Oracle Solaris Studio 12.3: Discover and Uncover User's Guide

Part No: E22000
December 2011

ORACLE

Copyright © 2010, 2011, Oracle and/or its affiliates. All rights reserved.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, the following notice is applicable:

U.S. GOVERNMENT RIGHTS. Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Xeon are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Opteron, the AMD logo, and the AMD Opteron logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information on content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services.

Ce logiciel et la documentation qui l'accompagne sont protégés par les lois sur la propriété intellectuelle. Ils sont concédés sous licence et soumis à des restrictions d'utilisation et de divulgation. Sauf disposition de votre contrat de licence ou de la loi, vous ne pouvez pas copier, reproduire, traduire, diffuser, modifier, breveter, transmettre, distribuer, exposer, exécuter, publier ou afficher le logiciel, même partiellement, sous quelque forme et par quelque procédé que ce soit. Par ailleurs, il est interdit de procéder à toute ingénierie inverse du logiciel, de le désassembler ou de le décompiler, excepté à des fins d'interopérabilité avec des logiciels tiers ou tel que prescrit par la loi.

Les informations fournies dans ce document sont susceptibles de modification sans préavis. Par ailleurs, Oracle Corporation ne garantit pas qu'elles soient exemptes d'erreurs et vous invite, le cas échéant, à lui en faire part par écrit.

Si ce logiciel, ou la documentation qui l'accompagne, est concédé sous licence au Gouvernement des Etats-Unis, ou à toute entité qui délivre la licence de ce logiciel ou l'utilise pour le compte du Gouvernement des Etats-Unis, la notice suivante s'applique:

U.S. GOVERNMENT RIGHTS. Programs, software, databases, and related documentation and technical data delivered to U.S. Government customers are "commercial computer software" or "commercial technical data" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, duplication, disclosure, modification, and adaptation shall be subject to the restrictions and license terms set forth in the applicable Government contract, and, to the extent applicable by the terms of the Government contract, the additional rights set forth in FAR 52.227-19, Commercial Computer Software License (December 2007). Oracle America, Inc., 500 Oracle Parkway, Redwood City, CA 94065.

Ce logiciel ou matériel a été développé pour un usage général dans le cadre d'applications de gestion des informations. Ce logiciel ou matériel n'est pas conçu ni n'est destiné à être utilisé dans des applications à risque, notamment dans des applications pouvant causer des dommages corporels. Si vous utilisez ce logiciel ou matériel dans le cadre d'applications dangereuses, il est de votre responsabilité de prendre toutes les mesures de secours, de sauvegarde, de redondance et autres mesures nécessaires à son utilisation dans des conditions optimales de sécurité. Oracle Corporation et ses affiliés déclinent toute responsabilité quant aux dommages causés par l'utilisation de ce logiciel ou matériel pour ce type d'applications.

Oracle et Java sont des marques déposées d'Oracle Corporation et/ou de ses affiliés. Tout autre nom mentionné peut correspondre à des marques appartenant à d'autres propriétaires qu'Oracle.

Intel et Intel Xeon sont des marques ou des marques déposées d'Intel Corporation. Toutes les marques SPARC sont utilisées sous licence et sont des marques ou des marques déposées de SPARC International, Inc. AMD, Opteron, le logo AMD et le logo AMD Opteron sont des marques ou des marques déposées d'Advanced Micro Devices. UNIX est une marque déposée d'The Open Group.

Ce logiciel ou matériel et la documentation qui l'accompagne peuvent fournir des informations ou des liens donnant accès à des contenus, des produits et des services émanant de tiers. Oracle Corporation et ses affiliés déclinent toute responsabilité ou garantie expresse quant aux contenus, produits ou services émanant de tiers. En aucun cas, Oracle Corporation et ses affiliés ne sauraient être tenus pour responsables des pertes subies, des coûts occasionnés ou des dommages causés par l'accès à des contenus, produits ou services tiers, ou à leur utilisation.

120118@25097

Contents

Preface

The *Oracle Solaris Studio 2.3 Discover and Uncover User's Guide* gives instructions on how to use the Memory Error Discovery Tool (Discover) to find memory-related errors in binaries, and the Code Coverage Tool (Uncover) to measure code coverage of applications.

Supported Platforms

This Oracle Solaris Studio release supports platforms that use the SPARC family of processor architectures running the Oracle Solaris operating system, as well as platforms that use the x86 family of processor architectures running Oracle Solaris or specific Linux systems.

This document uses the following terms to cite differences between x86 platforms:

- "x86" refers to the larger family of 64-bit and 32-bit x86 compatible products.
- "x64" points out specific 64-bit x86 compatible CPUs.
- "32-bit x86" points out specific 32-bit information about x86 based systems.

Information specific to Linux systems refers only to supported Linux x86 platforms, while information specific to Oracle Solaris systems refers only to supported Oracle Solaris platforms on SPARC and x86 systems.

For a complete list of supported hardware platforms and operating system releases, see the Oracle Solaris Studio 12.3 Release Notes.

Oracle Solaris Studio Documentation

You can find complete documentation for Oracle Solaris Studio software as follows:

- Product documentation is located at the Oracle Solaris Studio documentation web site, including release notes, reference manuals, user guides, and tutorials.
- Online help for the Code Analyzer, the Performance Analyzer, the Thread Analyzer, dbxtool, DLight, and the IDE is available through the Help menu, as well as through the F1 key and Help buttons on many windows and dialog boxes, in these tools.
- Man pages for command-line tools describe a tool's command options.

Resources for Developers

Visit the Oracle Technical Network web site to find these resources for developers using Oracle Solaris Studio:

- Articles on programming techniques and best practices
- Links to complete documentation for recent releases of the software
- Information on support levels
- User discussion forums.

Access to Oracle Support

Oracle customers have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Typographic Conventions

The following table describes the typographic conventions that are used in this book.

TABLE P–1 Typographic Conventions

Typeface	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories, and onscreen computer output	Edit your `.login` file. Use `ls -a` to list all files. `machine_name% you have mail.`
`AaBbCc123`	What you type, contrasted with onscreen computer output	`machine_name%` **`su`** `Password:`
aabbcc123	Placeholder: replace with a real name or value	The command to remove a file is `rm` *filename*.
AaBbCc123	Book titles, new terms, and terms to be emphasized	Read Chapter 6 in the *User's Guide*. A *cache* is a copy that is stored locally. Do *not* save the file. **Note:** Some emphasized items appear bold online.

Shell Prompts in Command Examples

The following table shows the default UNIX system prompt and superuser prompt for shells that are included in the Oracle Solaris OS. Note that the default system prompt that is displayed in command examples varies, depending on the Oracle Solaris release.

TABLE P–2 Shell Prompts

Shell	Prompt
Bash shell, Korn shell, and Bourne shell	`$`
Bash shell, Korn shell, and Bourne shell for superuser	`#`
C shell	`machine_name%`
C shell for superuser	`machine_name#`

◆ ◆ ◆ CHAPTER 1

Introduction

The *Oracle Solaris Studio 12.3 Discover and Uncover User's Guide* gives instructions on how to use the following tools:

Memory Error Discovery Tool (Discover)

The Memory Error Discovery Tool (Discover) software is an advanced development tool for detecting memory access errors. Discover works on binaries compiled with the Sun Studio 12 Update 1, Oracle Solaris Studio 12.2, or Oracle Solaris Studio 12.3 compilers; or the GCC for Sun Systems compilers starting with version 4.2.0. It works on systems running the Solaris 10 10/08 operating system or a later Solaris 10 update, or Oracle Solaris 11.

Memory-related errors in programs are notoriously difficult to find. Discover allows you to find such errors easily by pointing out the exact place where the problem exists in the source code. For example, if your program allocates an array and does not initialize it, then tries to read from one of the array locations, the program will probably behave erratically. Discover can catch this problem when you run the program in the normal way.

Other errors detected by Discover include:

- Reading from and writing to unallocated memory
- Accessing memory beyond allocated array bounds
- Incorrect use of freed memory
- Freeing the wrong memory blocks
- Memory leaks

Since Discover catches and reports memory access errors dynamically during program execution, if a portion of user code is not executed at run time, errors in that portion are not reported.

Discover is simple to use. Any binary (even a fully optimized binary) that has been prepared by the compiler can be instrumented with a single command, then run in the normal way. During the run, Discover produces a report of the memory anomalies, which you can view as a text file, or as HTML in a web browser.

Code Coverage Tool (Uncover)

Uncover is a simple and easy to use command-line tool for measuring code coverage of applications. Code coverage is an important part of software testing. It gives you information on which areas of your code are exercised in testing and which are not, enabling you to improve your test suites to test more of your code. The coverage information reported by Uncover can be at a function, statement, basic block, or instruction level.

Uncover provides a unique feature called uncoverage, which allows you to quickly find major functional areas that are not being tested. Other advantages of Uncover code coverage over other types of instrumentation are:

- The slowdown relative to uninstrumented code is fairly small.
- Since Uncover operates on binaries, it can work with any optimized binary.
- Measurements can be done by instrumenting the shipping binary. The application does not have to be built differently for coverage testing.
- Uncover provides a simple procedure for instrumenting the binary, running tests, and displaying the results.
- Uncover is multithread safe and multiprocess safe.

◆ ◆ ◆ CHAPTER 2

Memory Error Discovery Tool (Discover)

The Memory Error Discovery Tool (Discover) software is an advanced development tool for detecting memory access errors.

This chapter includes information about the following:

Requirements for Using Discover

Binaries Must Be Prepared Correctly

Discover works on binaries compiled with the Sun Studio 12 Update 1, Oracle Solaris Studio 12.2, or Oracle Solaris Studio 12.3 compilers; or the GCC for Sun Systems compilers starting with version 4.2.0. It works on a SPARC-based or x86-based system running the Solaris 10 10/08 operating system or a later Solaris 10 update, or Oracle Solaris 11.

Discover issues an error and does not instrument a binary if it does not meet these requirements. However, you can instrument a binary that does not meet these requirements and run it to detect a limited number of errors by using the `-l` option (see "Instrumentation Options" on page 16).

A binary compiled as described includes information called annotations to help Discover instrument it correctly. The addition of this small amount of information does not affect the performance of the binary or its runtime memory usage.

Using the `-g` option to generate debug information when compiling the binary allows Discover to display source code and line number information while reporting errors and warnings, and to produce more accurate results. If your binary is not compiled with the `-g` option, Discover displays only the program counters of the corresponding machine level instructions. Also, compiling with the `-g` option helps Discover produce more accurate reports (see "Interpreting Discover Error Messages" on page 32.

Binaries That Use Preloading or Auditing Cannot Be Used

Because Discover uses some special features of the runtime linker, you cannot use it with binaries that use preloading or auditing.

If a program requires the setting of the `LD_PRELOAD` environment variable, it probably won't work correctly with Discover, because Discover needs to interpose on certain system functions, and it cannot do so if the function has been preloaded.

Similarly, if a program uses runtime auditing (either the binary was linked with the `-p` option or the `-P` option, or it requires the `LD_AUDIT` environment variable to be set), this auditing will conflict with Discover's use of auditing. If the binary was linked with auditing, Discover fails at instrumentation time. If you set the `LD_AUDIT` environment variable at runtime, the results are undefined.

Binaries That Redefine Standard Memory Allocation Functions Can Be Used

Discover supports binaries that redefine the standard memory allocation functions: `malloc()`, `calloc()`, `memalign()`, `valloc()`, and `free()`.

Quick Start

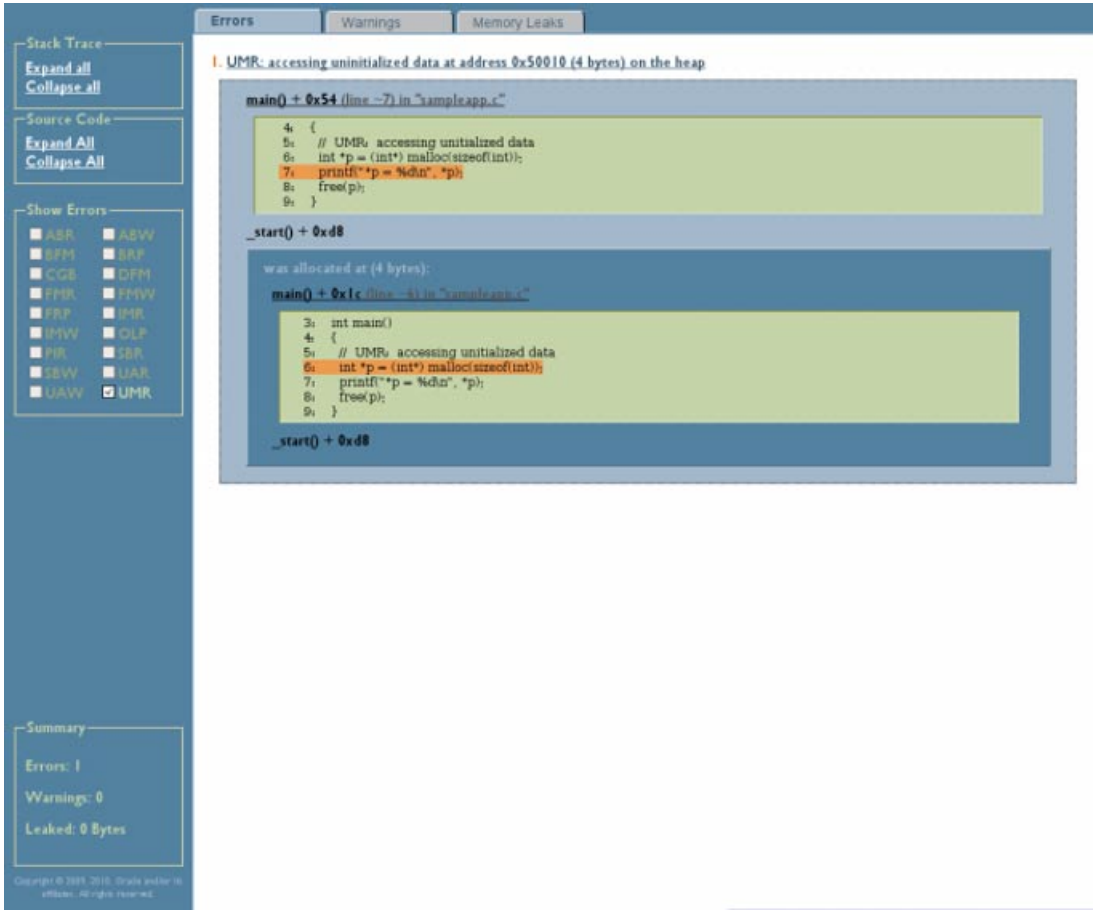
The following is an example of preparing a program, instrumenting it with Discover, and then running it and producing a report on the detected memory access errors. This example uses a simple program that accesses uninitialized data.

```
% cat test_UMR.c
#include <stdio.h>
#include <stdlib.h>
```

```
int main()
{
  //  UMR:  accessing uninitialized data
  int *p = (int*) malloc(sizeof(int));
  printf("*p = %d\n", *p);
  free(p);
}
```

```
% cc -g -02 test_UMR.c
% a.out
*p = 131464
% discover a.out
% a.out
```

The Discover output shows where the uninitialized memory was used and where it was allocated, along with summary of results.



Instrumenting a Prepared Binary

Once you have prepared the target binary, the next step is to instrument it. Instrumentation adds code in strategic places so that Discover can keep track of memory operations while the binary is running.

You instrument a binary using the `discover` command. For example, the following command instruments the binary `a.out` and overwrites the input `a.out` with the instrumented `a.out`:

```
discover a.out
```

When you run the instrumented binary, Discover monitors the program's use of memory. During the run, Discover writes a report detailing any memory access errors to an HTML file (in this case, by default, `a.out.html`) that you can view in your web browser. You can use the `-w` option when you instrument the binary to request that the report be written to an ASCII file or to `stderr`.

You can use the `-n` option to specify that you want Discover to do write-only instrumentation of the binary.

When Discover instruments a binary, if it finds any code that it cannot instrument because it is not annotated, it displays a warning like the following:

```
discover: (warning): a.out: 80% of code instrumented (16 out of 20 functions)
```

Non-annotated code could come from assembly language code linked into the binary, or from modules compiled with compilers or on operating systems older than those listed in "Binaries Must Be Prepared Correctly" on page 11.

Caching Shared Libraries

When Discover instruments a binary, it adds code to it that works with the runtime linker to instrument dependent shared libraries when they are loaded at runtime. The instrumented libraries are stored in a cache where they can be reused if the original has not changed since it was last instrumented. By default, the cache directory is `$HOME/SUNW_Bit_Cache`. You can change the directory with the `-D` option.

Instrumenting Shared Libraries

Discover produces the most accurate results if the entire program, including all shared libraries, is instrumented. By default, Discover checks and reports memory errors only in executables. You can use the `-c` option to specify that you want Discover to check for errors in the dependent shared libraries and libraries dynamically opened by `dlopen()`. You can use the `-n` option to specify that you want Discover to skip checking for errors in executables.

If you use the `-c` option to avoid checking for errors in a specific library, Discover does not report any errors in that library. However, Discover needs to track the memory state of the entire address space to correctly detect memory errors, so it records allocations and memory initializations in the entire program including all shared libraries.

All shared libraries used by the program should be prepared as described in "Binaries Must Be Prepared Correctly" on page 11. By default, if the runtime linker encounters an unprepared library, a fatal error occurs. You can, however, tell Discover to ignore one or more libraries.

Ignoring Libraries

Some libraries might not be possible to prepare, or they might not be instrumentable for some other reason. To provide for this case, with some loss of accuracy, you can tell Discover to ignore these libraries with the `-s`, `-T`, or `-N` option (see "Instrumentation Options" on page 16, or with specifications in `bit.rc` files (see "`bit.rc` Initialization Files" on page 18

If a library cannot be instrumented and is not designated as ignorable, then either Discover fails at instrumentation time or your program fails at runtime with an error message.

By default, Discover uses specifications in the system `bit.rc` file to set certain system and compiler-supplied libraries as ignorable because they are not prepared. The effect on accuracy is minimal because Discover knows the memory characteristics of the most commonly used libraries.

Command Line Options

You can use the following options with the `discover` command to instrument a binary.

Output Options

`-a` Write the error data to *binary_name*`.analyze/dynamic` directory for use by the Code Analyzer.

`-b` *browser* Start web browser *browser* automatically while running the instrumented program (off by default).

`-o` *file* Write the instrumented binary to *file*. By default, the instrumented binary overwrites the input binary.

`-w` *text_file* Write Discover's report on the binary to *text_file* . The file is created when you run the instrumented binary. If *text_file* is a relative pathname, the file is placed relative to the working directory where you run the instrumented binary. To make the filename unique for each time you run the binary, add the string `%p` to the filename to ask the Discover runtime to include the process id. For example,

the option `-w report.%p.txt` generates a report file with the filename `report.`*process_id*`.txt`. If you include `%p` in the filename more than once, only the first instance is replaced with the process id.

If you do not specify this option or the `-H` option, the report is written in HTML format to *output_file*`.html`, where *output_file* is the basename of the instrumented binary. The file is placed in the working directory where you run the instrumented binary.

You can specify both this option and the `-H` option to write the report in both text and HTML formats.

`-H` *html_file* Write Discover's report on the binary in HTML format to *html_file*. This file is created when you run the instrumented binary. If *html_file* is a relative pathname, it is placed relative to the working directory where you run the instrumented binary. To make the filename unique for each time you run the binary, add the string `%p` to the filename to ask the Discover runtime to include the process id. For example, the option `-H report.%p.html` generates a report file with the filename `report.`*process_id*`.html`. If you include `%p` in the filename more than once, only the first instance is replaced with the process id.

If you do not specify this option or the `-w` option, the report is written in HTML format to *output_file*`.html`, where *output_file* is the basename of the instrumented binary. The file is placed in the working directory where you run the instrumented binary.

You can specify both this option and the `-w` option to write the report in both text and HTML formats.

`-e` *n* Show only *n* memory errors in the report (default is show all errors).

`-E` *n* Show only *n* memory leaks in the report (default is 100).

`-f` Show offsets in the report (default is to hide them).

`-m` Show mangled names in the report (default is to show de-mangled names).

`-S` *n* Show only *n* stack frames in the report (default is 8).

Instrumentation Options

`-c` [`-` | *library* | *file*] Check for errors in all libraries, or in the specified *library*, or in the libraries listed in the specified *file*.

`-n` Do not check for errors in executables.

`-l` Run Discover in light mode. This option provides faster execution of your program and the program does not have to be specially prepared

as described in "Binaries Must Be Prepared Correctly" on page 11, but the number of errors detected is limited.

`-F [parent | child]` Specify what you want to happen if a binary you have instrumented with Discover forks while you are running it. By default, Discover continues to collect memory access error data from the parent process. If you want Discover to follow the fork and collect memory access data from the child process, specify `-F child`.

`-i` Instrument for data race detection using the Thread Analyzer. When you use this option, only data race detection is done at runtime; no other memory checking is done. The instrumented binary must be run with the `collect` command to generate an experiment that you can view in the Performance Analyzer (see the *Oracle Solaris Studio 12.3 Thread Analyzer User's Guide*).

`-s` Issue a warning, but do not flag an error, if an attempt is made to instrument an uninstrumentable binary.

`-T` Instrument the named binary only. Do not instrument any dependent shared libraries at runtime.

`-N` *library* Do not instrument any dependent shared library matching the prefix *library*. If the initial characters of a library name match *library*, the library is ignored. If *library* begins with a /, matching is done on the full absolute pathname of the library. Otherwise, matching is done on the basename of the library.

`-K` Do not read the `bit.rc` initialization files (see "`bit.rc` Initialization Files" on page 18).

Caching Options

`-D` *cache_directory* Use *cache_directory* as the root directory for storing cached instrumented binaries. By default, the cache directory is `$HOME/SUNW_Bit_Cache`.

`-k` Force reinstrumentation of any libraries found in the cache.

Other Options

`-h` or `-?` Help. Print a short usage message and exit.

`-v` Verbose. Print a log of what Discover is doing. Repeat the option for more information.

`-V` Print Discover version information and exit.

`bit.rc` Initialization Files

Discover initializes its state by reading a series of `bit.rc` files at startup. A system file, *Oracle_Solaris_Studio_installation_directory*`/prod/lib/postopt/bit.rc`, provides default values for certain variables. Discover reads this file first, followed by `$HOME/.bit.rc` if it exists, and *current_directory*`/.bit.rc` if it exists.

The `bit.rc` files contain commands to set, append to, and remove from certain variables. When Discover reads a `set` command, it discards the previous value, if any, of the variable. When it reads an `append` command, it appends the argument (after a colon separator) to the existing value of the variable. When it reads a `remove` command, it removes the argument and its colon separator from the existing value of the variable.

The variables set in the `bit.rc` files include the list of libraries to ignore when instrumenting, and lists of functions or function prefixes to ignore when computing the percentage of nonannotated (not prepared) code in a binary.

For more information, refer to the comments in the header of the system `bit.rc` file.

`SUNW_DISCOVER_OPTIONS` Environment Variable

You can change the runtime behavior of an instrumented binary by setting the `SUNW_DISCOVER_OPTIONS` environment variable to a list of the command-line options `-b`, `-e`, `-E`, `-f`, `-F`, `-H`, `-l`, `-L`, `-m`, `-S`, and `-w`. For example, if you want to change the number of errors reported to 50 and limit the stack depth in the report to 3, you would set the environment variable to `-e 50 -s 3`.

`SUNW_DISCOVER_FOLLOW_FORK_MODE` Environment Variable

By default, if a binary you have instrumented with Discover forks while you are running it, Discover continues to collect memory access error data from the parent process. If you want Discover to follow the fork and collect memory access data from the child process, set the `SUNW_DISCOVER_FOLLOW_FORK_MODE` environment variable.

Running an Instrumented Binary

After you have instrumented your binary with Discover, you run it the same way you would ordinarily. Typically, if a particular combination of input causes your program to behave strangely, you would instrument it with Discover and run it with the same input to investigate potential memory problems. While the instrumented program is running, Discover writes information about any memory problems it finds to the specified output files in the selected formats (text, HTML, or both). For information on interpreting the reports, see "Analyzing Discover Reports" on page 19.

Because of the overhead of the instrumentation, your program runs significantly slower after you instrument it. Depending on the frequency of memory access, it might run as much as 50 times slower.

Analyzing Discover Reports

The Discover report provides you with information to effectively pinpoint and fix the problems in your source code.

By default, the report is written in HTML format to *output_file*`.html`, where *output_file* is the basename of the instrumented binary. The file is placed in the working directory where you run the instrumented binary.

When you instrument your binary, you can use the `-H` option to request that the HTML output be written to a specified file, or the `-w` option to request that it be written to a text file (see "Command Line Options" on page 15).

After your binary is instrumented, you can change the settings of the `-H` and `-w` options for the report in the "`SUNW_DISCOVER_OPTIONS` Environment Variable" on page 18 if, for example, you want to write the report to a different file for a subsequent run of the program.

Analyzing the HTML Report

The HTML report format allows interactive analysis of your program. The data in HTML format can easily be shared between developers using email or placement on a web page. Combined with JavaScript interactive features, it provides a convenient way to navigate through the Discover messages.

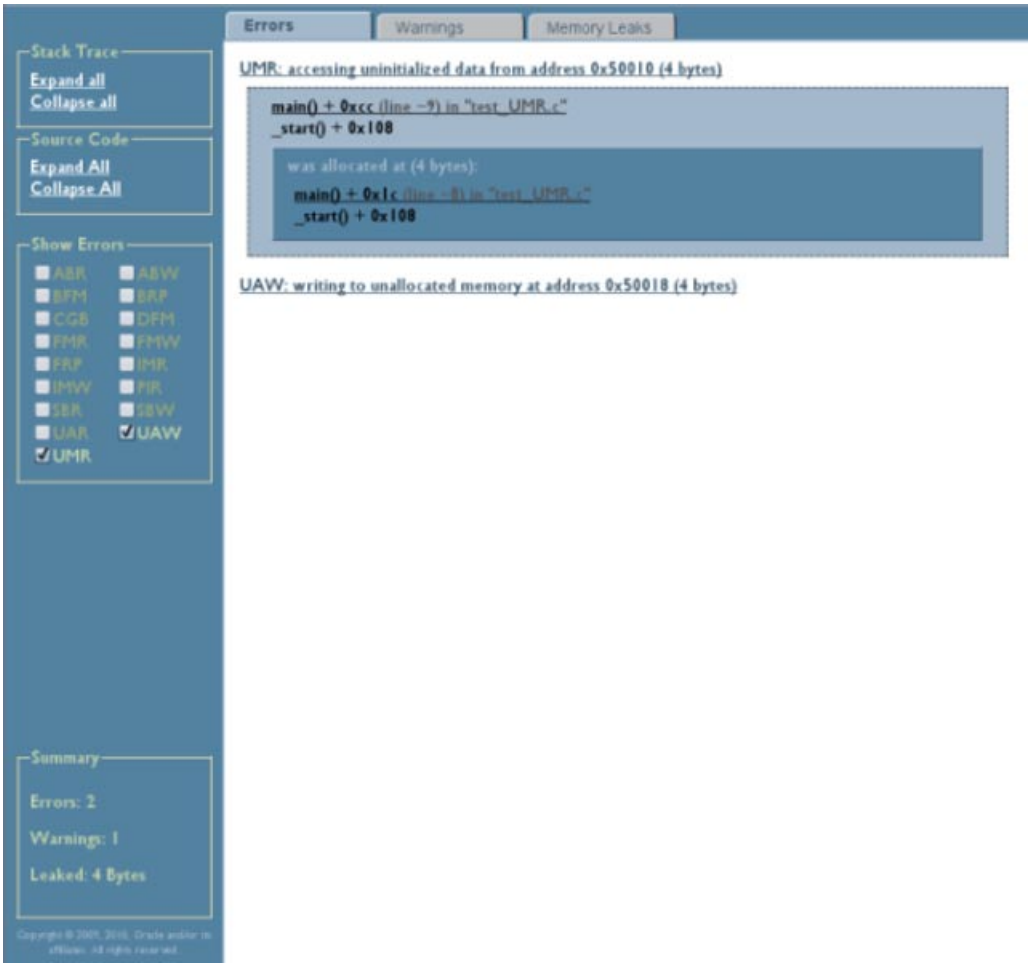
The Errors tab (see "Using the Errors Tab" on page 20), Warnings tab (see "Using the Warnings Tab" on page 22, and Memory Leaks tab (see "Using the Memory Leaks Tab" on page 23) let you navigate through error messages, warning messages, and the memory leak report, respectively.

The control panel on the left (see "Using the Control Panel" on page 25) lets you change the contents of the tab that is currently displayed on the right.

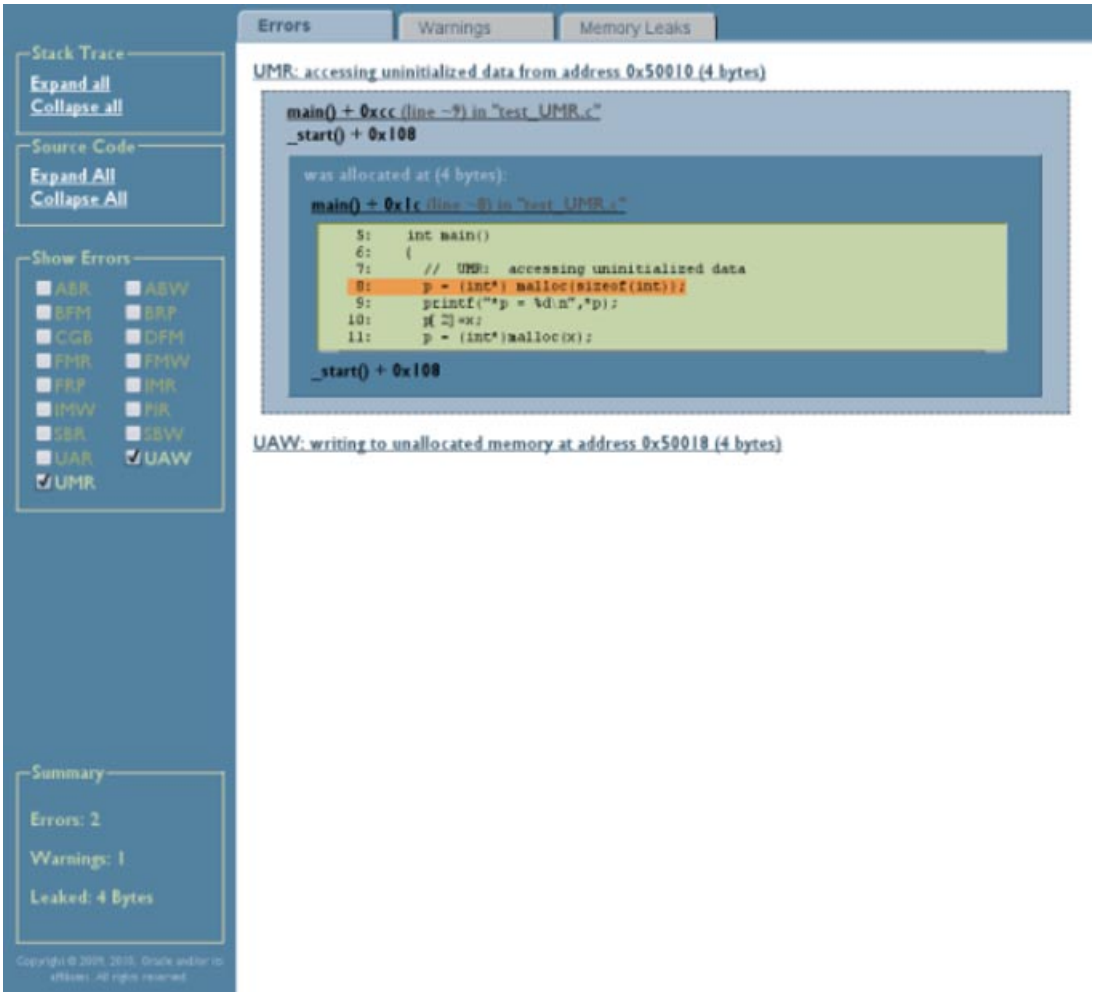
Using the Errors Tab

When you first open an HTML report in your browser, the Errors tab is selected and displays the list of memory access errors that occurred during execution of your instrumented binary.

When you click on an error, the stack trace at the time of the error is displayed:

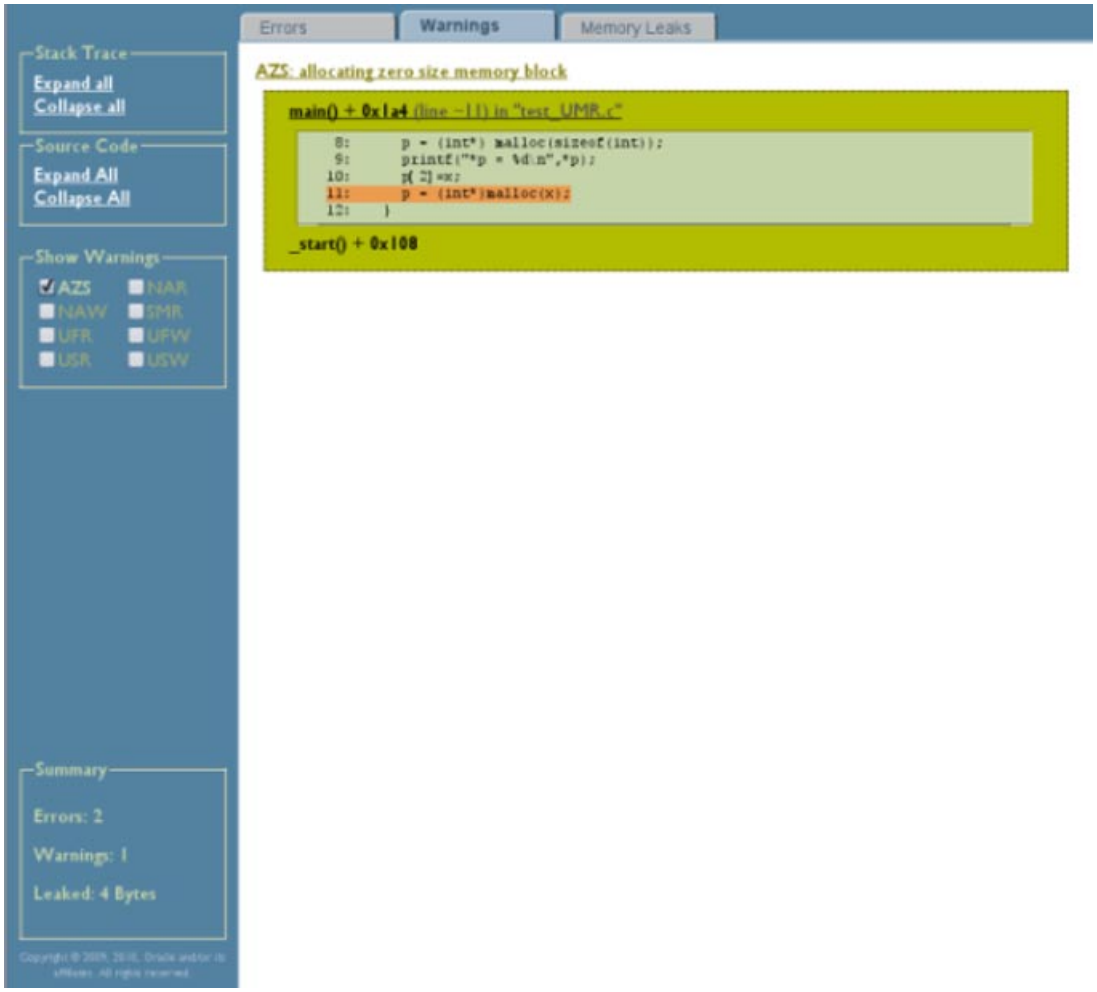


If you compiled your code with the -g option, you can see the source code for each function in the stack trace by clicking the function:



Using the Warnings Tab

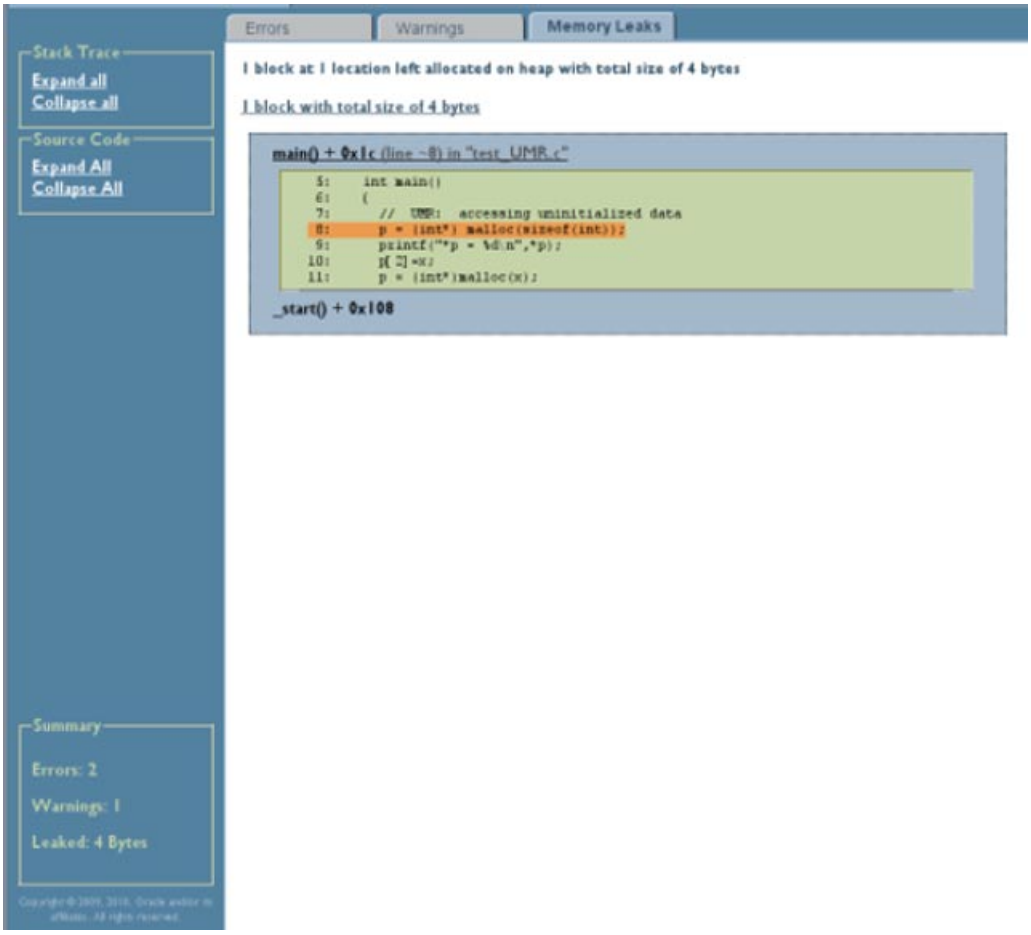
The Warnings tab displays all of the warning messages for possible access errors. When you click on a warning, the stack trace at the time of the warning is displayed. If you compiled your code with the `-g` option, you can see the source code for each function in the stack trace by clicking the function.



Using the Memory Leaks Tab

The Memory Leaks tab displays the total number of blocks remaining allocated at the end of the program's run at the top, with the blocks listed below.

When you click on a block, the stack trace for the block is displayed. If you compiled your code with the `-g` option, you can see the source code for each function in the stack trace by clicking the function.



Using the Control Panel

To see the stack traces for all of the errors, warnings, and memory leaks, click Expand All in the Stack Traces section of the control panel. To see the source code for all of the functions, click Expand All in the Source Code section of the control panel.

To hide the stack traces or source code for all of the errors, warnings, and memory leaks, click the corresponding Collapse All.

The Show Errors section of the control panel is displayed when the Errors tab is selected and lets you control which types of errors are displayed. By default, the checkboxes for all of the detected errors are checked. To hide a type of error, click its checkbox to remove the checkmark.

The Show Warnings section of the control panel is displayed when the Warnings tab is selected and lets you control which types of warnings are displayed. By default, the checkboxes for all of the detected warnings are checked. To hide a type of warning, click its checkbox to remove the checkmark.

A summary of the report listing the total numbers of errors and warnings, and the amount of leaked memory, is displayed at the bottom of the control panel.

Analyzing the ASCII Report

The ASCII (text) format of the Discover report is suitable for processing by scripts or when you don't have access to a web browser. The following is an example of an ASCII report.

```
$ a.out

ERROR 1 (UAW): writing to unallocated memory at address 0x50088 (4 bytes) at:
   main() + 0x2a0  <ui.c:20>
       17:     t = malloc(32);
       18:     printf("hello\n");
       19:     for (int i=0; i<100;i++)
       20:=>    t[32] = 234; // UAW
       21:     printf("%d\n", t[2]);   //UMR
       22:     foo();
       23:     bar();
   _start() + 0x108
ERROR 2 (UMR): accessing uninitialized data from address 0x50010 (4 bytes) at:
   main() + 0x16c  <ui.c:21>$
       18:     printf("hello\n");
       19:     for (int i=0; i<100;i++)
       20:      t[32] = 234; // UAW
       21:=>   printf("%d\n", t[2]);   //UMR
       22:     foo();
       23:     bar();
       24:    }
   _start() + 0x108
       was allocated at (32 bytes):
   main() + 0x24  <ui.c:17>
       14:     x = (int*)malloc(size); // AZS warning
       15:    }
       16:    int main() {
       17:=>   t = malloc(32);
       18:     printf("hello\n");
       19:     for (int i=0; i<100;i++)
       20:      t[32] = 234; // UAW
   _start() + 0x108
0
WARNING 1 (AZS): allocating zero size memory block at:
   foo() + 0xf4  <ui.c:14>
       11:    void foo() {
       12:     x = malloc(128);
       13:     free(x);
       14:=>   x = (int*)malloc(size); // AZS warning
```

```
       15:    }
       16:    int main() {
       17:     t = malloc(32);
   main() + 0x18c  <ui.c:22>
       19:     for (int i=0; i<100;i++)
       20:      t[32] = 234; // UAW
       21:     printf("%d\n", t[2]);   //UMR
       22:=>   foo();
       23:     bar();
       24:    }
   _start() + 0x108

***************** Discover Memory Report *****************

1 block at 1 location left allocated on heap with a total size of 128 bytes

   1 block with total size of 128 bytes
   bar() + 0x24  <ui.c:9>
       6:           7:    void bar() {
        8:     int *y;
        9:=>   y = malloc(128);  // Memory leak
       10:    }
       11:    void foo() {
       12:     x = malloc(128);
   main() + 0x194  <ui.c:23>
       20:      t[32] = 234; // UAW
       21:     printf("%d\n", t[2]);   //UMR
       22:     foo();
       23:=>   bar();
       24:    }
   _start() + 0x108

ERROR 1: repeats 100 times
DISCOVER SUMMARY:
   unique errors   : 2 (101 total, 0 filtered)
   unique warnings : 1 (1 total, 0 filtered)
```

The report consists of error and warning messages followed by a summary.

The error message starts with the word `ERROR` and contains a three-letter code, an id number, and an error description (`writing to unallocated memory` in the example). Other details include the memory address that was accessed and the number or bytes read or written. Following the description is a stack trace at the time of the error that pinpoints the location of the error in the process life cycle.

If the program was compiled with the `-g` option, the stack trace includes the source file name and line number. If the source file is accessible, the source code in the vicinity of the error is printed. The target source line in each frame is indicated by the `=>` symbol.

When the same kind of error at the same memory location with the same number of bytes repeats, the complete message including the stack trace is printed only once. Subsequent occurrences of the error are counted and a repetition count, as shown in the following example, is listed at the end of the report for each identical error that occurs multiple times.

```
ERROR 1: repeats 100 times
```

If the address of the faulty memory access is on the heap, then information on the corresponding heap block is printed after the stack trace. The information includes the block starting address and size, and a stack trace at the time the block was allocated. If the block was freed, a stack trace of the deallocation point is also included.

Warning messages are printed in the same format as error messages except that they start with the word `WARNING`. In general, these messages alert you to conditions that do not affect application correctness, but provide useful information that you can use to improve the program. For example, allocating memory of zero size is not harmful, but if it happens too often, it can potentially degrade performance.

The memory leak report contains information about memory blocks allocated on the heap but not released at program exit. The following is an example of a memory leak report.

```
$ DISCOVER_MEMORY_LEAKS=1 ./a.out
...
***************** Discover Memory Report *****************

2 blocks left allocated on heap with total size of 44 bytes
    block at 0x50008 (40 bytes long) was allocated at:
        malloc() + 0x168 [libdiscover.so:0xea54]
        f() + 0x1c [a.out:0x3001c]
          <discover_example.c:9>:
                 8:    {
                 9:=>    int *a = (int *)malloc( n * sizeof(int) );
                10:      int i, j, k;
        main() + 0x1c [a.out:0x304a8]
          <discover_example.c:33>:
                32:      /* Print first N=10 Fibonacci numbers */
                33:=>    a = f(N);
                34:      printf("First %d Fibonacci numbers:\n", N);
        _start() + 0x5c [a.out:0x105a8]
...
```

The first line following the header summarizes the number of heap blocks left allocated on the heap and their total size. The reported size is from the developer's perspective, that is, it does not include the bookkeeping overhead of the memory allocator.

After the memory leak summary, detailed information is printed on each unfreed heap block with a stack trace of its allocation point. The stack trace report is similar to the one described for error and warning messages.

The Discover report is concluded with an overall summary. It reports the number of unique warnings and errors and in parentheses, the total numbers of errors and warnings, including repeated ones. For example:

```
DISCOVER SUMMARY:
        unique errors   : 3 (3 total)
        unique warnings : 1 (5 total)
```

Memory Access Errors and Warnings

Discover detects and reports many memory access errors, as well as warning you about accesses that might be errors.

Memory Access Errors

Discover detects the following memory access errors:

- ABR: beyond Array Bounds Read
- ABW: beyond Array Bounds Write
- BFM: Bad Free Memory
- BRP: Bad Realloc address Parameter
- CGB: Corrupted array Guard Block
- DFM: Double Freeing Memory
- FMR: Freed Memory Read
- FMW: Freed Memory Write
- FRP: Freed Realloc Parameter
- IMR: Invalid Memory Read
- IMW: Invalid Memory Write
- Memory leak
- OLP: OverLaPping source and destination
- PIR: Partially Initialized Read
- SBR: beyond Stack frame Bounds Read
- SBW: beyond Stack frame Bounds Write
- UAR: UnAllocated memory Read
- UAW: UnAllocated memory Write
- UMR: Unitialized Memory Read

The following sections list some simple sample programs that will produce some of these errors.

ABR

```
//  ABR: reading memory beyond array bounds at address 0x%1x (%d byte%s)"
int *a = (int*) malloc(sizeof(int[5]));
printf("a[5] = %d\n",a[5]);
```

ABW

```
//  ABW:  writing to memory beyond array bounds
int *a = (int*) malloc(sizeof(int[5]));
a[5] = 5;
```

BFM

```
//  BFM:  freeing wrong memory block
int *p = (int*) malloc(sizeof(int));
free(p+1);
```

BRP

```
//  BRP is "bad address parameter for realloc 0x%1x"
int *p = (int*) realloc(0,sizeof(int));
int *q = (int*) realloc(p+20,sizeof(int[2]));
```

DFM

```
// DFM is "double freeing memory"
int *p = (int*) malloc(sizeof(int));
free(p);
free(p);'
```

FMR

```
//  FMR is "reading from freed memory at address 0x%1x  (%d byte%s)"
int *p = (int*) malloc(sizeof(int));
free(p);
printf("p = 0x%h\n",p);
```

FMW

```
//  FMW is "writing to freed memory at address 0x%1x (%d byte%s)"
int *p = (int*) malloc(sizeof(int));
free(p);
*p = 1;
```

FRP

```
//  FRP:  freed pointer passed to realloc
int *p = (int*) malloc(sizeof(int));
free(0);
int *q = (int*) realloc(p,sizeof(int[2]));
```

IMR

```
//  IMR:  read from invalid memory address
int *p = 0;
int i = *p;   // generates Signal 11...
```

IMW

```
//  IMW:  write to invalid memory address
int *p = 0;
*p = 1;       // generates Signal 11...
```

OLP

```
char *s=(char *) malloc(15);
memset(s, 'x', 15);
memcpy(s, s+5, 10);
return 0;
```

PIR

```
//  PIR:  accessing partially initialized data
int *p = (int*) malloc(sizeof(int));
*((char*)p) = 'c';
printf("*(p = %d\n",*(p+1));
```

SBR

```
int a[2]={0,1};
printf("a[-10]=%d\n",a[-10]);
return 0;
```

SBW

```
int a[2]={0,1)'
a[-10]=2;
return 0;
```

UAR

```
//  UAR is "reading from unallocated memory"
int *p = (int*) malloc(sizeof(int));
printf("*(p+1) = %d\n",*(p+1));
```

UAW

```
// UAW is "writing to unallocated memory"
int *p = (int*) malloc(sizeof(int));
*(p+1) = 1;
```

UMR

```
// UMR is "accessing uninitialized data from address 0x%1x (A%d byte%s)"
int *p = (int*) malloc(sizeof(int));
printf("*p = %d\n",*p);
```

Memory Access Warnings

Discover reports the following memory access warnings:

- AZS: allocating zero size
- SMR: speculative unitialized memory read

The following section lists a simple example program that will produce an AZS warning.

AZS

```
//  AZS:  allocating zero size memory block
int *p = malloc();
```

Interpreting Discover Error Messages

In some cases, Discover can report an error that is not actually an error. Such cases are called false positives. Discover analyzes code at instrumentation time to reduce the occurrence of false positives compared to similar tools, but there are cases where they still occur. The following sections provide a few tips that might help you to identify and possibly avoid false positives in Discover reports.

Partially Initialized Memory

Bit fields in C and C++ allow you to create compact data types. For example:

```
struct my_struct {
  unsigned int valid : 1;
  char         c;
};
```

In the example, the structure member `my_struct.valid` takes only one bit in memory. However, on SPARC platforms, the CPU can modify memory only in bytes, so the whole byte containing `struct.valid` must be loaded in order to access or modify the structure member. Moreover, sometimes the compiler might find it more efficient to load several bytes (for example, a machine word of four bytes) at once. When Discover detects such a load, without additional information it assumes that all four bytes are used. And if, for example, the field `my_struct.valid` was initialized, but the field `my_struct.c` was not, and the machine word containing both fields was loaded, Discover would flag a partially initialized memory read (PIR).

Another source of false positives is initialization of a bit field. To write a part of a byte, the compiler must first generate code that loads the byte. If the byte was not written prior to a read, the result is an uninitialized memory read error (UMR).

To avoid false positives for bit fields, use the `-g` option or the `-g0` option when compiling. These options provide extra debugging information to Discover to help it identify bit field loads and initialization, which will eliminate most false positives. If you cannot compile with the `-g` option for some reason, then initialize structures with a function such as `memset()`. For example:

```
...
struct my_struct s;
/* Initialize structure prio to use */
memset(&sm 0, sizeof(struct my_struct));
...
```

Speculative Loads

Sometimes the compiler generates a load from a known memory address under conditions where the result of the load is not valid on all program paths. This situation often occurs on SPARC platforms because such a load instruction can be placed in the delay slot of a branch instruction. For example, here is a C code fragment:

```
int i’
if (foo(&i) != 0) { /* foo returns nonzero if it has initialized i */
  printf("5d\n", i);
}
```

From this code, the compiler could generate code equivalent to:

```
int i;
int t1, t2’
t1 = foo(&i);
t2 = i; /* value in i is loaded */
if (t1 != 0) {
  printf("%d\n", t2);
}
```

Assume that in the example, the function `foo()` returns `0` and does not initialize `i`. The load from `i` is still generated, though not used. But the load will be seen by Discover, which will report a load of an uninitialized variable (UMR).

Discover uses dataflow analysis to identify such cases whenever possible, but sometimes they are impossible to detect.

You can reduce the occurrence of these types of false positives by compiling with a lower optimization level.

Uninstrumented Code

Sometimes it is not possible for Discover to instrument 100% of your program. Perhaps some of your code comes from an assembly language source file or a third-party library that cannot be recompiled and so cannot be instrumented. Discover has no knowledge of the memory blocks the non-instrumented code is accessing and modifying. Assume for example that a function from a third-party shared library initializes a block of memory that is later read by the main

(instrumented) program. Since Discover does not know that the memory has been initialized by the library, the subsequent read generates an uninitialized memory error (UMR).

To provide a solution for such cases, the Discover API includes the following functions:

```
void __ped_memory_write(unsigned long addr, long size, unsigned long pc);
void __ped_memory_read(unsigned long addr, long size, unsigned long pc);
void __ped_memory_copy(unsigned long src, unsigned lond dst, long size, unsigned long pc);
```

You can call the API functions from your program to inform Discover of specific events such as a write to a memory area (`__ped_memory_write()`) or a read from a memory area (`__ped_memory read()`). In both cases, the starting address of the memory area is passed in the `addr` parameter and its size is passed in the `size` parameter. Set the `pc` parameter to `0`.

Use the `__ped_memory_copy` function to inform Discover of memory that is being copied from one location to another. The starting address of the source memory is passed in the `src` parameter, the starting address of the destination area is passed in the `dst` parameter, and the size is passed in the `size` parameter. Set the `pc` parameter to `0`.

To use the API, declare these functions in your program as weak. For example, include the following code fragment in your source code.

```
#ifdef __cplusplus
extern "C" {
#endif

extern void __ped_memory_write(unsigned long addr, long size, unsigned long pc);
extern void __ped_memory_read(unsigned long addr, long size, unsigned long pc);
extern void __ped_memory_copy(unsigned long src, unsigned long dst, long size, unsigned long pc);

#prgama weak __ped_memory_write
#pragma weak __ped_memory_read
#pragma weak __ped_memory_copy

#ifdef __cplusplus
}
#endif
```

The API functions are defined in the internal Discover library, which is linked with your program at instrumentation time. However, when your program is not instrumented, this library is not linked and thus all calls to the API functions will result in application hang-up. So you must disable these functions when you are not running your program under Discover. Alternatively, you can create a dynamic library with empty definitions of the API functions and link it with your program. In this case, when you run your program without Discover, your library will be used, but when you run it under Discover, the real API functions will be called automatically.

Limitations When Using Discover

Only Annotated Code is Instrumented

Discover can instrument only code that has been prepared as described in "Binaries Must Be Prepared Correctly" on page 11. Non-annotated code might come from assembly language code linked into the binary, or from modules compiled with older compilers or operating systems than those listed in that section.

Specifically excluded from preparation are assembly language modules and functions that contain `asm` statements or `.il` templates.

Machine Instruction Might Differ From Source Code

Discover operates on machine code. The tool detects errors on machine instructions such as loads and stores, and correlates the errors with the source code. Some source code statements do not have associated machine instructions, so it may appear that Discover did not detect an obvious user error. For example, consider the following C code fragment:

```
int *p = (int *)malloc(sizeof(int));
int i;

i = *p;  /* compiler may not generate code for this statement */
printf("Hello World!\n");

return;
```

Reading a value stored at the address pointed to by `p` is a potential user error since the memory was not initialized. However, an optimizing compiler will detect that the variable `i` is not used, so the code for the statement reading from memory and assigning to `i` will not be generated. In this case, Discover will not report uninitialized memory usage (UMR).

Compiler Options Affect the Generated Code

Compiler-generated code is not always as you expect it to be. Because the code the compiler generates varies depending on the compiler options you use, including the `-O`*n* optimization options, the errors reported by Discover might also vary. For example, errors reported in code generated at the `-O1` optimization level could disappear for code generated at the `-O4` optimization level.

System Libraries Can Affect the Errors Reported

System libraries are preinstalled with the operating system and cannot be recompiled for instrumentation. Discover provides support for the common function from the standard C library (`libc.so`); that is, Discover knows what memory is accessed or modified by these functions. However, if your application uses other system libraries, you might see false positives in the Discover report. If false positives are reported, you can call the Discover API from your code to eliminate them.

Custom Memory Management Can Affect the Accuracy of the Data

Discover can track heap memory when it is allocated by standard programming language mechanisms like `malloc()`, `calloc()`, `free()`, `operator new()`, and `operator delete()`.

If your application uses a custom memory management system working on top of the standard functions (for example, pool allocation management implemented with `malloc()`), then Discover works, but is not guaranteed to correctly report leaks or access to freed memory.

Discover does not support the following memory allocators:

- Custom heap allocators that use `brk(2)()` or `sbrk(2)()` system calls directly
- Standard heap management function linked statically into a binary
- Memory allocated from the user code using `mmap(2)()` and `shmget(2)()` system calls

The `sigaltstack(2)()` function is not supported.

Out of Bounds Errors for Static and Automatic Arrays Cannot Be Detected

Because of the algorithms that Discover uses to detect array bounds, it is not possible to detect out of bounds access errors for static and automatic (local) arrays. Errors can be detected only for dynamically allocated arrays.

◆ ◆ ◆ CHAPTER 3

Code Coverage Tool (Uncover)

Requirements for Using Uncover

Uncover works on binaries compiled with the Sun Studio 12 Update 1, Oracle Solaris Studio 12.2, or Oracle Solaris Studio 12.3 compilers, or the GCC for Sun Systems 4.2.0 or later compilers. It work on a SPARC-based or x86-based system running the Solaris 10 10/08 operating system or a later Solaris 10 update, or Oracle Solaris 11.

A binary compiled as described includes information that Uncover uses to reliably disassemble the binary to instrument it for coverage data collection.

Using the `-g` option to generate debug information when compiling the binary allows Uncover to use source code level coverage information. If your binary is not compiled with the `-g` option, Uncover uses only program counter (PC) based coverage information.

Uncover works with any binary built with Oracle Solaris Studio compilers, but works best with binaries built with no optimization option. (Previous releases of Uncover required at least the `-O1` optimization level.) If your binary is built with an optimization option, Uncover results will be better with lower optimization levels (`-O1` or `-O2`). Uncover derives the source line level coverage by relating the instructions to line numbers using the debug information generated when the binary is built with the `-g`option. At optimization levels `-O3` and higher, the compiler might delete some code that might never be executed or is redundant, which might result in no binary instructions for some source code lines. In such cases, no coverage information will be reported for those lines. See "Limitations When Using Uncover" on page 53 for more information.

Using Uncover

Generating coverage information using Uncover is a three-step process:

1. Instrumenting the binary
2. Running the instrumented binary
3. Generating and viewing coverage reports

Instrumenting the Binary

The input binary can be an executable or a shared library. You must instrument each binary you want to analyze separately.

You instrument the binary with the `uncover` command. For example, the following command instruments the binary `a.out` and overwrites the input `a.out` with the instrumented `a.out`. It also creates a directory with the suffix `.uc` (`a.out.uc` in this case) in which the coverage data will be collected. A copy of the input binary is saved in this directory.

```
uncover a.out
```

You can use the following options when instrumenting your binary:

`-c`	Turn on reporting of execution counts for instructions, blocks, and functions. By default only information on code that is covered or not covered is reported. (Specify this option both when instrumenting your binary and when generating the coverage report.)
`-d` *directory*	Tells Uncover to create the coverage data directory in *directory*. This option is useful when you are collecting coverage data for multiple binaries, so that all of the coverage data directories are created in the same directory. Also, if you run different instances of the same instrumented binary from different locations, using this option ensures that the coverage data from all of these runs is accumulated in the same coverage data directory. If you do not use the `-d` option, the coverage data directory is created in the current run directory.
`-m on | off`	Turns thread-safe profiling on and off. The default is on. Use this option in combination with the `-c` runtime option. If you instrument a binary that uses threads with `-m off`, the binary fails at runtime and a message is displayed asking you to reinstrument the binary with `-m on`.
`-o` *output_binary_file*	Writes the instrumented binary file to the specified file. The default is to overwrite the input binary file with the instrumented file.

If you run the `uncover` command on a input binary that is already instrumented, Uncover issues an error message telling you that the binary cannot be instrumented because it is already instrumented, and that you can run it to generate coverage data.

Running the Instrumented Binary

After you have instrumented your binary, you can run it normally. Every time you run the instrumented binary, code coverage data is collected in the coverage data directory with the .uc suffix that Uncover created during the instrumentation. Since Uncover data collection is multi-thread safe and multi-process safe, there is no restriction on the number of simultaneous runs or threads in the process. The coverage data is accumulated over all of the runs and threads.

Generating and Viewing the Coverage Report

To generate a coverage report, run the `uncover` command on the coverage data directory. For example:

```
uncover a.out.uc
```

This command generates an Oracle Solaris Studio Performance Analyzer experiment directory called *binary_name*`.er` from the coverage data in the `a.out.uc` directory, starts the Performance Analyzer GUI, and displays the experiment. If you have an `.er.rc` file (see the *Oracle Solaris Studio 12.2 Performance Analyzer* manual) in the current directory or your home directory, it might affect the way the Analyzer displays the experiment.

You can also use `uncover` command options to generate the report as HTML and view it in your web browser, as ASCII to view in a terminal window. Or you direct the data to a directory where it can be analyzed and displayed by the Code Analyzer.

`-a`	Write error data to *binary_name*`.analyze/coverage` directory for use by the Code Analyzer.
`-c`	Turn on reporting of execution counts for instructions, blocks, and functions. By default only information on code that is covered or not covered is reported. (Specify this option both when instrumenting your binary and when generating the coverage report.)
`-e on | off`	Generate experiment directory for the coverage report and display the experiment in the Performance Analyzer GUI. On by default.
`-H` *html_directory*	Save the coverage data as HTML in the specified directory and automatically display it in your web browser. Off by default.
`-h` or `-?`	Help.

`-n`	Generate coverage reports but do not start viewers like the Performance Analyzer or web browser.
`-t` *ascii_file*	Generate an ASCII coverage report in the specified file. Off by default.
`-V`	Print Uncover version and exit.
`-v`	Verbose. Print a log of what Uncover is doing.

Only one output format is enabled, so if you specify multiple output options, Discover uses the last option in the command.

Examples

`uncover a.out`

This command instruments the binary `a.out`, overwrites the input `a.out`, creates an `a.out.uc` coverage data directory in the current directory, and saves a copy of the input `a.out` in the `a.out.uc` directory. If `a.out` is already instrumented, a warning message is displayed and no instrumentation is done.

`uncover -d coverage a.out`

This command does everything that the first example does, except it creates the `a.out.uc` coverage directory in the directory `coverage`.

`uncover a.out.uc`

This command uses the data in the `a.out.uc` coverage directory to create a code coverage experiment (`a.out.er`) in your working directory, and starts the Performance Analyzer GUI to display the experiment.

`uncover -H a.out.html a.out.uc`

This command uses the data in the `a.out.uc` coverage directory to create an HTML code coverage report in the directory `a.out.html` and displays the report in your web browser.

`uncover -t a.out.txt a.out.uc`

This command uses the data in the `a.out.uc` coverage directory to create an ASCII code coverage report in the file `a.out.txt`.

`uncover -a a.out.uc`

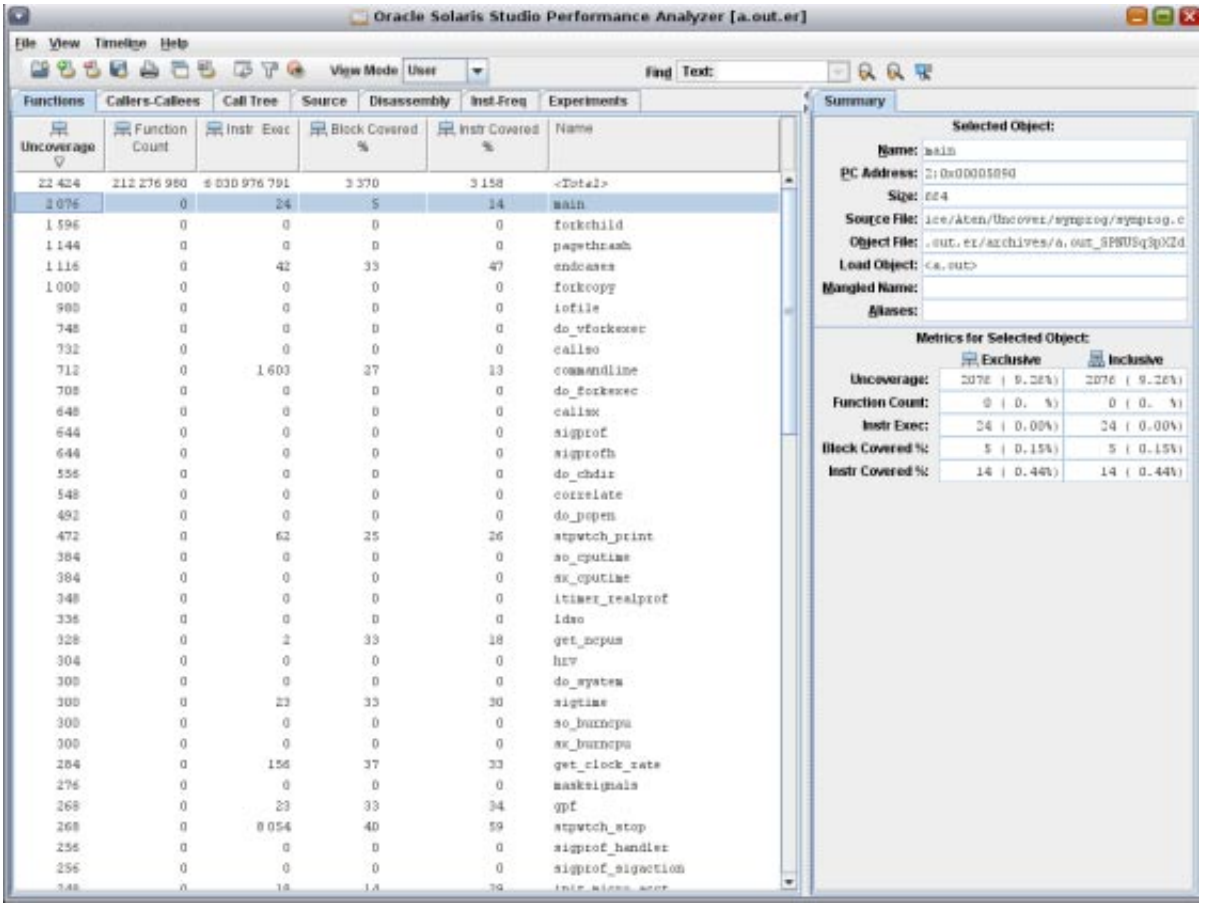
This command uses the data in the `a.out.c` coverage directory to create a coverage report in the *binary_name*`.analyze/coverage` directory for use by the Code Analyzer.

Understanding the Coverage Report in the Performance Analyzer

By default, when you run the `uncover` command on the coverage directory, the coverage report is opened as an experiment in the Oracle Solaris Studio Performance Analyzer. The Analyzer uses the Functions, Source, Disassembly, and Inst-Freq tabs to display the coverage data.

The Functions Tab

When you open the coverage report in the Analyzer, the Functions tab is selected. The tab displays columns listing the Uncoverage, Function Count, Instr Exec, Block Covered %, and Instr Covered % counters for each function. You can make any column the sort key for the data by clicking on the column header. Clicking the arrow on the column header reverses the sort order.



The Uncoverage Counter

The Uncoverage metric is a very powerful feature of Uncover. If you use this column as the sort key, in decreasing order, the top functions in the display are the functions that offer the greatest potential to increase coverage. In the example, the `main()` function is at the top of the list because it has the largest number in the Uncoverage column. (The `sigprof()` and `sigprofh()` functions all have the same number, so they are listed in alphabetical order.)

The Uncoverage number for the `main()` function is number of bytes of code that could potentially be covered if a test is added to the suite that causes the function to be called. The amount that coverage would actually increase varies according to the structure of the function. If there are no branches in the function, and all the functions it calls are also straight line functions, then coverage will indeed increase by the stated number of bytes. But in general, the coverage increase is less than the potential, perhaps much less.

The uncovered functions with non-zero values in the Uncoverage column are called root uncovered functions, meaning that they are all called by covered functions. Functions that are called only by non-root uncovered functions do not have their own uncoverage numbers. It is presumed that these functions will be either covered, or revealed as uncovered, in subsequent runs, as the test suite is improved to cover the high-potential uncovered functions.

The coverage numbers are non-exclusive.

The Function Count Counter

The Function Count reports the covered functions and uncovered functions. All that matters is whether the count is zero or non-zero. If the count is zero, the function is not covered. If the count is non-zero, the function is covered. If any instruction in the function is executed, the function is considered to be covered.

You can detect non-top-level uncovered functions in this column. If the Function Count for a function is zero and the Uncoverage number is also zero, the function is not a top-level covered function.

The Instr Exec Counter

The Instr Exec counter displays the covered instructions and uncovered instructions. A zero count means that the instruction is not executed; a non-zero count means that the instruction is executed.

In the Functions tab, this counter shows the total number of instructions executed for each function. This counter also appears in the Source tab (see "The Source Tab" on page 44) and the Disassembly tab (see "The Disassembly Tab" on page 45).

The Block Covered % Counter

For each function, the Block Covered % counter displays the percentage of basic blocks in the function that are covered. This number gives you an idea of how well the function is covered. Disregard this number in the `<Total>` row; it is the sum of percentages in the column and is meaningless.

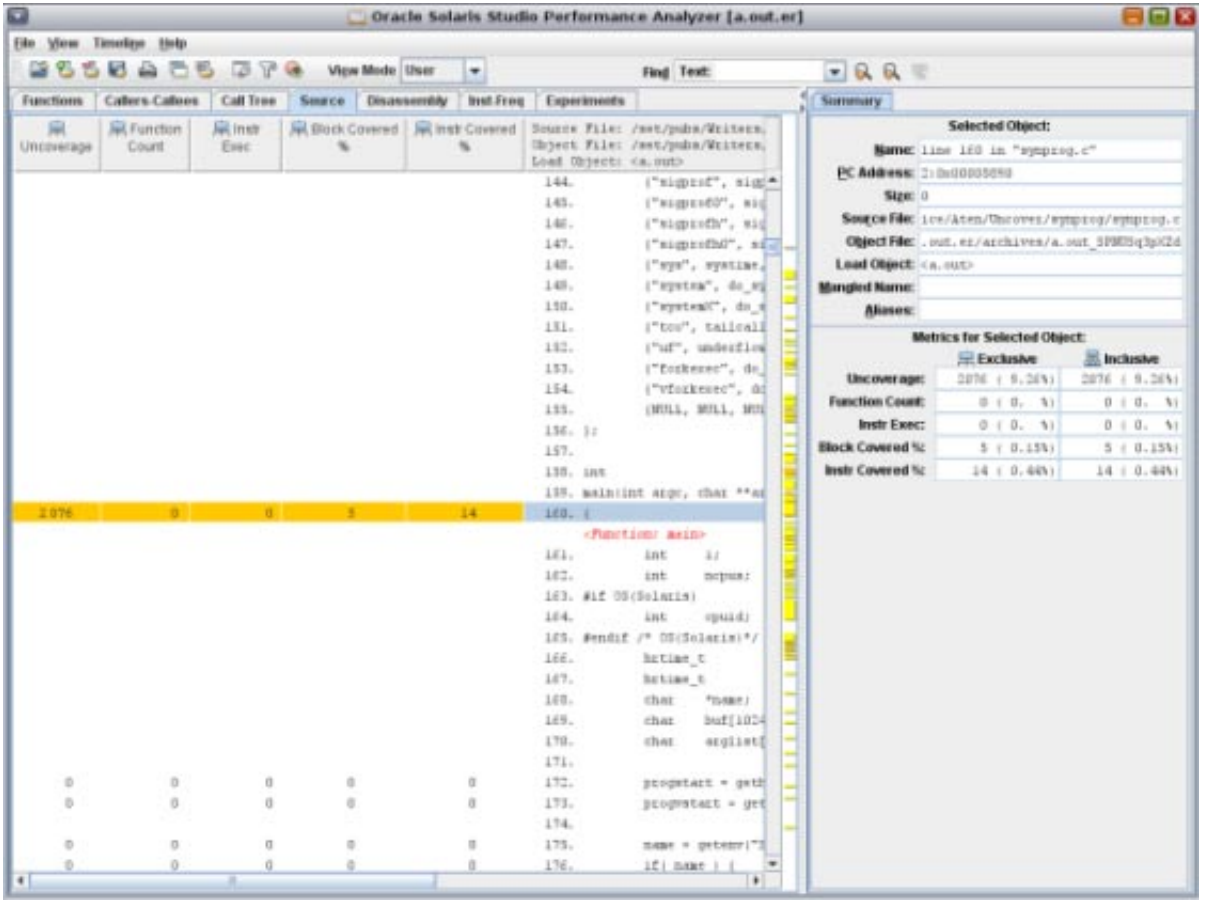
The Instr Covered % Counter

For each function, the Instr Covered % counter displays the percentage of instructions in the function that are covered. This number also gives you an idea of how well the function is covered. Disregard this number in the `<Total>` row; it is the sum of percentages in the column and is meaningless.

The Source Tab

If you compiled your binary with the `-g` option, the Source tab displays the source code of your program. Because Uncover instruments your program at the binary level, and you have compiled the program with optimization, the coverage information in this tab can be puzzling to interpret.

The Instr Exec counter in the Source tab shows the total number of instructions executed for each source line, which is essentially the statement level code coverage information. A non-zero value implies that the statement is covered; a zero value means that the statement is not covered. Variable declarations and comments have no Instr Exec counts.



Some source code lines might not have any coverage information associated with them. In these cases, the rows are blank and have no numbers in any of the fields. These rows occur because:

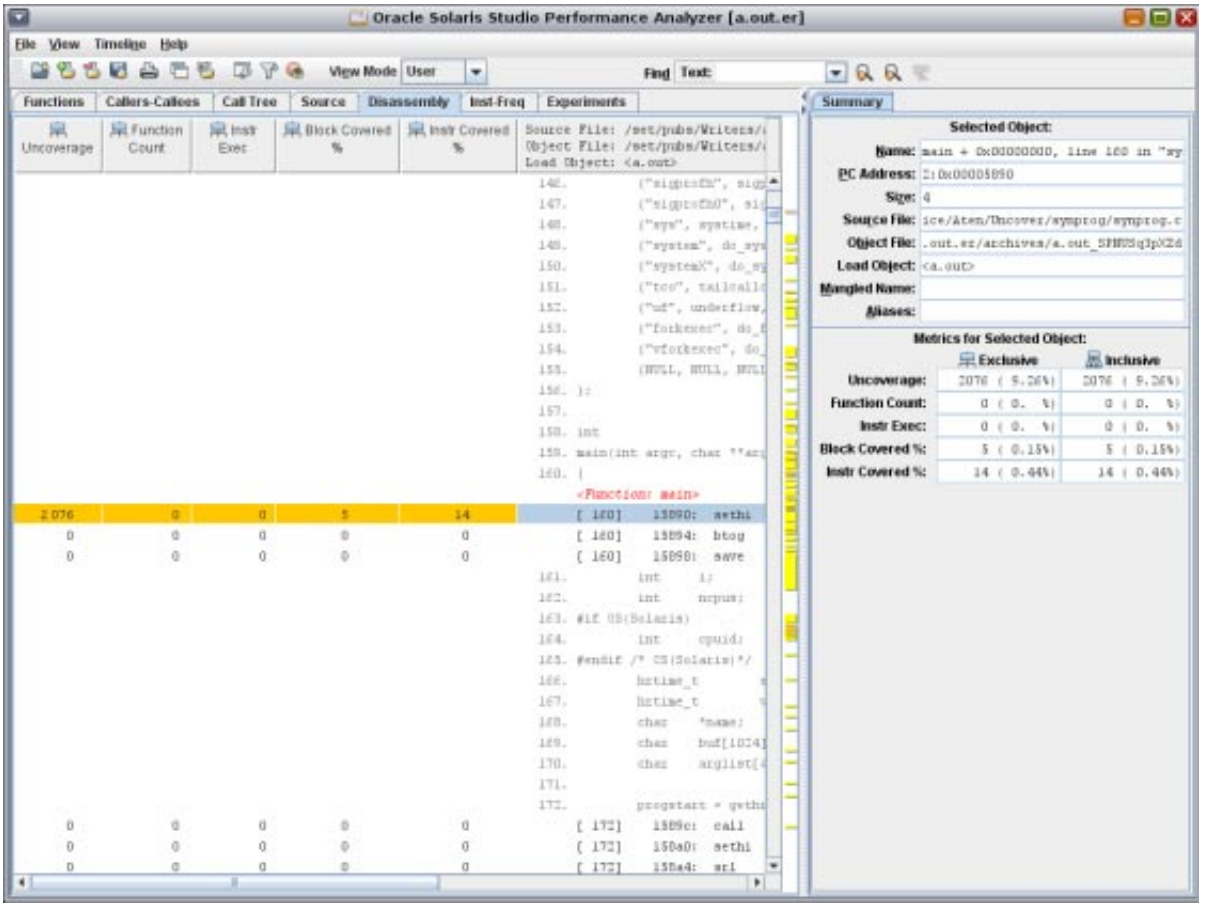
- Comments, blank lines, declarations, and other language constructs do not contain executable code.
- Compiler optimizations have deleted the code corresponding to the lines because:
  - The code will never be executed (dead code).
  - The code can be executed but is redundant.

For more information, see "Limitations When Using Uncover" on page 53.

The Disassembly Tab

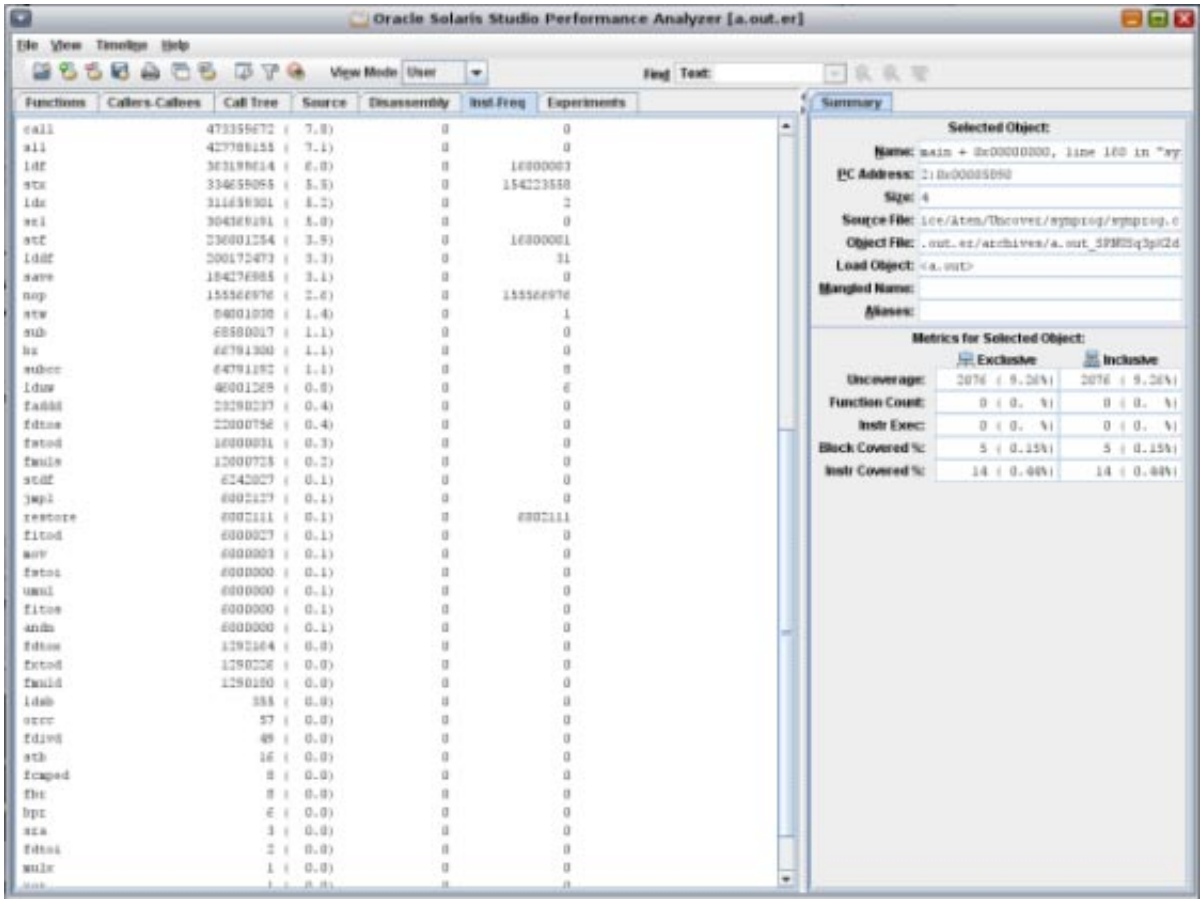
If you select a line in the Source tab, and then select the Disassembly tab, the Analyzer tries to find the selected line in the binary and display its disassembly.

The Instr Exec counter in this tab shows the number of times each instruction was executed.



The Inst-Freq Tab

The Inst-Freq tab displays the overall coverage summary.



Understanding the ASCII Coverage Report

If you specify the `-t` option when you generate the coverage report from the coverage data directory, Uncover writes a coverage report to the specified ASCII (text file).

```
UNCOVER Code Coverage
Total Functions: 95
Covered Functions: 58
Function Coverage: 61.1%
Total Basic Blocks: 568
Covered Basic Blocks: 258
Basic Block Coverage: 45.4%
Total Basic Block Executions: 564,812,760
Average Executions per Basic Block: 994,388.66
Total Instructions: 6,201
Covered Instructions: 3,006
Instruction Coverage: 48.5%
```

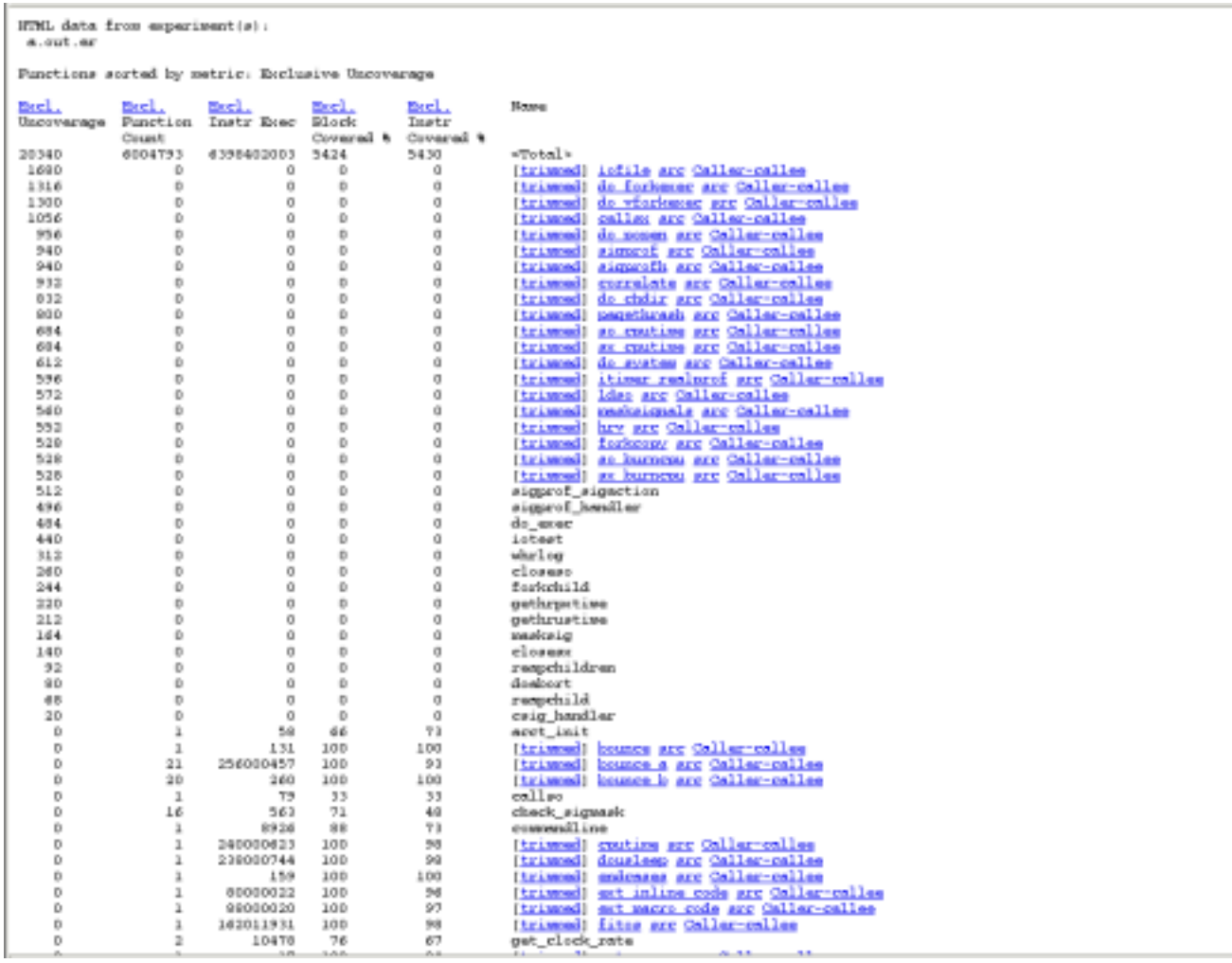
```
Total Instruction Executions: 4,760,934,518
Average Executions per Instruction: 767,768.83
Number of times this program was executed: unavailable
Functions sorted by metric: Exclusive Uncoverage

Excl.       Excl.     Excl.      Excl.       Name
Uncoverage  Function  Block      Instr
            Count     Covered %  Covered %
13404       6004876   5464       5384        <Total>
 1036             0      0          0        main
  980             0      0          0        iofile
  748             0      0          0        do_vforkexec
  732             0      0          0        callso
  708             0      0          0        do_forkexec
  648             0      0          0        callsx
  644             0      0          0        sigprof
  644             0      0          0        sigprofh
  556             0      0          0        do_chdir
  548             0      0          0        correlate
  492             0      0          0        do_popen
  404             0      0          0        pagethrash
  384             0      0          0        so_cputime
  384             0      0          0        sx_cputime
  348             0      0          0        itimer_realprof
  336             0      0          0        ldso
  304             0      0          0        hrv
  300             0      0          0        do_system
  300             0      0          0        do_burncpu
  300             0      0          0        sx_burncpu
  288             0      0          0        forkcopy
  276             0      0          0        masksignals
  256             0      0          0        sigprof_handler
  256             0      0          0        sigprof_sigaction
  216             0      0          0        do_exec
  196             0      0          0        iotest
  176             0      0          0        closeso
  156             0      0          0        gethrustime
  144             0      0          0        forkchild
  144             0      0          0        gethrpxtime
  136             0      0          0        whrlog
  112             0      0          0        masksig
   92             0      0          0        closesx
   84             0      0          0        reapchildren
   36             0      0          0        reapchild
   32             0      0          0        doabort
    8             0      0          0        csig_handler
    0             1     66         72        acct_init
    0             1    100        100        bounce
    0            63    100         96        bounce_a
    0            60    100        100        bounce-b
    0            16     71         58        check_sigmask
    0             1     83         77        commandline
    0             1    100         98        cputime
    0             1    100         98        dousleep
    0             1    100        100        endcases
    0             1    100         95        ext_inline_code
    0             1    100         96        ext_macro_code
    0             1    100         99        fitos
```

```
    0             2     81         80        get_clock_rate
    0             1    100        100        get_ncpus
    0             1    100        100        gpf
    0             1    100        100        gpf_a
    0             1    100        100        gpf_b
    0            10    100         93        gpf_work
    0             1    100         97        icputime
    0             1    100         96        inc_body
    0             1    100         96        inc_brace
    0             1    100         95        inc_entry
    0             1    100         95        inc_exit
    0             1    100         96        inc_func
    0             1    100         94        inc_middle
    0             1     57         72        init_micro_acct
    0             1     50         43        initcksig
    0             1    100         95        inline_code
    0             1    100         95        macro_code
    0             1    100         98        muldiv
    0       6000000    100        100        my_irand
    0             1    100         98        naptime
    0            19     50         83        prdelta
    0            21    100        100        prhrdelta
    0            21    100        100        prhrvdelta
    0             1    100        100        prtime
    0           552    100         98        real_recurse
    0             1    100        100        recurse
    0             1    100        100        recursedeep
    0             1    100         95        s_inline_code
    0             1    100        100        sigtime
    0             1    100         95        sigtime_handler
    0            19    100        100        snaptod
    0             1    100        100        so_init
    0             2     66         75        stpwtch_alloc
    0             1    100        100        stpwtch_calibrate
    0             2     75         66        stpwtch_print
    0          2002    100        100        stpwtch_start
    0          2000     90         91        stpwtch_stop
    0             1    100        100        sx_init
    0             1    100         99        systime
    0             3    100         95        tailcall_a
    0             3    100         95        tailcall_b
    0             3    100         95        tailcall_c
    0             1    100        100        tailcallopt
    0             1    100         97        underflow
    0            21     75         71        whrvlog
    0            19    100        100        wlog

Instruction frequency data from experiment a.out.er

Instruction frequencies of /export/home1/synprog/a.out.uc
Instruction               Executed     ()
 TOTAL                  4760934518 (100.0)
 float ops              2383657378 ( 50.1)
 float ld st            1149983523 ( 24.2)
 load store             1542440573 ( 32.4)
 load                    882693735 ( 18.5)
 store                   659746838 ( 13.9)
-----------------------------------------
```

```
Instruction              Executed     ()      Annulled   In Delay Slot
 TOTAL                 4760934518 (100.0)
 add                    713013787 ( 15.0)           16         1501335
 subcc                  558774858 ( 11.7)            0            6002
 br                     558769261 ( 11.7)            0               0
 stf                    432500661 (  9.1)          726        36299281
 ldf                    408226488 (  8.6)           40       103000396
 faddd                  391230847 (  8.2)            0               0
 fdtos                  366200726 (  7.7)            0               0
 fstod                  360200000 (  7.6)            0               0
 lddf                   288250336 (  6.1)          500       282200229
 stw                    138028738 (  2.9)        26002        25974065
 lduw                   118004305 (  2.5)           71        94000270
 ldx                     68212446 (  1.4)            0            2000
 stx                     68211370 (  1.4)            7        23532716
 fitod                   36026002 (  0.8)            0               0
 sethi                   36002986 (  0.8)            0             228
 fdtoi                   30000001 (  0.6)            0               0
 fdivd                   26000088 (  0.5)            0               0
 call                    22250348 (  0.5)            0               0
 srl                     21505246 (  0 5)            0              21
 stdf                    21006038 (  0.4)            0               0
 or                      19464766 (  0.4)            0        10981277
 fmuls                    6004907 (  0.3)            0               0
 jmpl                     6004853 (  0.1)            0               0
 save                     6004852 (  0.1)            0               0
 restore                  6002294 (  0.1)            0         6004852
 sub                      6000019 (  0.1)            0               0
 xor                      6000000 (  0.1)            0               0
 fitos                    6000000 (  0.1)            0               0
 fstoi                    6000000 (  0.1)            0               0
 and                      6000000 (  0.1)            0               0
 andn                     6000000 (  0.1)            0               0
 sll                      3505225 (  0.1)            0               0
 nop                      3505219 (  0.1)            0         3505219
 fxtod                       7763 (  0.0)            0               0
 bpr                         6000 (  0.0)            0               0
 fcmped                      4837 (  0.0)            0               0
 fbr                         4837 (  0.0)            0               0
 fmuld                       2850 (  0.0)            0               0
 orcc                         383 (  0.0)            0               0
 sra                          241 (  0.0)            0               0
 ldsb                         160 (  0.0)            0               0
 mulx                          87 (  0.0)            0               0
 stb                           31 (  0.0)            0               0
 mov                           21 (  0.0)            0               0
 fdtox                         15 (  0.0)            0               0
=========================================================
```

Understanding the HTML Coverage Report

The HTML report is similar to the report displayed in the Performance Analyzer.

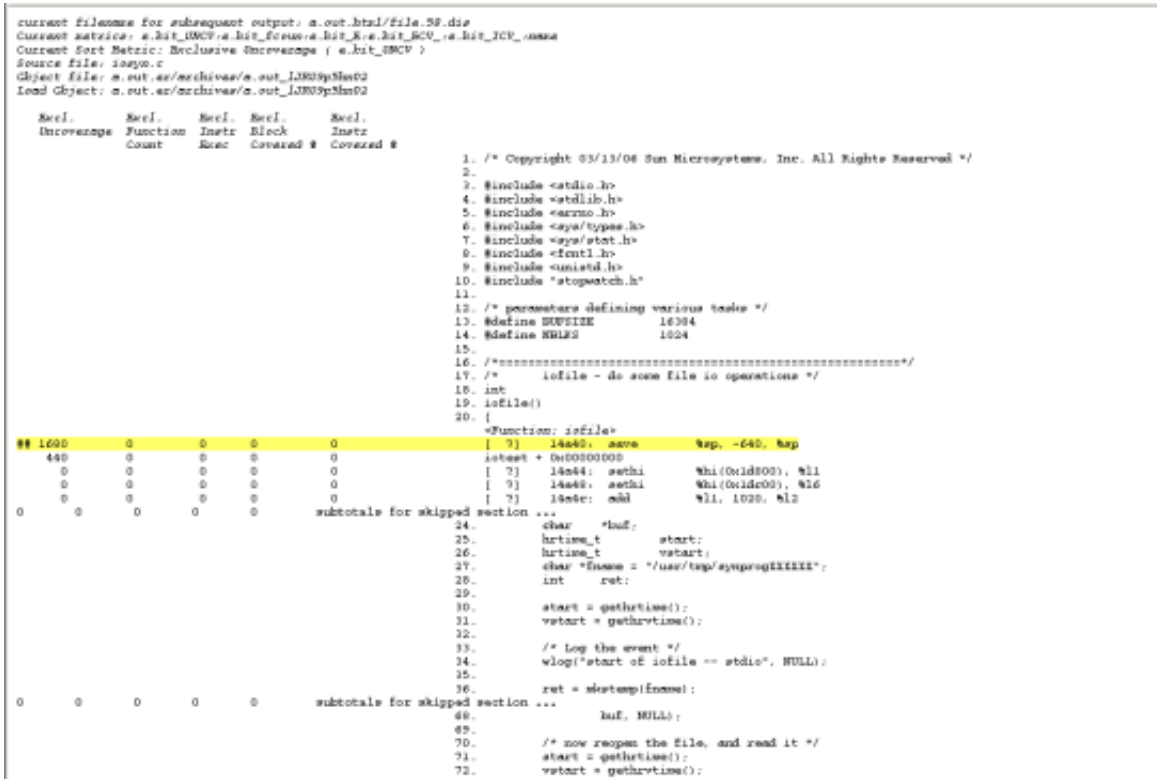


HTML data from experiment(s):
 a.out.er

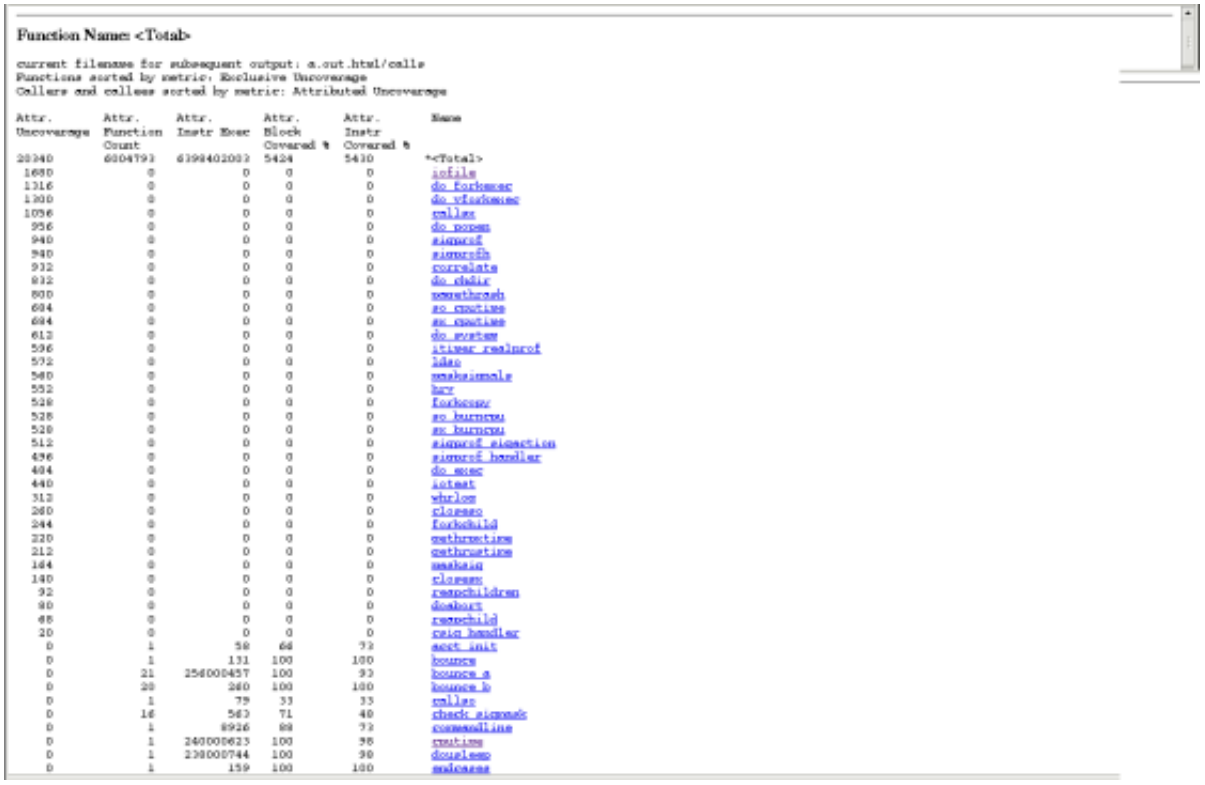
Functions sorted by metric: Exclusive Uncoverage

Excl. Uncoverage	Excl. Function Count	Excl. Instr Exec	Excl. Block Covered %	Excl. Instr Covered %	Name
20340	6004793	6398402003	5424	5430	<Total>
1680	0	0	0	0	[trimmed] isfile src Caller-callee
1316	0	0	0	0	[trimmed] do_forkexec src Caller-callee
1300	0	0	0	0	[trimmed] do_vforkexec src Caller-callee
1056	0	0	0	0	[trimmed] callso src Caller-callee
956	0	0	0	0	[trimmed] do_xopen src Caller-callee
940	0	0	0	0	[trimmed] sigprof src Caller-callee
940	0	0	0	0	[trimmed] sigprof5 src Caller-callee
932	0	0	0	0	[trimmed] correlate src Caller-callee
932	0	0	0	0	[trimmed] do_chdir src Caller-callee
900	0	0	0	0	[trimmed] pagefrash src Caller-callee
684	0	0	0	0	[trimmed] ze_routine src Caller-callee
684	0	0	0	0	[trimmed] zx_routine src Caller-callee
612	0	0	0	0	[trimmed] do_system src Caller-callee
596	0	0	0	0	[trimmed] itimer_realprof src Caller-callee
572	0	0	0	0	[trimmed] ldso src Caller-callee
560	0	0	0	0	[trimmed] enablesigsals src Caller-callee
552	0	0	0	0	[trimmed] hrv src Caller-callee
520	0	0	0	0	[trimmed] forkcopy src Caller-callee
528	0	0	0	0	[trimmed] zx_burncpu src Caller-callee
528	0	0	0	0	[trimmed] zx_burncpu src Caller-callee
512	0	0	0	0	sigprof_sigaction
496	0	0	0	0	sigprof_handler
484	0	0	0	0	do_exec
440	0	0	0	0	isteet
312	0	0	0	0	whelog
280	0	0	0	0	closeso
244	0	0	0	0	forkchild
220	0	0	0	0	gethrgetime
212	0	0	0	0	gethrustime
164	0	0	0	0	maskzig
140	0	0	0	0	closeso
92	0	0	0	0	reapchildren
80	0	0	0	0	doabort
68	0	0	0	0	reapchild
20	0	0	0	0	csig_handler
0	1	58	66	73	sret_init
0	1	131	100	100	[trimmed] bounce src Caller-callee
0	21	256030457	100	93	[trimmed] bounce_a src Caller-callee
0	20	260	100	100	[trimmed] bounce_b src Caller-callee
0	1	79	33	33	callso
0	16	561	71	40	check_sigmask
0	1	8928	88	73	commandline
0	1	240030823	100	98	[trimmed] routine src Caller-callee
0	1	238030744	100	98	[trimmed] douslesp src Caller-callee
0	1	159	100	100	[trimmed] endsesss src Caller-callee
0	1	80030022	100	98	[trimmed] get_inline_code src Caller-callee
0	1	98030020	100	97	[trimmed] get_macro_code src Caller-callee
0	1	162011931	100	98	[trimmed] fitss src Caller-callee
0	2	10470	76	67	get_clock_rate

If you click the function name link or the `trimmed` link for a function, the disassembly data for that function is displayed.



If you click the `Caller-callee` link for a function, the Caller-Callee data is displayed.



Function Names: <Total>

current filename for subsequent output: a.out.html/calls
Functions sorted by metric: Exclusive Uncoverage
Callers and callees sorted by metric: Attributed Uncoverage

Attr. Uncoverage	Attr. Function Count	Attr. Instr Exec	Attr. Block Covered %	Attr. Instr Covered %	Name
28240	6804793	6198402003	5424	5430	*<Total>
1680	0	0	0	0	zofile
1316	0	0	0	0	do_forkexec
1300	0	0	0	0	do_vforkexec
1056	0	0	0	0	exiller
956	0	0	0	0	do_xopen
940	0	0	0	0	sigmcnf
940	0	0	0	0	sigmcnfh
932	0	0	0	0	correlate
832	0	0	0	0	do_chdir
800	0	0	0	0	vexecthrough
684	0	0	0	0	xc_cnotices
684	0	0	0	0	xx_cnotices
612	0	0	0	0	do_system
596	0	0	0	0	itimer_realprof
572	0	0	0	0	idle
560	0	0	0	0	vmakesimple
552	0	0	0	0	her
528	0	0	0	0	forkcnv
528	0	0	0	0	xc_burncnv
528	0	0	0	0	xx_burncnv
512	0	0	0	0	sigmcnf_sigaction
496	0	0	0	0	sigmcnf_handler
484	0	0	0	0	do_exec
440	0	0	0	0	lofmat
312	0	0	0	0	whrlcw
260	0	0	0	0	closexc
244	0	0	0	0	forkchild
220	0	0	0	0	cathructicw
212	0	0	0	0	cathructicw
164	0	0	0	0	exakain
140	0	0	0	0	closexc
92	0	0	0	0	reapchildren
80	0	0	0	0	doabort
48	0	0	0	0	reapchild
20	0	0	0	0	cxic_handler
0	1	58	66	72	acct_init
0	1	131	100	100	bourcw
0	21	256000457	100	93	bourcw_a
0	20	240	100	100	bourcw_b
0	1	79	33	33	exiller
0	16	563	71	40	check_sigmask
0	1	6926	88	72	commandline
0	1	240000623	100	96	cnotices
0	1	230000744	100	90	doxclemo
0	1	159	100	100	exitcxcxc

Limitations When Using Uncover

Only Annotated Code Can Be Instrumented

Uncover can instrument only code that has been prepared as described in "Requirements for Using Uncover" on page 37. Non-annotated code might come from assembly language code linked into the binary, or from modules compiled with older compilers or operating systems than those listed in that section.

Specifically excluded from preparation are assembly language modules and functions that contain `asm` statements or `.il` templates.

Machine Instructions Might Differ From Source Code

Uncover operates on machine code. It finds coverage of machine instructions and then correlates this coverage with source code. Some source code statements do not have associated machine instructions, so Uncover might appear to not report coverage for such statements.

Example 1

Consider the following code fragment:

```
#define A 100
#define B 200
...
   if (A>B) {
        ...
   }
```

You might expect Uncover to report a non-zero execution count for the `if` statement, but the compiler is likely to remove this code, so Uncover will not see it during instrumentation. So no coverage will be reported for these instructions.

Example 2

The following is an example of dead code:

```
1  void foo()
2  {
3     A();
4     return;
5     B();
6     C();
7     D();
8     return;
9   }
```

Corresponding assembly shows that calls to B,C,D are deleted because this code is never executed.

```
foo:
                        .L900000109:
/* 000000          2 */         save    %sp,-96,%sp
/* 0x0004          3 */         call    A       ! params =      ! Result =
/* 0x0008            */         nop
/* 0x000c          8 */         ret     ! Result =
/* 0x0010            */         restore %g0,%g0,%g0
```

So no coverage will be reported for lines 5 through 6.

```
   Excl.       Excl.     Excl.  Excl.      Excl.
   Uncoverage  Function  Instr  Block      Instr
               Count     Exec   Covered %  Covered %
```

```
                                                        1.  void foo()
## 0          1         1      100        100          2.   {
                                                          <Function: foo
## 0          0         2        0          0          3.      A();
                                                        4.      return;
                                                        5.      B();
                                                        6.      C();
                                                        7.      D();
                                                        8.     return;
## 0          0         2        0          0          9.    }
```

Example 3

The following is an example of redundant code:

```
1  int g;
2  int foo() {
3    int x;
4    x = g;
5    for (int i=0; i<100; i++)
6        x++;
7    return x;
8  }
```

At low optimization levels, the compiler may generate code for all the lines:

```
foo:
                        .L900000107:
 /* 000000          3 */         save    %sp,-112,%sp
 /* 0x0004          5 */         sethi   %hi(g),%l1
 /* 0x0008            */         ld      [%l1+%lo(g)],%l3 ! volatile
 /* 0x000c            */         add     %l1,%lo(g),%l2
 /* 0x0010          6 */         st      %g0,[%fp-12]
 /* 0x0014          5 */         st      %l3,[%fp-8]
 /* 0x0018          6 */         ld      [%fp-12],%l4
 /* 0x001c            */         cmp     %l4,100
 /* 0x0020            */         bge,a,pn        %icc,.L900000105
 /* 0x0024          8 */         ld      [%fp-8],%l1
                       .L17:
 /* 0x0028          7 */         ld      [%fp-8],%l1
                        .L900000104:
 /* 0x002c          6 */         ld      [%fp-12],%l3
 /* 0x0030          7 */         add     %l1,1,%l2
 /* 0x0034            */         st      %l2,[%fp-8]
 /* 0x0038          6 */         add     %l3,1,%l4
 /* 0x003c            */         st      %l4,[%fp-12]
 /* 0x0040            */         ld      [%fp-12],%l5
 /* 0x0044            */         cmp     %l5,100
 /* 0x0048            */         bl,a,pn %icc,.L900000104
 /* 0x004c          7 */         ld      [%fp-8],%l1
 /* 0x0050          8 */         ld      [%fp-8],%l1
                        .L900000105:
 /* 0x0054          8 */         st      %l1,[%fp-4]
 /* 0x0058            */         ld      [%fp-4],%i0
 /* 0x005c            */         ret     ! Result =  %i0
```

```
 /* 0x0060            */         restore %g0,%g0,%g0
```

At high optimization levels, most of the executable source lines do not have any corresponding instructions :

```
foo:
 /* 000000          5 */         sethi   %hi(g),%o5
 /* 0x0004            */         ld      [%o5+%lo(g)],%o4
 /* 0x0008          8 */         retl    ! Result =  %o0
 /* 0x000c          5 */         add     %o4,100,%o0
```

So no coverage will be reported for some lines.

```
    Excl.       Excl.     Excl.  Excl.      Excl.
   Uncoverage  Function  Instr  Block      Instr
               Count     Exec   Covered %  Covered %
                                                      1. int g;
   0           0         0        0          0        2. int foo() {
                                                      <Function foo>
                                                      3.   int x;
                                                      4.   x = g;

                                                    Source loop below has tag L1
                                                    Induction variable substitution performed on L1
                                                    L1 deleted as dead code
## 0           1         3      100        100        5.   for (int i=0; i<100; i++)
                                                      6.       x++;
                                                      7.   return x;
   0           0         1        0          0        8. }
```

Index