparing locks to hold order info

LockLint processes the asserted lock order information available to it. If LockLint detects a cycle in the asserted lock order, the cycle is reported as an error.

### 3. Checking for function pointers with no targets

LockLint cannot always deduce assignments to function pointers. During this phase, LockLint reports any function pointer for which it does not think there is at least one target, whether deduced from the source or declared a `func.ptr` target.

#### 4. Removing accesses to ignored variables

To improve performance, LockLint removes references to ignored variables at this point. (This affects the output of the `vars` subcommands.)

#### 5. Preparing functions for analysis

During this phase, LockLint determines what side effects each function has on locks. (A *side effect* is a change in a lock's state that is not reversed before returning.) An error results if

- The side effects do not match what LockLint expects
- The side effects are different depending upon the path taken through the function
- A function with such side effects is recursive

LockLint expects that a function will have no side effects on locks, except where side effects have been added using the `assert side effect` subcommand.

#### 6. Preparing to recognize calling sequences to `allow/disallow`

Here LockLint is processing the various `allow/disallow` subcommands that were issued, if any. No errors or warnings are reported.

#### 7. Checking locking side effects in function pointer targets

Calls through function pointers may target several functions. All functions that are targets of a particular function pointer must have the same side effects on locks (if any). If a function pointer has targets that differ in their side effects, analysis does not proceed.

#### 8. Checking for consistent use of locks with condition variables

Here LockLint checks that all waits on a particular condition variable use the same mutex. Also, if you assert that particular lock to protect that condition variable, LockLint makes sure you use that lock when waiting on the condition variable.

#### 9. Determining locks consistently held when each function is entered

During this phase, LockLint reports violations of assertions that locks should be held upon entry to a function (see `assert` subcommand). Errors such as locking a mutex lock that is already held, or releasing a lock that is not held, are also reported. Locking an anonymous lock, such as `foo::lock`, more than once is not considered an error, unless the `declare one` command has been used to indicate otherwise. (See "Lock Inversions" on page 133 for details on anonymous data.)

#### 10. Determining locks consistently held when each variable is accessed

During this phase, LockLint reports violations of assertions that a lock should be held when a variable is accessed (see the `assert` subcommand). Also, any writes to read-only variables are reported.

Occasionally you may get messages that certain functions were never called. This can occur if a set of functions (none of which are root functions) call each other. If none of the functions is called from outside the set, LockLint reports that the functions were never called at all. The `declare root` subcommand can be used to fix this situation for a subsequent analysis.

Using the `disallow` subcommand to disallow all sequences that reach a function will also cause a message that the function is never called.

Once the analysis is done, you can find still more potential problems in the output of the `vars` and `order` subcommands.

## assert

`assert` has the following syntax:

---

<code>assert side effect</code>	<i>mutex</i>	acquired in	<i>func ...</i>
<code>assert side effect</code>	<i>rwlock</i> [read]	acquired in	<i>func ...</i>
<code>assert side effect</code>	<i>lock</i>	released in	<i>func ...</i>
<code>assert side effect</code>	<i>rwlock</i>	upgraded in	<i>func ...</i>
<code>assert side effect</code>	<i>rwlock</i>	downgraded in	<i>func ...</i>
<code>assert <i>mutex</i>   <i>rwlock</i></code>	protects		<i>var ...</i>
<code>assert <i>mutex</i></code>	protects		<i>func ...</i>
<code>assert <i>rwlock</i></code>	protects	[reads in]	<i>func ...</i>
<code>assert order</code>			<i>lock lock ...</i>
<code>assert read only</code>			<i>var ...</i>
<code>assert <i>rwlock</i></code>	covers		<i>lock ...</i>

---

These subcommands tell LockLint how the programmer expects locks and variables to be accessed and modified in the application being checked. During analysis any violations of such assertions are reported.

---

**Note** - If a variable is asserted more than once, only the last `assert` takes effect.

---

### `assert side effect`

A *side effect* is a change made by a function in the state of a lock, a change that is not reversed before the function returns. If a function contains locking side effects and no assertion is made about the side effects, or the side effects differ from those that are asserted, a warning is issued during the analysis. The analysis then continues as if the unexpected side effect never occurred.

---

**Note** - There is another kind of side effect called an *inversion*. See “Lock Inversions” on page 133, and the `locks` or `funcs` subcommands, for more details.

---

Warnings are also issued if the side effects produced by a function could differ from call to call (for example, conditional side effects). The keywords `acquired in`, `released in`, `upgraded in`, and `downgraded in` describe the type of locking side effect being asserted about the function. The keywords correspond to the side effects available via the threads library interfaces and the DDI and DKI Kernel Functions (see `mutex(3T)`, `rwlock(3T)`, `mutex(9F)` and `rwlock(9F)`).

The side effect assertion for `rwlocks` takes an optional argument `read`; if `read` is present, the side effect is that the function acquires read-level access for that lock. If `read` is not present, the side effect specifies that the function acquires write-level access for that lock.

### `assert mutex|rwlock protects`

Asserting that a mutex lock protects a variable causes an error whenever the variable is accessed without holding the mutex lock. Asserting that a readers-writer lock protects a variable causes an error whenever the variable is read without holding the lock for read access or written without holding the lock for write access. Subsequent assertions as to which lock protects a variable override any previous assertions; that is, only the last lock asserted to protect a variable is used during analysis.

## `assert mutex protects`

Asserting that a mutex lock protects a function causes an error whenever the function is called without holding the lock. For root functions, the analysis is performed as if the root function were called with this assertion being true.

## `assert rwlock protects`

Asserting that a readers-writer lock protects a function causes an error whenever the function is called without holding the lock for write access. Asserting that a readers-writer lock protects reads in a function causes an error whenever the function is called without holding the lock for read access. For root functions, the analysis is performed as if the root function were called with this assertion being true.

---

**Note** - To avoid flooding the output with too many violations of a single `assert... protects` subcommand, a maximum of 20 violations of any given assertion is shown. This limit does not apply to the `assert order` subcommand.

---

## `assert order`

Informs LockLint of the order in which locks should be acquired. That is, LockLint assumes that the program avoids deadlocks by adhering to a well-known lock order. Using this subcommand, you can make LockLint aware of the intended order so that violations of the order can be printed during analysis.

## `assert read only`

States that the given set of variables should never be written by the application; LockLint reports any writes to the variables. Unless a variable is read-only, reading the variable while no locks are held will elicit an error since LockLint assumes that the variable could be written by another thread at the same time.

## `assert rwlock covers`

Informs LockLint of the existence of a hierarchical locking relationship. A readers-writer lock may be used in conjunction with other locks (mutex or readers-writer) in the following way to increase performance in certain situations:

- A certain readers-writer lock, called the *cover*, must be held while any of a set of other covered locks is held. That is, it is illegal (under these conventions) to hold a covered lock while not also holding the cover, with at least read access.

- While holding the cover for write access, you can access any variable protected by one of the covered locks without holding the covered lock. This works because it is impossible for another thread to hold the covered lock (since it would also have to be holding the cover). The time saved by not locking the covered locks can increase performance if there is not excessive contention over the cover.

Using `assert rwlock covers` prevents LockLint from issuing error messages when a thread accesses variables while holding the cover for write access but not the covered lock. It also enables checks to ensure that a covered lock is never held when its cover is not.

## declare

`declare` has the following syntax:

---

<code>declare</code>	<code>mutex</code>	<code><i>mutex</i> ...</code>	
<code>declare</code>	<code>rwlocks</code>	<code><i>rwlock</i> ...</code>	
<code>declare</code>	<code><i>func_ptr</i></code>	<code>targets</code>	<code><i>func</i> ...</code>
<code>declare</code>	<code>nonreturning</code>	<code><i>func</i> ...</code>	
<code>declare</code>	<code>one</code>	<code><i>tag</i> ...</code>	
<code>declare</code>	<code>readable</code>	<code><i>var</i> ...</code>	
<code>declare</code>	<code>root</code>	<code><i>func</i> ...</code>	

---

These subcommands tell LockLint things that it cannot deduce from the source presented to it.

## `declare mutex mutex` `declare rwlocks rwlock`

These subcommands (along with `declare root`, below) are typically used when analyzing libraries without a supporting harness. The subcommands `declare mutex` and `declare rwlocks` create mutex and reader-writer locks of the given names. These symbols can be used in subsequent `assert` subcommands.

## declare *func\_ptr* targets *func*

Adds the specified functions to the list of functions that could be called through the specified function pointer.

LockLint manages to gather a good deal of information about function pointer targets on its own by watching initialization and assignments. For example, for the code

```
struct foo { int (*fp)(); } foo1 = { bar };
```

LockLint does the equivalent of the command

```
% lock_lint declare foo::fp targets bar
```



---

**Caution** - LockLint does not yet do the following (for the above example):

```
% lock_lint declare foo1.fp targets bar
```

However, it does manage to do both for assignments to function pointers. See “Lock Inversions” on page 133.

---

## declare nonreturning *func*

Tells LockLint that the specified functions do not return. LockLint will not give errors about lock state after calls to such functions.

## declare one *tag*

Tells LockLint that only one unnamed instance exists of each structure whose tag is specified. This knowledge makes it possible for LockLint to give an error if a lock in that structure is acquired multiple times without being released. Without this knowledge, LockLint does not complain about multiple acquisitions of anonymous locks (for example, `foo::lock`), since two different instances of the structure could be involved. (See “Lock Inversions” on page 133.)

## declare readable *var*

Tells LockLint that the specified variables may be safely read without holding any lock, thus suppressing the errors that would ordinarily occur for such unprotected reads.

## declare root *func*

Tells LockLint to analyze the given functions as a root function; by default, if a function is called from any other function, LockLint does not attempt to analyze that function as the root of a calling sequence.

A *root function* is a starting point for the analysis; functions that are not called from within the loaded files are naturally roots. This includes, for example, functions that are never called directly but are the initial starting point of a thread (for example, the target function of a `thread_create` call). However, a function that *is* called from within the loaded files might also be called from outside the loaded files, in which case you should use this subcommand to tell LockLint to use the function as a starting point in the analysis.

## disallow

`disallow` has the following syntax:

```
disallow func ...
```

Tells LockLint that the specified calling sequence should not be analyzed. For example, to prevent LockLint from analyzing any calling sequence in which `f()` calls `g()` calls `h()`, use the subcommand

```
% lock_lint disallow f g h
```

Function pointers can make a program appear to follow many calling sequences that do not in practice occur. Bogus locking problems, particularly deadlocks, can appear in such sequences. (See also the description of the subcommand “`reallow`” on page 126.) `disallow` prevents LockLint from following such sequences.

## disallows

`disallows` has the following syntax:

disallows

Lists the calling sequences that are disallowed by the `disallow` subcommand.

## exit

There is no exit subcommand for LockLint. To exit LockLint, use the exit command for the shell you are using.

## files

`files` has the following syntax:

```
files
```

Lists the `.ll` versions of the source code files loaded with the `load` subcommand.

## funcptrs

`funcptrs` has the following syntax:

```
funcptrs [-botu] func_ptr ...  
funcptrs [-blotuz]
```

Lists information about the function pointers used in the loaded files. One line is produced for each function pointer.

**TABLE A-6** `funcptrs` Options

Definition
<i>(bound)</i> This option lists only function pointers to which function targets have been bound, that is it suppresses the display of function pointers for which there are no bound targets.
<i>(long)</i> Equivalent to <code>-ot</code> .
1

TABLE A-6 funcptrs Options (continued)

Definition	
<i>(other)</i> This presents the following information about each function pointer:	
○	
calls=#	Indicates the number of places in the loaded files this function pointer is used to call a function.
=nonreturning	Indicates that a call through this function pointer never returns (none of the functions targeted ever return).
<i>(targets)</i> This option lists the functions currently bound as targets to each function pointer listed, as follows: targets={ <b>func</b> ... }	
<i>(unbound)</i> This lists only those function pointers to which no function targets are bound. That is, suppresses the display of function pointers for which there are bound targets.	
<i>(zero)</i> This lists function pointers for which there are no calls. Without this option information is given only on function pointers through which calls are made.	

You can combine various options to funcptrs:

- This example lists information about the specified function pointers. By default, this variant of the subcommand gives all the details about the function pointers, as if -ot had been specified.

```
funcptrs [-botu] func_ptr ...
```

- This example lists information about all function pointers through which calls are made. If -z is used, even function pointers through which no calls are made are listed.

```
funcptrs [-blotuz]
```

## funcs

funcs has the following syntax:

funcs [-adehou]				func ...
funcs [-adehilou]	[directly]			
funcs [-adehlou]	[directly]	called by		func ...
funcs [-adehlou]	[directly]	calling		func ...

funcs [-adehlou]	[directly]	reading	var ....
funcs [-adehlou]	[directly]	writing	var ...
funcs [-adehlou]	[directly]	accessing	var ...
funcs [-adehlou]	[directly]	affecting	lock ...
funcs [-adehlou]	[directly]	inverting	lock ...

---

funcs lists information about the functions defined and called in the loaded files. Exactly one line is printed for each function.

TABLE A-7 funcs Options

---

**Definition**

---

(*asserts*) This option shows information about which locks are supposed to be held on entry to each function, as set by the `assert` subcommand. When such assertions have been made, they show as:

```
asserts={ lock ... }
```

```
read_asserts={ lock ... }
```

An asterisk appears before the name of any lock that was not consistently held upon entry (after analysis).

(*effects*) This option shows information about the side effects each function has on locks (for example, “acquires mutex lock foo”). If a function has such side effects, they are shown as:

```
side_effects={ effect [, effect] ... }
```

Using this option prior to analysis shows side effects asserted by an `assert side effect` subcommand. After analysis, information on side effects discovered during the analysis is also shown.

(*defined*) This option shows only those functions that are *defined* in the loaded files. That is, that it suppresses the display of undefined functions.

---

TABLE A-7 funcs Options (continued)

---

**Definition**

---

*(held)* This option shows information about which locks were consistently held when the function was called (after analysis). Locks consistently held for read (or write) on entry show as:

held={ lock ... }+{ lock ... }

read\_held={ lock ... }+{ lock ... }

The first list in each set is the list of locks *consistently* held when the function was called; the second is a list of *inconsistently* held locks—locks that were sometimes held when the function was called, but not every time.

*(ignored)* This option lists ignored functions.

i

*(long)* Equivalent to -aeoh.

l

---

TABLE A-7 func Options (continued)

Definition	
<i>(other)</i> This option causes LockLint to present, where applicable, the following information about each function	
=ignored	Indicates that LockLint has been told to ignore the function using the ignore subcommand.
=nonreturning	Indicates that a call through this function never returns (none of the functions targeted ever return).
=rooted	Indicates that the function was made a root using the declare root subcommand.
=root	Indicates that the function is naturally a root (is not called by any function).
=recursive	Indicates that the function makes a call to itself.
=unanalyzed	Indicates that the function was never called during analysis (and is therefore unanalyzed). This differs from =root in that this can happen when foo calls bar and bar calls foo, and no other function calls either foo or bar, and neither have been rooted (see =rooted). So, because foo and bar are not roots, and they can never be reached from any root function, they have not been analyzed.
calls=#	Indicates the number of places in the source code, as represented by the loaded files, where this function is called. These calls may not actually be analyzed; for example, a disallow subcommand may prevent a call from ever really taking place.
<i>(undefined)</i> This option shows only those functions that are <i>undefined</i> in the loaded files.	
u	

func [-adehou] *func* ...

Lists information about individual functions. By default, this variant of the subcommand gives all the details about the functions, as if -aeho had been specified.

func [-adehilou]

Lists information about all functions that are not ignored. If -i is used, even ignored functions are listed.

`funcs [-adehlou] [directly] called by func ...`

Lists only those functions that may be called as a result of calling the specified functions. If `directly` is used, only those functions called by the specified functions are listed. If `directly` is not used, any functions *those* functions called are also listed, and so on.

`funcs [-adehlou] [directly] calling func ...`

Lists only those functions that, when called, may result in one or more of the specified functions being called. See notes below on `directly`.

`funcs [-adehlou] [directly] reading var ...`

Lists only those functions that, when called, may result in one or more of the specified variables being read. See notes below on `directly`.

`funcs [-adehlou] [directly] writing var ...`

Lists only those functions that, when called, may result in one or more of the specified variables being written. See notes below on `directly`.

`funcs [-adehlou] [directly] accessing var ...`

Lists only those functions that, when called, may result in one or more of the specified variables being accessed (read or written). See notes below on `directly`.

`funcs [-adehlou] [directly] affecting lock ...`

Lists only those functions that, when called, may result in one or more of the specified locks being affected (acquired, released, upgraded, or downgraded). See notes below on `directly`.

`funcs [-adehlou] [directly] inverting lock ...`

Lists only those functions that invert one or more of the specified locks. (See “Lock Inversions” on page 133.) If `directly` is used, only those functions that themselves invert one or more of the locks (actually release them) are listed. If `directly` is not used, any function that is called with a lock already held, and then calls another function that inverts the lock, is also listed, and so on.

For example, in the following code, `f3()` directly inverts lock `m`, and `f2()` indirectly inverts it:

```
f1() { pthread_mutex_unlock(&m); f2(); pthread_mutex_lock(&m); }
f2() { f3(); }
f3() { pthread_mutex_unlock(&m); pthread_mutex_lock(&m); }
```

## About `directly`

Except where stated otherwise, variants that allow the keyword `directly` only list the functions that *themselves* fit the description. If `directly` is not used, all the functions that call those functions are listed, and any functions that call *those* functions, and so on.

## `help`

`help` has the following syntax:

```
help [keyword]
```

Without a keyword, `help` displays the subcommand set.

With a keyword, `help` gives helpful information relating to the specified keyword. The keyword may be the first word of any LockLint subcommand. There are also a few other keywords for which help is available:

```
condvars
locking
example
makefile
ifdef
names
inversions
overview
limitations
shell
```

If environment variable `PAGER` is set, that program is used as the pager for `help`. If `PAGER` is not set, `more` is used.

## `ignore`

`ignore` has the following syntax:

```
ignore
func|var ... [ in func ... ]
```

Tells LockLint to exclude certain functions and variables from the analysis. This exclusion may be limited to specific functions using the `in func ...` clause; otherwise the exclusion applies to all functions.

The commands

```
% lock_lint funcs -io | grep =ignored
% lock_lint vars -io | grep =ignored
```

show which functions and variables are ignored.

## load

load has the following syntax:

```
load file ...
```

Loads the specified `.ll` files. The extension may be omitted, but if an extension is specified, it must be `.ll`. Absolute and relative paths are allowed. You are talking to a shell, so the following are perfectly legal (depending upon your shell's capabilities):

```
% lock_lint load *.ll
% lock_lint load ../foo/abcdef{1,2}
% lock_lint load `find . -name \*.ll -print`
```

The text for load is changed extensively. To set the new text, type:

```
% lock_lint help load
```

## locks

locks has the following syntax:

```
locks [-col] lock ...
locks [-col]
locks [-col] [directly] affected by func ...
locks [-col] [directly] inverted by func ...
```

Lists information about the locks of the loaded files. Only those variables that are actually *used* in lock manipulation routines are shown; locks that are simply declared but never manipulated are not shown.

TABLE A-8 locks Options

---

Definition
<p>(<i>cover</i>) This option shows information about lock hierarchies. Such relationships are described using the <code>assert rwlock covers</code> subcommand. (When locks are arranged in such a hierarchy, the covering lock must be held, at least for read access, whenever any of the covered locks is held. While holding the covering lock for write access, it is unnecessary to acquire any of the covered locks.) If a lock covers other locks, those locks show as <code>covered={ lock ... }</code></p> <p>If a lock is covered by another lock, the covering lock shows as <code>cover=lock</code></p> <p>(<i>long</i>) Equivalent to <code>-co</code>.</p> <p>1</p> <p>(<i>other</i>) Causes the type of the lock to be shown as (<i>type</i>) where <i>type</i> is <i>mutex</i>, <i>rwlock</i>, or <i>ambiguous type</i> [used as a mutex in some places and as a rwlock (readers-writer) in other places].</p>

---

`locks [-co] lock ...`

Lists information about individual locks. By default, this variant of the subcommand gives all the details about the locks, as if `-co` had been specified.

`locks [-col]`

Lists information about all locks.

`locks [-col] [directly] affected by func ...`

Lists only those locks that may be affected (acquired, released, upgraded, or downgraded) as a result of calling the specified functions. If the keyword `directly` is used, only functions that use the threads library routines `directly` to affect a lock (acquire, release, upgrade, or downgrade) are listed. If the keyword `directly` is not

used, any function that calls a function that affects a lock will be listed, and any function calling that function are listed, and so on.

locks [-col] [directly] inverted by *func* ...

Lists only those locks that may be inverted by calling one of the specified functions. (See “Lock Inversions” on page 133.)

If the keyword `directly` is used, only those locks that are directly inverted by the specified functions (that is, the functions that actually release and reacquire locks using a threads library routine) are listed. If the keyword `directly` is not used, a lock that is held by one of the specified functions and inverted by some function called from it (and so on) is also listed. For example, in the following code `f1` directly inverts `m1`, and indirectly inverts `m2`.

```
f1() { pthread_mutex_unlock(&m1); f2(); pthread_mutex_lock(&m1); }
f2() { f3(); }
f3() { pthread_mutex_unlock(&m2); pthread_mutex_lock(&m2); }
```

## members

`members` has the following syntax:

```
members struct_tag
```

Lists the members of the `struct` with the specified tag, one per line. For structures that were not assigned a tag, the notation *file@line* is used (for example, `x.c@29`), where the file and line number are the source location of the `struct` declaration.

`members` is particularly useful to use as input to other LockLint subcommands. For example, when trying to assert that a lock protects all the members of a `struct`, the following command suffices:

```
% lock_lint assert foo::lock protects `lock_lint members foo`
```

---

**Note** - The `members` subcommand does not list any fields of the `struct` that are defined to be of type `mutex_t`, `rwlock_t`, `krwlock_t`, or `kmutex_t`.

---

## order

`order` has the following syntax:

```
order [lock [lock]]
order summary
```

The `order` subcommand lists information about the order in which locks are acquired by the code being analyzed. It may be run only after the `analyze` subcommand.

```
order [lock [lock]]
```

Shows the details about lock pairs. For example, the command

```
% lock_lint order foo bar
```

shows whether an attempt was made to acquire lock `bar` while holding lock `foo`. The output looks something like the following:

```
:foo :bar seen (first never write-held), valid
```

First the output tells whether such an attempt actually occurred (*seen* or *unseen*). If the attempt occurred, but never with one or both of the locks write-held, a parenthetical message to that effect appears, as shown. In this case, `foo` was never write-held while acquiring `bar`.

If an assertion was made about the lock order, the output shows whether the specified order is *valid* or *invalid* according to the assertion. If there was no assertion about the order of `foo` and `bar`, or if both orders were asserted (presumably because the user wanted to see all places where one of the locks was held while acquiring the other), the output indicates neither valid nor invalid.

```
order summary
```

Shows in a concise format the order in which locks are acquired. For example, the subcommand might show

```
:f :e :d :g :a
:f :c :g :a
```

In this example, there are two orders because there is not enough information to allow locks `e` and `d` to be ordered with respect to lock `c`.

Some cycles are shown, while others are not. For example,

```
:a :b :c :b
```

is shown, but

```
:a :b :c :a
```

(where no other lock is ever held while trying to acquire one of these) is not. Deadlock information from the analysis is still reported.

## pointer calls

`pointer calls` has the following syntax:

```
pointer calls
```

Lists calls made through function pointers in the loaded files. Each call is shown as:

```
function [location of call] calls through funcptr func_ptr
```

For example,

```
foo.c:func1 [foo.c,84] calls through funcptr bar::read
```

means that at line 84 of `foo.c`, in `func1` of `foo.c`, the function pointer `bar::read` (member `read` of a pointer to struct of type `bar`) is used to call a function.

## reallow

`reallow` has the following syntax:

```
reallow func ...
```

Allows you to make exceptions to `disallow` subcommands. For example, to prevent LockLint from analyzing any calling sequence in which `f()` calls `g()` calls `h()`, except when `f()` is called by `e()` which was called by `d()`, use the commands

```
% lock_lint disallow f g h  
% lock_lint reallow d e f g h
```

In some cases you may want to state that a function should only be called from a particular function, as in this example:

```
% lock_lint disallow f
% lock_lint reallow e f
```

---

**Note** - A `reallow` subcommand only suppresses the effect of a `disallow` subcommand if the sequences end the same. For example, after the following commands, the sequence `d e f g h` would still be disallowed:

```
% lock_lint disallow e f g h
% lock_lint reallow d e f g
```

---

## reallows

`reallows` has the following syntax:

```
reallows
```

Lists the calling sequences that are reallowed, as specified using the `reallow` subcommand.

## refresh

`refresh` has the following syntax:

```
refresh
```

Pops the saved state stack, restoring LockLint to the state of the top of the saved-state stack, prints the description, if any, associated with that state, and saves the state again. Equivalent to `restore` followed by `save`.

## restore

`restore` has the following syntax:

```
restore
```

Pops the saved state stack, restoring LockLint to the state of the top of the saved-state stack, and prints the description, if any, associated with that state.

The saved state stack is a LIFO (Last-In-First-Out) stack. Once a saved state is restored (popped) from the stack, that state is no longer on the saved-state stack. If the state needs to be saved and restored repeatedly, simply save the state again immediately after restoring it, or use the `refresh` subcommand.

## save

`save` has the following syntax:

```
save description
```

Saves the current state of the tool on a stack. The user-specified *description* is attached to the state. Saved states form a LIFO (Last-In-First-Out) stack, so that the last state saved is the first one restored.

This subcommand is commonly used to save the state of the tool before running the `analyze` subcommand, which can be run only once on a given state. For example, you can do the following:

```
#: lock_lint load *.ll
#: lock_lint save Before Analysis
#: lock_lint analyze
  <output from analyze>
#: lock_lint vars -h | grep \*
  <apparent members of struct foo are not consistently protected>
#: lock_lint refresh Before Analysis
#: lock_lint assert lock1 protects `lock_lint members foo`
#: lock_lint analyze
  <output now contains info about where the assertion is violated>
```

## saves

`saves` has the following syntax:

```
saves
```

Lists the descriptions of the states saved on the saved stack via the `save` subcommand. The descriptions are shown from top to bottom, with the first

description being the most recently saved state that has not been restored, and the last description being the oldest state saved that has not been restored.

## start

`start` has the following syntax:

```
start [cmd]
```

Starts a LockLint session. A LockLint session must be started prior to using any other LockLint subcommand. By default, `start` establishes LockLint's context and starts a subshell for the user, as specified via `$SHELL`, within that context. The only piece of the LockLint context exported to the shell is the environment variable `LL_CONTEXT`, which contains the path to the temporary directory of files used to maintain a LockLint session.

`cmd` specifies a command and its path and options. By default, if `cmd` is not specified, the value of `$SHELL` is used.

---

**Note** - To exit a LockLint session use the `exit` command of the shell you are using.

---

See "Limitations of LockLint" on page 84, for more on setting up the LockLint environment for a `start` subcommand.

## Start Examples

The following examples show variations of the `start` subcommand.

### Using the default

```
% lock_lint start
```

LockLint's context is established and `LL_CONTEXT` is set. Then the program identified by `$SHELL` is executed. Normally, this is your default shell. LockLint subcommands can now be entered. Upon exiting the shell, the LockLint context is removed.

### Using a pre-written script

```
% lock_lint start foo
```

The LockLint context is established and `LL_CONTEXT` is set. Then, the command `/bin/csh -c foo` is executed. This results in executing the C shell command file

`foo`, which contains LockLint commands. Upon completing the execution of the commands in `foo` by `/bin/csh`, the LockLint context is removed.

If you use a shell script to start LockLint, insert `#!` in the first line of the script to define the name of the interpreter that processes that script. For example, to specify the C-shell the first line of the script is:

```
#! /bin/csh
```

## Starting up with a specific shell

In this case, the user starts LockLint with the Korn shell:

```
% lock_lint start /bin/ksh
```

After establishing the LockLint context and setting `LL_CONTEXT`, the command `/bin/ksh` is executed. This results in the user interacting with an interactive Korn shell. Upon exiting the Korn shell, the LockLint context is removed.

## sym

`sym` has the following syntax:

```
sym name ...
```

Lists the fully qualified names of various things the specified names could refer to within the loaded files. For example, `foo` might refer both to variable `x.c:func1/foo` and to function `y.c:foo`, depending on context.

## unassert

`unassert` has the following syntax:

```
unassert vars var ...
```

Undoes any assertion about locks protecting the specified variables. There is no way to remove an assertion about a lock protecting a *function*.

## vars

vars has the following syntax:

```
vars [-aho] var ...
vars [-ahilo]
vars [-ahlo] protected by lock
vars [-ahlo] [directly] read by func ...
vars [-ahlo] [directly] written by func ...
vars [-ahlo] [directly] accessed by func ...
```

Lists information about the variables of the loaded files. Only those variables that are actually *used* are shown; variables that are simply declared in the program but never accessed are not shown.

TABLE A-9 vars Options

Option
<p><i>(assert)</i> Shows information about which lock is supposed to protect each variable, as specified by the <code>assert <i>mutex</i> <i>rwlock</i> protects</code> subcommand. The information is shown as follows <code>assert=<i>lock</i></code></p> <p>If the assertion is violated, then after analysis this will be preceded by an asterisk, such as <code>*assert=&lt;<i>lock</i>&gt;</code>.</p>
<p><i>(held)</i> Shows information about which locks were consistently held when the variable was accessed. This information is shown after the <code>analyze</code> subcommand has been run. If the variable was never accessed, this information is not shown. When it is shown, it looks like this:</p> <pre>held={ &lt;<i>lock</i>&gt; ... }</pre> <p>If no locks were consistently held and the variable was written, this is preceded by an asterisk, such as <code>*held={ }</code>. Unlike <code>funcs</code>, the <code>vars</code> subcommand lists a lock as protecting a variable even if the lock was not actually held, but was simply covered by another lock. (See <code>assert <i>rwlock</i> covers</code> in the description of “<code>assert</code>” on page 109.)</p>
<p><i>(ignored)</i> causes even <i>ignored</i> variables to be listed.</p> <p>i</p>
<p><i>(long)</i> Equivalent to <code>-aho</code>.</p> <p>l</p>

TABLE A-9 vars Options (continued)

Definition	
	( <i>other</i> ) Where applicable, shows information about each variable
	o
=cond_var	Indicates that this variable is used as a condition variable.
=ignored	Indicates that LockLint has been told to ignore the variable explicitly via an ignore subcommand.
=read-only	Means that LockLint has been told (by <code>assert read only</code> ) that the variable is read-only, and will complain if it is written. If it is written, then after analysis this will be followed by an asterisk, such as <code>=read-only*</code> for example.
=readable	Indicates that LockLint has been told by a <code>declare readable</code> subcommand that the variable may be safely read without holding a lock.
=unwritten	May appear after analysis, meaning that while the variable was not declared read-only, it was never written.

vars [-aho] *var* ...

Lists information about individual variables. By default, this variant of the subcommand gives all the details about the variables, as if `-aho` had been specified.

vars [-ahilo]

Lists information about all variables that are not ignored. If `-i` is used, even ignored variables are listed.

vars [-ahlo] protected by *lock*

Lists only those variables that are protected by the specified lock. This subcommand may be run only after the `analyze` subcommand has been run.

vars [-ahlo] [directly] read by *func* ...

Lists only those variables that may be read as a result of calling the specified functions. See notes below on `directly`.

vars [-ahlo] [directly] written by *func* ...

Lists only those variables that may be written as a result of calling the specified functions. See notes below on *directly*.

vars [-ahlo] [directly] accessed by *func* ...

Lists only those variables that may be accessed (read or written) as a result of calling the specified functions.

## About *directly*

Those variants that list the variables accessed by a list of functions can be told to list only those variables that are *directly* accessed by the specified functions. Otherwise the variables accessed by those functions, the functions they call, the functions *those* functions call, and so on, are listed.

---

# Lock Inversions

A function is said to *invert* a lock if the lock is already held when the function is called, and the function releases the lock, such as:

```
foo() {
  pthread_mutex_unlock(&mtx);
  ...
  pthread_mutex_lock(&mtx);
}
```

Lock inversions are a potential source of insidious race conditions, since observations made under the protection of a lock may be invalidated by the inversion. In the following example, if `foo()` inverts `mtx`, then upon its return `zort_list` may be `NULL` (another thread may have emptied the list while the lock was dropped):

```
ZORT* zort_list;
/* VARIABLES PROTECTED BY mtx: zort_list */

void f() {
  pthread_mutex_lock(&mtx);
  if (zort_list == NULL) { /* trying to be careful here */
    pthread_mutex_unlock(&mtx);
    return;
  }
  foo();
  zort_list->count++; /* but zort_list may be NULL here!! */
  pthread_mutex_unlock(&mtx);
}
```

```
}
```

Lock inversions may be found using the commands:

```
% lock_lint funcs [directly] inverting lock ...
% lock_lint locks [directly] inverted by func ...
```

An interesting question to ask is “Which functions acquire locks that then get inverted by calls they make?” That is, which functions are in danger of having stale data? The following (Bourne shell) code can answer this question:

```
$ LOCKS=`lock_lint locks`
$ lock_lint funcs calling `lock_lint funcs inverting $LOCKS`
```

The following gives similar output, separated by lock:

```
for lock in `lock_lint locks`
do
  echo `functions endangered by inversions of lock $lock`
  lock_lint funcs calling `lock_lint funcs inverting $lock`
done
```

# Index

---