Oracle® Solaris Studio 12.3 Overview

Part No: E21989
December 2011

ORACLE

120118@25097

Contents

Preface

This manual is for C, C++, and Fortran application developers who are new to Oracle Solaris Studio and want to get an understanding of the many tools, compilers, and programming libraries available in the product. The manual does not provide detailed information about using the tools, but does show how a developer might use them together to edit, build, and analyze software applications under development.

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	Description	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#`

Oracle Solaris Studio 12.3 Overview

Oracle Solaris Studio software is a set of software development tools for C, C++, and Fortran development on Solaris and Linux platforms, with support of multicore systems with SPARC® processors and x86 processors.

Introduction to Oracle Solaris Studio Software

Oracle Solaris Studio consists of two suites of tools: a compiler suite and an analysis suite. The tools of each suite are designed to work together to provide an optimized development environment for the development of single, multithreaded, and distributed applications.

Oracle Solaris Studio provides everything you need to develop C, C++, and Fortran applications to run in Oracle Solaris 10 or Oracle Solaris 11 on SPARC or x86 and x64 platforms, or in Oracle Linux on x86 and x64 platforms. The compilers and analysis tools are engineered to make your applications run optimally on Oracle Sun systems.

In particular, Oracle Solaris Studio compilers and analysis tools are designed to leverage the capabilities of multicore CPUs including the SPARC T4, SPARC T3, UltraSPARC T2, and UltraSPARC T2 Plus processors, and the Intel® Xeon® and AMD Opteron processors. Oracle Solaris Studio enables you to more easily create parallel and concurrent software applications for these platforms.

The components of Oracle Solaris Studio include:

- IDE for application development in a graphical environment. The Oracle Solaris Studio IDE integrates several other Oracle Solaris Studio tools and uses Oracle Solaris technologies such as DTrace.
- C, C++, and Fortran compilers for compiling your code at the command line or through the IDE. The compilers are engineered to work well with the Oracle Solaris Studio debugger (`dbx`), and include options for optimizing your code for specific processors.
- Libraries to add advanced performance and multithreading capabilities to your applications.

- Make utility (`dmake`) for building your code in distributed computing environments at the command line or through the IDE.
- Debugger (`dbx`) for finding bugs in your code at the command line, or through the IDE, or through an independent graphical interface (`dbxtool`).
- Code Analyzer tools for finding static code errors in your code during compilation, and memory access and code coverage errors during execution.
- Performance Analyzer tools that employ Oracle Solaris technologies such as DTrace and can be used at the command line or through graphical interfaces to find trouble spots in your code that you cannot detect through debugging.
- Thread Analyzer for examining multithreaded programs to detect programming errors that cause data races and deadlocks.

These tools together enable you to build, debug, and tune your applications for high performance on Oracle Solaris running on Oracle Sun systems. Each component is described in greater detail later in this document.

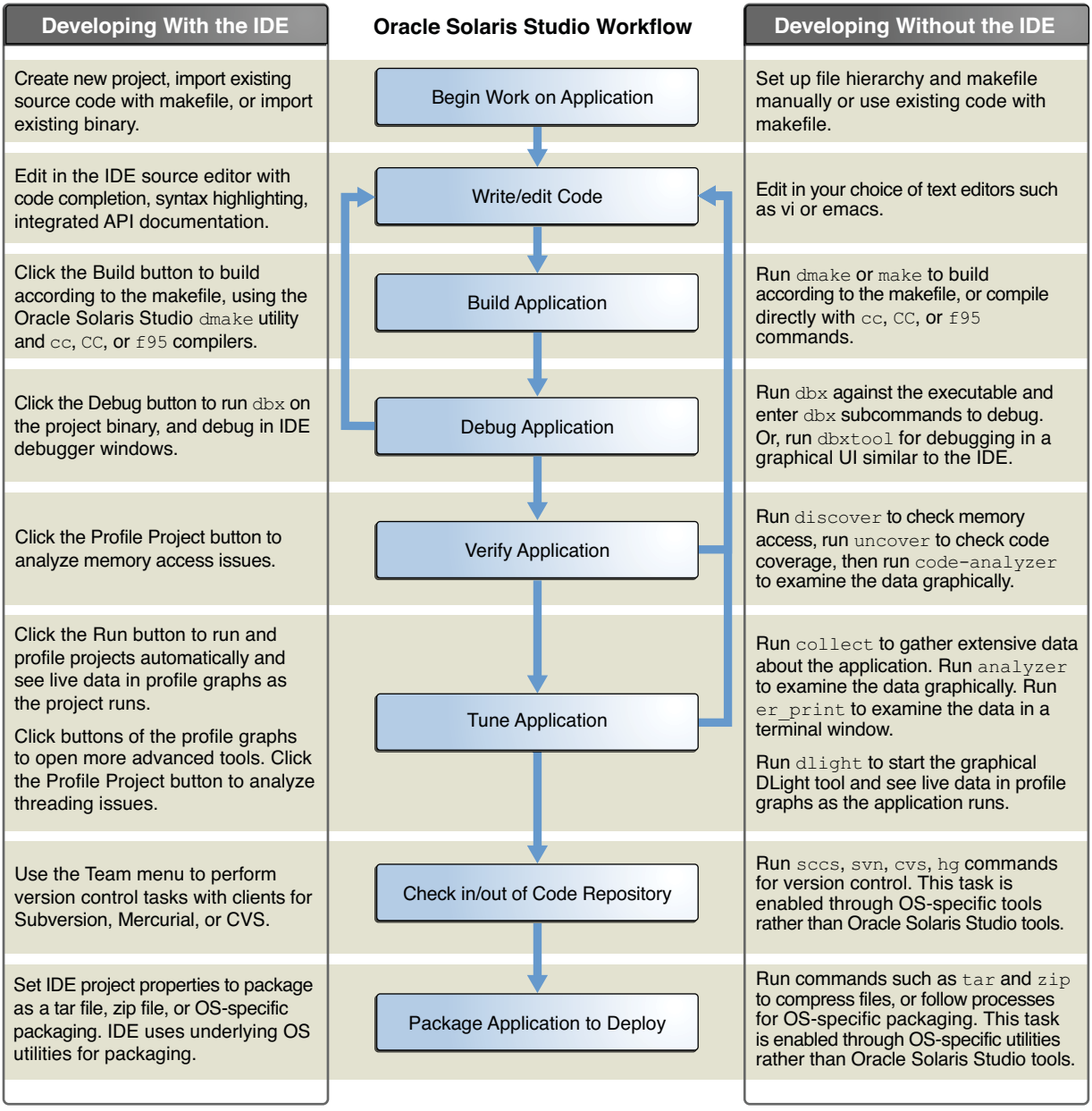
Developer Workflow for Oracle Solaris Studio

Oracle Solaris Studio provides tools to help developers create applications that run on Oracle Solaris. The tools can support developers who want a graphical IDE to manage many development tasks for them, and developers who want to control all aspects of their software development using their own methods.

You do not need to make a commitment to using the IDE or the command line because the tools are designed to be used in any combination. You can create a project in the IDE and still build the source of the project with `dmake` or `make` from the command line if you want. You can use the Performance Analyzer or DLight on the binary of a project you created in the IDE. The IDE keeps its project files separate from the source files so there is no dependency.

If you are a devoted Emacs or vi user, you can continue to use your accustomed environment and ignore the IDE, but adopt the Oracle Solaris Studio compilers and performance tools to make your application run optimally in Oracle Solaris on Oracle Sun hardware.

The following diagram shows the developer workflow for the Oracle Solaris Studio tools when developing with or without the graphical IDE.



The rest of this document describes the components of the Oracle Solaris Studio software, explains the ways that the components are integrated, and briefly shows how to use them.

Oracle Solaris Studio IDE

The Oracle Solaris Studio IDE is based on the NetBeans IDE, an open-source integrated development environment. The Oracle Solaris Studio IDE includes the core NetBeans IDE, the NetBeans C/C++ plugin module, and additional integrated Oracle Solaris Studio components that are not available in the open-source NetBeans IDE.

The Oracle Solaris Studio IDE uses the Oracle Solaris Studio C, C++, and Fortran compilers, the `dmake` build utility, and the `dbx` debugger. In addition, the IDE provides graphical profiling tools that use Oracle Solaris DTrace and Oracle Solaris Studio performance utilities invisibly to collect data on your running project.

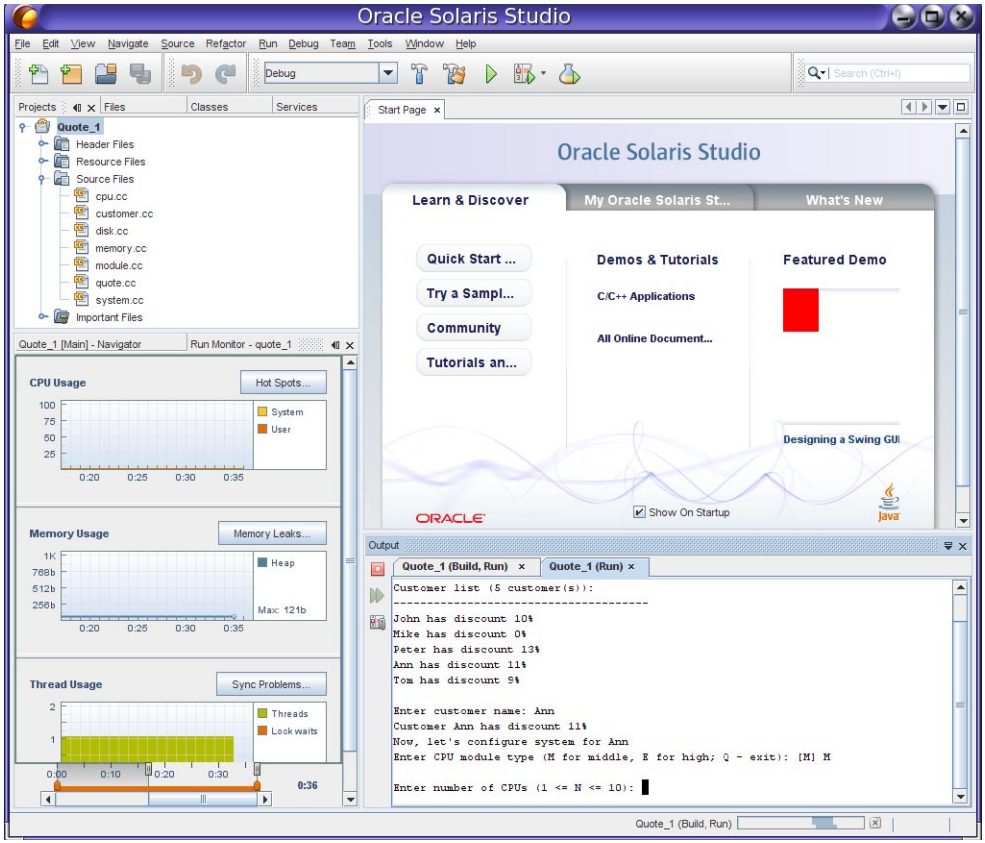
Using the Oracle Solaris Studio IDE offers some advantages over development using text editors and the command line:

- **Code editing**. Working with code can be more efficient with syntax highlighting, code completion, navigation between code elements, and integrated API documentation and man pages.
- **Code investigation**. When you are trying to become familiar with some code or looking for a root cause of a bug, you might find useful such IDE features as Go to Symbol, Find Usages, the Classes window, the Include hierarchy, and the Call Graph.
- **Refactoring**. You can find all usages of a variable or operation within a project, and rename all occurrences of the variable or operation and refactor throughout the product. Before performing the refactoring, you can preview the changes in a split-screen UI and approve them individually or all at once.
- **Remote development**. You can run the IDE on a desktop system while using the Oracle Solaris Studio compilers and tools that are installed on a remote server. The tools run on the remote server while displaying back to your IDE which is running on your local system, with much better response time than typical remote display solutions.

To start the IDE, type the following command:

```
% solstudio
```

The following figure shows the IDE with the Quote sample project running.



A C or C++ or Fortran project is a group of source files and associated information about how to compile and link the source files and run the resulting program. In the IDE, you always work inside a project even if your program is contained in a single source file. The IDE stores project information in a project folder that includes a makefile and metadata files. Your source directories do not need to be physically located in the project folder.

Each project (except a project created from an existing binary) must have a makefile so the IDE can build the project. A project's makefile can be generated by the IDE or you can use a makefile that was previously created outside the IDE. You can create projects from existing sources that already include a makefile, or that build a makefile when you run configure scripts.

You can build, run, and debug projects by clicking toolbar buttons, or by selecting menu commands. The IDE is preconfigured to use the Studio C, C++, and Fortran compilers, `dmake`, and `dbx` by default. However, if you have GNU compilers on your system, the IDE can usually find them if they are on your PATH. You can use the GNU tool collection by setting the Tool Collection in your project's properties.

You can learn about using the Oracle Solaris Studio IDE by reading the IDE's integrated help, which you can access through the IDE's Help menu or by pressing the F1 key. Many dialog boxes also have a Help button for information about how to use the dialog box.

The Oracle Solaris Studio 12.3: IDE Quick Start Tutorial shows how to get started using the IDE. In addition, the tutorials on the NetBeans IDE C/C++ Learning Trail can also be helpful for learning how to use the Oracle Solaris Studio IDE, although there are some differences between the user interfaces and features. In particular, NetBeans documentation about debugging does not apply to Oracle Solaris Studio IDE.

Oracle Solaris Studio Compilers

Oracle Solaris Studio software includes C, C++, and Fortran compilers, which have the following features:

- Comply with modern standards for C, C++, and Fortran programming languages.
- Produce code that is targeted for specific operating systems, processors, architectures, memory models (32–bit and 64–bit), floating-point arithmetic, and more, according to command-line options you specify.
- Perform automatic parallelization on serial source code to produce binaries that exhibit enhanced performance on multicore systems.
- Produce code that is optimized in ways that you can specify through command-line options to suit your application and deployment environment.
- Prepare binaries for enhanced debugging or analysis by other Oracle Solaris Studio tools.
- Use the same command-line options across all the compilers to specify these features.

Some of the Oracle Solaris Studio compiler options you can use to optimize your compiled code for speed and take the best advantage of processor instruction sets and features are as follows:

`-O`*n* Specifies a level of optimization indicated by *n*, which can be a number from 1 to 5. A higher optimization level creates a binary with better runtime performance.

`-fast` Selects the optimum combination of compilation options for speed of executable code. `-fast` can be used effectively as a starting point for tuning an executable for maximum runtime performance.

`-g` Produces additional information in the binary for debugging with `dbx` and analysis with Performance Analyzer. Compiling with the `-g` option enables you to use the full capabilities of the Performance Analyzer, such as viewing annotated source, function information, and compiler commentary messages that describe the optimizations and transformations that the compiler made while compiling your program.

The Oracle Solaris Studio compilers provide significantly more information to help you understand your code than other compilers. With optimization, the compilers insert

commentary describing the transformations performed on the code, any obstacles to parallelization, operation counts for loop iterations, and so forth. The compiler commentary can be displayed in tools such as the Performance Analyzer.

C Compiler

The Oracle Solaris Studio C compiler conforms to the *ISO/IEC 9899:1999, Programming Language - C* and *ISO/IEC 9899:1990, Programming Languages-C* standards. The C compiler also supports the OpenMP 3.1 shared-memory parallelism API.

The C compilation system consists of a compiler, an assembler, and a linker. The `cc` command invokes each of these components automatically unless you use command-line options to perform the steps separately.

`cc` Command Syntax

The syntax of the `cc` command is:

`cc [`*compiler-options*`] `*source-files*` [-l`*library*`]...`

The compiler options precede the source file names. You can type `cc -flags` to see short descriptions of all the possible compiler options.

The source file names can end in `.c`, `.s`, `.S`, or `.i`. Files whose names do not end in one of these suffixes are passed to the link editor.

Following the source file names, you can optionally specify the `-l`*library* option to add object libraries to the linker's list of search libraries.

The link editor produces a dynamically linked executable named `a.out` by default. You can use the `-o` *filename* option to specify a different executable name. You can use the `-c` option to compile a source file and produce an object (`.o`) file but suppress linking.

To compile a source file named `test.c` and produce an executable file named `a.out`:

```
% cc test.c
```

To compile source files `test1.c` and `test2.c` and link them into an executable file called `test`:

```
% cc -o test test1.c test2.c
```

To compile the two source files separately and then link them into an executable:

```
% cc -c test1.c
% cc -c test2.c
% cc test1.o test2.o
```

C Documentation

For complete information about using the C compiler, and the `cc` command and its options, see the *Oracle Solaris Studio 12.3: C User's Guide* and the `cc`(1) man page. For information about the new and changed features, software corrections, problems and workarounds, and limitations and incompatibilities of the compiler, see *What's New in the Oracle Solaris Studio 12.3 Release*.

C++ Compiler

The Oracle Solaris Studio C++ compiler (`CC`) supports the *ISO International Standard for C++, ISO IS 14822:2003, Programming Language — C++*. The `CC` compiler also supports the OpenMP 3.1 shared-memory parallelism API. The OpenMP 3.1 API is included with Oracle Solaris Studio 12.3.

The C++ compiler consists of a front end, optimizer, code generator, assembler, template prelinker, and link editor. The `CC` command invokes each of these components automatically unless you use command-line options to specify otherwise.

`CC` Command Syntax

The syntax of the `CC` command is:

`CC [`*compiler-options*`] `*source-files*` [-l`*library*`]...`

The compiler options precede the source file names. You can type `CC -flags` to see short descriptions of all the possible `CC` compiler options.

The source file names can end in `.c`, `.C`, `.cc`, `.cxx`, `.c++`, `.cpp`, or `.i`. Files whose names do not end with one of these suffixes are treated as object files or libraries and are handed over to the link editor.

Following the source file names, you can optionally specify the `-l`*library* option to add object libraries to the linker's list of search libraries.

By default, the files are compiled and linked in the order given to produce an output file named `a.out`. You can use the `-o` *filename* option to specify a different executable name. You can use the `-c` option to compile a source file and produce an object (`.o`) file, but suppress linking.

To compile a source file named `test.C` and produce an executable file named `a.out`:

```
% CC test.c
```

To compile the two source files `test1.c` and `test2.C` separately and then link them into an executable file called `test`:

```
% CC -c test1.c
% CC -c test2.C
% CC -o test test1.o test2.o
```

C++ Documentation

For complete information about using the C++ compiler , and the `CC` command and its options, see the *Oracle Solaris Studio 12.3: C++ User's Guide* and the `CC`(1) man page. For information about the new and changed features, software corrections, problems and workarounds, and limitations and incompatibilities of the compiler, see *What's New in the Oracle Solaris Studio 12.3 Release*.

Fortran 95 Compiler

The Fortran compiler in Oracle Solaris Studio is optimized for Oracle Solaris on multiprocessor systems. The compiler can perform both automatic and explicit loop parallelization to enable your programs to run efficiently on multiprocessor systems.

The Fortran compiler offers compatibility with Fortran77, Fortran90, and Fortran95 standards. and support of OpenMP 3.1.

The `f95` command invokes the Oracle Solaris Studio Fortran compiler.

`f95` Command Syntax

The syntax of the `f95` command is:

`f95 [`*compiler-options*`] `*source-files*`... [-l`*library*`]`

The compiler options precede the source file names. You can type `f95 -flags` to see short descriptions of all the possible compiler options.

The source file names must be one or more Fortran source file names ending in `.f`, `.F`, `.f90`, `.f95`, `.F90`, `.F95`, or `.for`.

Following the source file names, you can optionally specify the `-l`*library* option to add object libraries to the linker's list of search libraries.

A sample command to compile a Fortran program from two source files:

```
% f95 -o hello_1 foo.f bar.f
```

To compile the same program with separate compile and link steps:

```
% f95 -c -o bar.o bar.f
% f95 -c  -o foo.o foo.f
% f95 -o hello_1 bar.o foo.o
```

To compile the same program and link in a library called `libexample`:

```
% f95 -o hello_1 foo.f bar.f -lexample
```

Fortran Documentation

For complete information about using the Fortran 95 compiler, and a description of the `f95` command and its options, see the *Oracle Solaris Studio 12.3: Fortran User's Guide* and the `f95`(1) man page. For information about the new and changed features, software corrections, problems and workarounds, and limitations and incompatibilities of the compiler, see *What's New in the Oracle Solaris Studio 12.3 Release*.

C/C++/Fortran Libraries

Oracle Solaris Studio compilers make use of the operating system's native libraries. The Oracle Solaris operating system provides many system libraries installed in `/usr/lib`, including the C runtime `libc` and C++ runtime `libCrun` libraries. The `intro`(3) man page describes each library and refers to additional man pages for detailed information about each library. Type `man intro.3` to view the page.

Most `/usr/lib` libraries have a C interface. The `libc` and `libm` libraries are linked by the `cc` and `CC` compiling systems by default, so you do not need to link to them explicitly. To link any other `/usr/lib` system library, use the appropriate `-l` option with the compiler. For example, to link the `libmalloc` library, specify `-lmalloc` on the `cc` and `CC` command line at link time.

Fortran runtime libraries are provided with Oracle Solaris Studio, not with the operating systems.

Fortran programs can also use the Oracle Solaris `/usr/lib` libraries that have a C interface. See the *Fortran Programming Guide* for information about the C-Fortran interface.

See the *Linker and Libraries Guide* in the Oracle Solaris documentation for more information about linking libraries.

OpenMP 3.1 for Parallel Programming

The Oracle Solaris Studio compilers conform to the OpenMP 3.1 shared memory parallelization API specification. OpenMP consists of a set of compiler directives, library routines, and environment variables that you can use to develop multithreaded applications.

To take advantage of the compiler OpenMP support, use OpenMP directives and functions to parallelize sections of your code, and use the `-xopenmp` option when compiling. To run a parallelized program in a multithreaded environment, you must set the `OMP_NUM_THREADS` environment variable prior to execution to specify the number of threads the program can use. The default is 2 if `OMP_NUM_THREADS` is not specified. See the *Oracle Solaris Studio 12.3: OpenMP API User's Guide* for details.

Sun Performance Library for Programs With Intensive Computation

Sun Performance Library is a set of optimized, high-speed mathematical subroutines for solving linear algebra and other numerically intensive problems. Sun Performance Library is based on a collection of public domain subroutines available from Netlib at `http://www.netlib.org`. Sun enhanced these public domain subroutines and bundled them as the Sun Performance Library.

Sun Performance Library routines can increase application performance on both serial and multiprocessor (MP) platforms, because the serial speed of many Sun Performance Library routines has been increased, and many routines have been parallelized. Sun Performance Library routines also have SPARC and AMD specific optimizations that are not present in the base Netlib libraries, including extension that support Fortran 95 and C language interfaces.

The Sun Performance Library is linked into an application with the `-library` switch instead of the `-l` switch that is used to link other libraries.

To compile Fortran source that uses Performance Library routines:

```
% f95 -dalign filename.f -library=sunperf
```

The `-dalign` option is required because this option was used to compile the Performance Library to control the alignment of data.

To compile C or C++ source that uses Performance Library routines:

```
% cc filename.c -library=sunperf
% CC filename.cpp -library=sunperf
```

To compile and link statically so that you can deploy the application to a system that does not have the Sun Performance Library, you must use the options `-library=sunperf` and `-staticlib=sunperf`.

For complete information about using the Sun Performance Library, see the *Oracle Solaris Studio 12.2: Sun Performance Library User's Guide*. For man pages for each function and subroutine in the library, see section 3p of the Oracle Solaris Studio man pages. For new and changed features, software corrections, known problems, limitations, and incompatibilities of the current release of the Sun Performance Library, see *What's New in the Oracle Solaris Studio 12.3 Release*.

dmake Utility for Building Applications

The `dmake` utility is a command-line tool, compatible with `make`(1), for building software project targets that are defined in makefiles. `dmake` can build targets in grid, distributed, parallel, or serial mode. If you use the standard `make`(1) utility, the transition to `dmake` requires little if any alteration to your makefiles. `dmake` is a superset of the `make` utility.

`dmake` parses your makefiles to determine which targets can be built concurrently, and distributes the build of those targets over a number of hosts that you specify in a `.dmakerc` file.

The Oracle Solaris Studio IDE uses `dmake` by default when you build and run your project in the IDE, using the targets in the makefile for the project. You can also execute individual makefile targets with `dmake` through the IDE. If you prefer you can configure the IDE to use `make` instead.

For information about how to use `dmake` from the command line and how to create your `.dmakerc` file, see *Oracle Solaris Studio 12.3: Distributed Make (dmake)* or the `dmake`(1) man page. For a list of the new and changed features, software corrections, known problems, limitations, and incompatibilities in the current release of `dmake`, see *What's New in the Oracle Solaris Studio 12.3 Release*.

Tools for Debugging Applications

Oracle Solaris Studio includes the `dbx` debugger to help you detect errors in your applications.

`dbx` is an interactive, source-level, command-line debugging tool. You can use it to run a C, C++, or Fortran program in a controlled manner and to inspect the state of a stopped program. `dbx` gives you complete control of the dynamic execution of a program, including collecting performance and memory usage data, monitoring memory access, and detecting memory leaks.

dbx enables you to perform the following tasks:

- Examine a core file from a program that has crashed
- Set breakpoints
- Step through your program
- Examine the call stack
- Evaluate variables and expressions
- Use runtime checking to find memory access problems and memory leaks
- Use fix-and-continue to modify and recompile a source file and continue executing without rebuilding the entire program

You can use the dbx debugger on the command line, graphically through the Oracle Solaris Studio IDE, or through a separate graphical interface called dbxtool.

For more information about using dbx in the different user interfaces, see the following sections:

- "dbx on the Command Line" on page 21
- "dbx in the IDE" on page 22
- "dbx in dbxtool" on page 23

dbx on the Command Line

The basic syntax of the dbx command to start dbx is:

```
dbx [options] [program-name|-] [process-ID]
```

To start a dbx session and load the program test to be debugged:

```
% dbx test
```

To start a dbx session and attach it to a program that is already running with the process ID 832:

```
% dbx - 832
```

When your dbx session starts, dbx loads the program information for the program you are debugging. Then dbx waits in a ready state visiting the main block of the program such as the main() function in a C or C++ program. The (dbx) command prompt is displayed.

You can type commands at the (dbx) prompt. Typically, you first set a breakpoint by typing a command such as stop in main and then type a run command to run your program:

```
(dbx) stop in main
(4) stop in main
```

```
(dbx) run
Running: quote_1
(process id 5685)
(dbx)
```

When execution stops at the breakpoint, you can type commands such as `step` and `next` to single-step through your code, and `print` and `display` to evaluate expressions and variables.

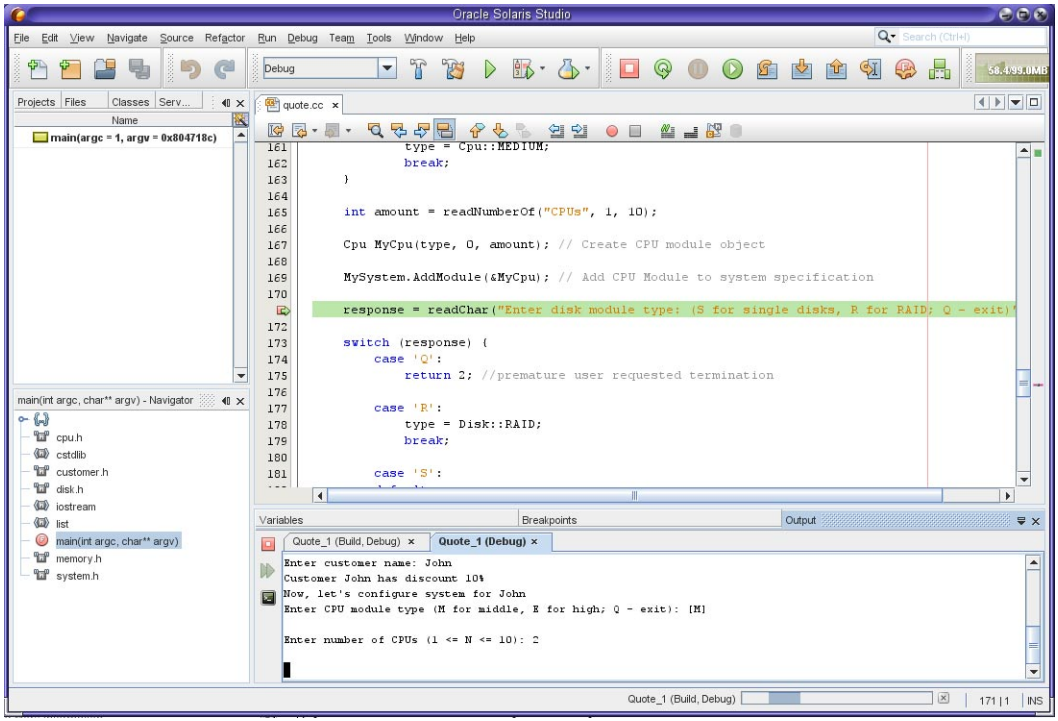
For information about the command-line options for the `dbx` utility, see the `dbx`(1) man page.

For complete information about using `dbx` including a command reference section, see *Oracle Solaris Studio 12.3: Debugging a Program With dbx*. You can also learn about the `dbx` commands and other topics by typing `help` at the `(dbx)` command line. For a list of the new and changed features, software corrections, known problems, limitations, and incompatibilities in the current release of `dbx`, see *What's New in the Oracle Solaris Studio 12.3 Release*.

`dbx` in the IDE

You can use `dbx` in the Oracle Solaris Studio IDE by opening your project, creating breakpoints in the source, and clicking the Debug button. The IDE enables you to use menu options and buttons to step through your program, and provides a complete set of debugging windows.

As with building your application, the IDE debugs your application as a project. In the following screen capture, one of the IDE sample projects is running in `dbx`. You can use commands in the Debug menu or the buttons at the top right in the IDE window to control the debugger. As you use the Debug commands and buttons, the IDE issues commands to `dbx` and displays output in the various debugging windows.



In the figure , the debugger is stopped at a breakpoint and the Output window shows the program interaction. Some debugger windows such as Variables and Breakpoints are also shown but not selected. You can open more debugging windows by selecting from the Window → Debugging menu. One of the debugging windows is the Debugger Console window, which displays the interaction with `dbx`. You can also type commands at the `(dbx)` prompt in the Debugger Console window.

For more information about using `dbx` in the IDE, see the integrated help in the IDE and Oracle Solaris Studio 12.3: IDE Quick Start Tutorial.

`dbx` in `dbxtool`

You can also use `dbx` through `dbxtool`, a graphical user interface that is separate from the IDE, but includes similar debugging windows and an editor. Unlike the IDE, `dbxtool` does not use projects, and you can use it to debug any C, C++, or Fortran executable or core file.

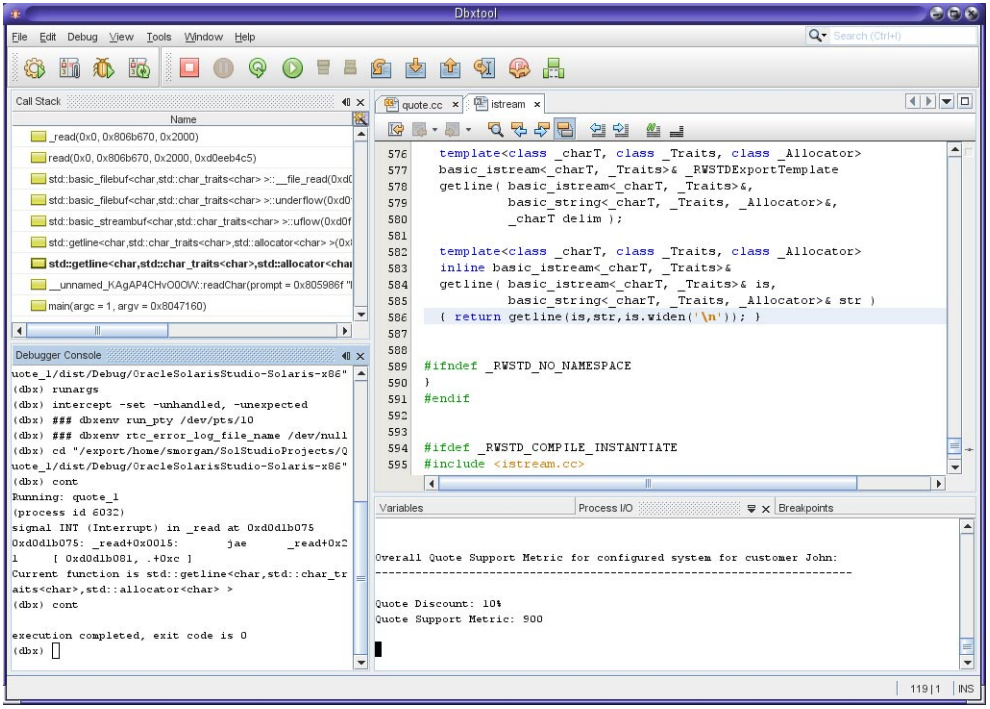
To start `dbxtool`, type:

```
% dbxtool executable-name
```

You can also omit the executable name and specify it from within `dbxtool` instead.

As with the IDE, you can issue commands to `dbx` by clicking toolbar buttons or using Debug menu options in `dbxtool`. You can also type commands at the `(dbx)` prompt in the Debugger Console window.

In the following figure, `dbx` is running in `dbxtool` on the `quote_1` program. The Debugger Console window is selected and you can see the `(dbx)` prompt and commands that have been entered by `dbxtool` in response to the user's selections.



For information about using `dbxtool`, see the `dbxtool`(1) man page and the integrated help in `dbxtool`. The Oracle Solaris Studio 12.3: dbxtool Tutorial shows how to use `dbxtool`.

Tools for Verifying Applications

Oracle Solaris Studio provides tools to help verify your application's stability. The following tools combine dynamic, static and code coverage analysis to detect application vulnerabilities, including memory leaks and memory access violations.

- **Discover**. A command-line utility that helps detect memory access errors in your code.
- **Uncover**. A command-line utility that shows you which areas of your application code are not covered by testing.
- **Code Analyzer**. A graphical tool that analyzes and displays static code error data collected by the C or C++ compiler, and data collected by Discover and Uncover. By integrating static error data with dynamic memory access error data and code coverage data, the Code Analyzer lets you find errors in your application that you would not find when using other error detection tools by themselves.

Discover Tool for Detecting Memory Errors

The Memory Error Discovery Tool (Discover) is an advanced development tool for detecting memory access errors in your programs. Compiling a binary with `-g` enables Discover to display source code and line number information while reporting errors and warnings.

Discover is simple to use. After compiling your binary with the `-g` option, you run the `discover` command on the binary to instrument it. Then you run the instrumented binary to create a Discover report. You can request the Discover report in HTML format, text format, or both. The report shows memory errors, warnings, and memory leaks, and you can display the source code and stack trace for each error or warning.

The following example from the `discover`(1) man page shows how to prepare, instrument, and run an executable to generate a Discover report for detecting memory access errors. The `-w` option on the `discover` command line indicates the report should be written as text and the `-` option indicates that the output should go to the screen.

```
% cc -g -O2 test.c -o test.prep
% discover -w - -o test.disc test.prep
% ./test.disc
ERROR (UMR): accessing uninitialized data from address 0x5000c (4 bytes) at:
     foo() + 0xdc  <ui.c:6>
          3:    int *t;
          4:    foo() {
          5:     t = malloc(5*sizeof(int));
          6:=>   printf("%d0, t[1]);
          7:    }
          8:
          9:    main()
     main() + 0x1c
     _start() + 0x108
```

```
    block at 0x50008 (20 bytes long) was allocated at:
     malloc() + 0x260
     foo() + 0x24  <ui.c:5>
          2:
          3:    int *t;
          4:    foo() {
          5:=>   t = malloc(5*sizeof(int));
          6:     printf("%d0, t[1]);
          7:    }
          8:
     main() + 0x1c
     _start() + 0x108

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

1 block at 1 location left allocated on heap with a total size of 20 bytes

    1 block with total size of 20 bytes
     malloc() + 0x260
     foo() + 0x24  <ui.c:5>
          2:
          3:    int *t;
          4:    foo() {
          5:=>   t = malloc(5*sizeof(int));
          6:     printf("%d0, t[1]);
          7:    }
          8:
     main() + 0x1c
     _start() + 0x108
```

For more information, see the `discover`(1) man page and the *Oracle Solaris Studio 12.3: Discover and Uncover User's Guide*.

Uncover Tool for Measuring Code Coverage

Uncover is a command-line tool for measuring code coverage. The tool shows you which areas of your application code are exercised when the application is run, and which are not exercised and not covered by testing. Uncover produces a report with statistics and metrics to help you determine which functions should be added to the test suite to ensure that more of the code is covered during testing.

Uncover works with any binary that is built with an Oracle Solaris Studio compiler, and works best when the binary is built without optimization. Compiling a binary with `-g` enables Uncover to display source code and line-number information while reporting on code coverage.

After compiling the binary, you run the `uncover` command on the binary. Uncover creates a new binary with added instrumentation code and also creates a directory named *binary*`.uc` that will contain the code coverage data for your program. Each time you run the instrumented binary, code coverage data is collected and stored in the *binary*`.uc` directory.

You can display the experiment data in the Performance Analyzer, or generate the Uncover report as HTML and display it in your web browser.

The following example shows how to prepare, instrument, and run an executable to generate an Uncover report for examining code coverage. The optimized binary is `test` and is replaced by the instrumented binary also named `test`.

```
% cc -g -O2 test.c -o test
% uncover test
% test
```

The experiment directory is `test.uc` and contains the data that is generated when the instrumented `test` runs. The `test.uc` directory also contains a copy of the uninstrumented `test` binary.

To view the experiment in Performance Analyzer:

```
% uncover test.uc
```

To view the experiment in an HTML page in a browser:

```
% uncover -H test.html test.uc
```

For more information, see the `uncover`(1) man page and the *Oracle Solaris Studio 12.3: Discover and Uncover User's Guide*.

Code Analyzer Tool For Integrated Error Checking

The Oracle Solaris Studio Code Analyzer is a graphical tool that enables you do an integrated analysis of your code. The Code Analyzer uses three types of information that you gather using other tools:

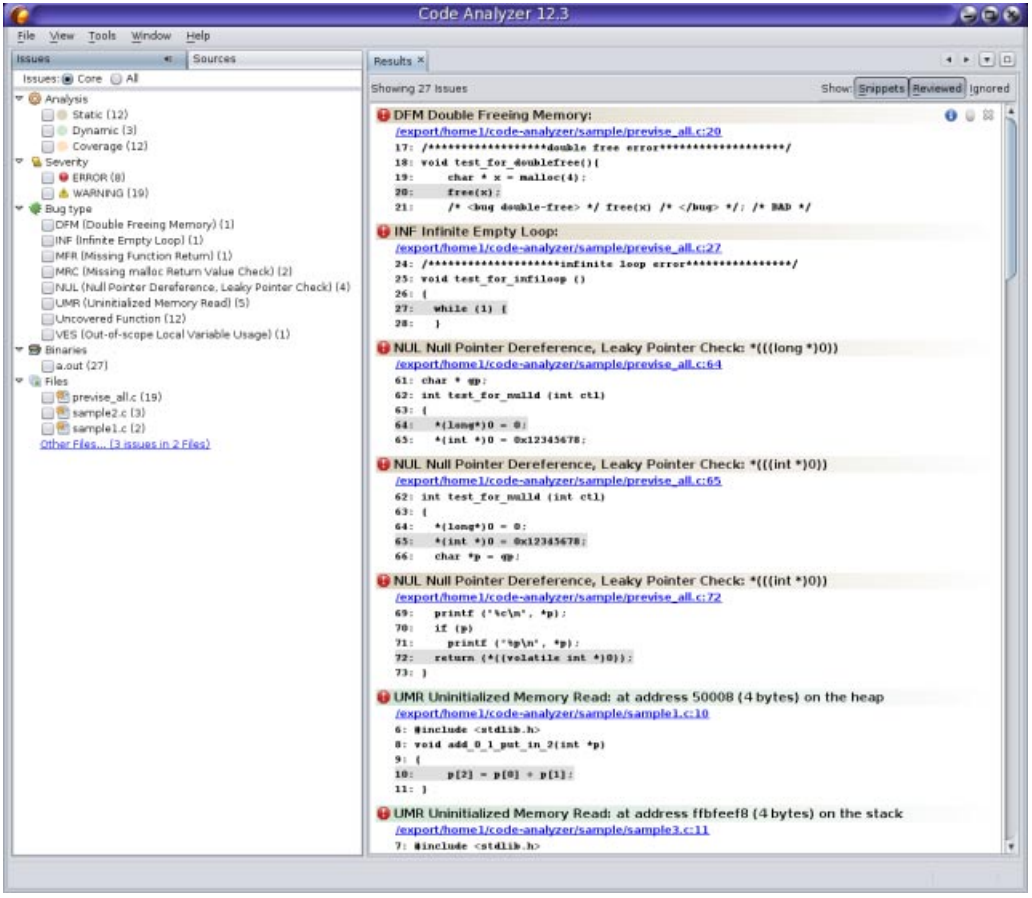
- Static code checking, which is performed when you compile your application with the Oracle Solaris Studio C or C++ compiler and specify the `-xanalyze=code` option.
- Dynamic memory access checking, which is performed when you instrument your binary with Discover using the `-a` option, and then run the instrumented binary.
- Code coverage checking, which is performed when you instrument your binary with Uncover, run the instrumented binary, and then run Uncover with the `-a` option on the collected coverage data.

You can use the Code Analyzer on a binary that has been prepared with any one of these tools or any combination of these tools. However, the integrated view of the three types of data offers the most revealing look into your code and enables you to create a more secure and robust application.

The following example shows how to run the Code Analyzer on a binary named `a.out` that has been prepared previously with Discover and Uncover.

```
% code-analyzer a.out
```

In the following figure, the Code Analyzer is displaying the issues found in the `a.out` binary.



Tools for Tuning Application Performance

Oracle Solaris Studio software features several tools you can use to examine your application's behavior, enabling you to tune its performance.

The performance tools include the following:

- **Performance Analyzer and associated tools.** A set of advanced performance tools and utilities to help you identify locations in your code where problems affect performance.
- **Simple Performance Optimization Tool (SPOT).** A command-line tool that works with the Performance Analyzer tools and produces web pages to report the data gathered by the tools.
- **Profiling Tools in DLight.** A set of graphical tools that run simultaneously, enabling you to analyze data about a running application from multiple sources in a synchronized fashion.
- **Profiling Tools in the IDE.** Profiling tools similar to those in DLight, enabling you to examine the performance of your projects from within the IDE.

Performance Analyzer Tools

The Oracle Solaris Studio software provides a set of advanced performance tools and utilities that work together. The Collector, the Performance Analyzer, the Thread Analyzer, and the `er_print` utility help you assess the performance of your code, identify potential performance problems, and locate the part of the code where the problems occur. These tools together are referred to as the Performance Analyzer tools.

You can use options for the Oracle Solaris Studio C, C++, and Fortran compilers to target hardware and advanced optimization techniques that will improve your program's performance. The Performance Analyzer tools also are engineered for use on Oracle Sun hardware together with the compilers, and can help you improve your program's performance when running on Oracle Sun machines.

Compared to the DLight profiling tools, the Performance Analyzer tools allow you to have greater control over the data that is collected, inspect the data more deeply, and examine your program's interaction with the hardware. The Performance Analyzer tools are designed for and tested with complex compute-intensive applications running on current Oracle Sun hardware.

The Performance Analyzer tools also feature profiling of OpenMP parallel applications and MPI-based distributed applications, to help you to determine if you are using these technologies effectively in your application.

To use the Performance Analyzer tools, you must perform two steps:

1. Collect performance data with the Collector.

2. Examine the data with the Performance Analyzer graphical tool, the `er_print` command line utility, or the Thread Analyzer to detect data races and deadlocks on multithreaded applications.

Collect Performance Data With the Collector

The Collector collects performance data using profiling and by tracing function calls. The data can include call stacks, microstate accounting information (on Oracle Solaris platforms only), thread synchronization delay data, hardware counter overflow data, Message Passing Interface (MPI) function call data, memory allocation data, and summary information for the operating system and the process. The Collector can collect all types of data for C, C++, and Fortran programs, and profiling data for applications written in the Java programming language. You can run the Collector using the `collect` command, or from the Performance Analyzer, or by using the `dbx` debugger's `collect` subcommand.

The Oracle Solaris Studio IDE profiling tools also use the Collector to gather information.

To collect data with the `collect` command:

```
% collect [collect-options] executable executable-options
```

You can include options to the `collect` command to specify the type of data you want to collect. For example, the `-c on` option causes the Collector to record instruction counts. You can pass arguments to the target executable by specifying the arguments after the executable.

The Collector creates a data directory with the name `test.1.er` by default, but you can specify a different name on the command line. The `test.1.er` directory is known as an experiment, and the name must always end in `.er` in order for the tools to recognize it as an experiment.

The following command shows how to use `collect` on the `synprog` program:

```
% collect synprog
Creating experiment database test.1.er ...
00:00:00.000  ===== (15909) synprog run
00:00:00.002  ===== (15909) Thu  10 Nov 11  15:12:18 Stopwatch calibration
  OS release 5.10 -- enabling microstate accounting 5.10.
        0.001498 s.  (32.8 % of 0.004568 s.) -- inner
        N = 1000, avg = 1.498 us., min = 0.721, max = 596.665
        0.003482 s.  (72.9 % of 0.004776 s.) -- outer
        N = 1000, avg = 3.482 us., min = 2.883, max = 599.007
00:00:00.007  ===== (15909)  Begin commandline
        icpu.md.cpu.rec.recd.dousl.gpf.fitos.uf.ec.tco.b.nap.sig.sys.so.sx.so
00:00:00.008  ===== (15909) start of icputime
    3.019055 wall-secs.,   2.328491 CPU-secs., in icputime
00:00:03.027  ===== (15909) start of muldiv
    3.012635 wall-secs.,   2.675769 CPU-secs., in muldiv
00:00:06.040  ===== (15909) start of cputime
    3.000567 wall-secs.,   2.591964 CPU-secs., in cputime
00:00:09.041  ===== (15909) start of recurse
    ...
```

(output edited to conserve space)
...

The data is stored in the `test.1.er` directory, which can be viewed using Performance Analyzer or `er_print`.

For information about using the Collector, see the Help menu in the Performance Analyzer, the *Oracle Solaris Studio 12.3: Performance Analyzer* manual, and the `collect`(1) man page.

Examine Performance Data With the Performance Analyzer

The Performance Analyzer is a graphical user interface (GUI) that displays metrics for the data recorded by the Collector. These metrics are:

- Clock profiling metrics, which tell you where your program spent time in several categories.
- Hardware counter metrics, which show information about CPU-specific events experienced by your program.
- Synchronization delay metrics, which show delays in the synchronization of tasks performed by different threads of a multithreaded program.
- Memory allocation metrics, which shows memory leaks in your program.
- MPI tracing metrics, which can help you identify places where your MPI program has a performance problem due to MPI calls.

You can run the Performance Analyzer with the `analyzer` command. The basic syntax of the `analyzer` command to start the Performance Analyzer is:

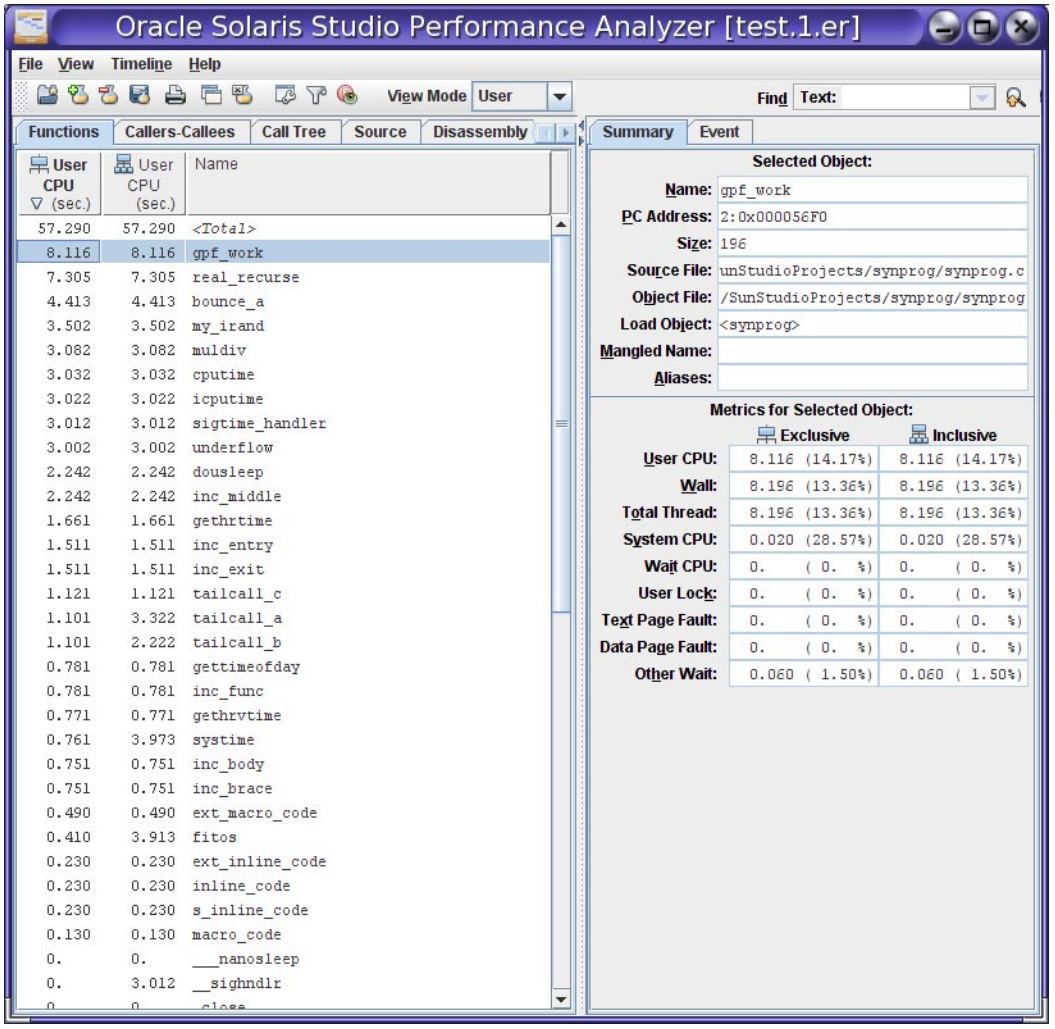
```
% analyzer [experiment-list]
```

The *experiment-list* is one or more file names of experiments that were collected with the Collector. If you want to load more than one experiment, specify the names separated by spaces. When invoked on more than one experiment, the Analyzer aggregates the experiment data by default, but can also be used to compare the experiments.

To open the experiment `test.1.er` in Performance Analyzer:

```
% analyzer test.1.er
```

The following figure shows the Performance Analyzer's Functions tab for a `test.1.er` experiment that was made on the `synprog` example. The Functions tab shows the CPU time used by each function of the `synprog` program. When you click the function `gpf_work` the Summary tab on the right side shows details about the `gpf_work` function's resource usage.



For information about using the Performance Analyzer, see the *Oracle Solaris Studio 12.3: Performance Analyzer* manual, the Performance Analyzer integrated help, and the `analyzer`(1) man page.

Examine Performance Data With the `er_print` Utility

The `er_print` utility presents in plain text most of the displays that are presented in the Performance Analyzer except the Timeline display, the MPI Timeline display, and the MPI Chart display.

You can use the `er_print` utility to display the performance metrics for functions, callers and callees, the call tree, source code listing, disassembly listing, sampling information, dataspace data, thread analysis data, and execution statistics.

The general syntax of the `er_print` command is:

```
% er_print -command experiment-list
```

You can specify one or more commands to indicate the type of data you want to display. The *experiment-list* is one or more file names of experiments that were collected with the Collector. When invoked on more than one experiment, `er_print` aggregates the experiment data by default, but can also be used to compare the experiments.

The following example shows the command for displaying function information for a program. The output shown is for the same experiment that was used in the screen capture of Performance Analyzer in the previous section of this document.

```
%  er_print -functions test.1.er
Functions sorted by metric: Exclusive User CPU Time

Excl.     Incl.      Name
User CPU  User CPU
  sec.      sec.
57.290    57.290     <Total>
 8.116     8.116     gpf_work
 7.305     7.305     real_recurse
 4.413     4.413     bounce_a
 3.502     3.502     my_irand
 3.082     3.082     muldiv
 3.032     3.032     cputime
 3.022     3.022     icputime
 3.012     3.012     sigtime_handler
 3.002     3.002     underflow
 2.242     2.242     dousleep
 2.242     2.242     inc_middle
 1.661     1.661     gethrtime
 1.511     1.511     inc_entry
 1.511     1.511     inc_exit
 1.121     1.121     tailcall_c
 1.101     3.322     tailcall_a
 1.101     2.222     tailcall_b
 0.781     0.781     gettimeofday
 0.781     0.781     inc_func
 0.771     0.771     gethrvtime
 0.761     3.973     systime
 0.751     0.751     inc_body
 0.751     0.751     inc_brace
 0.490     0.490     ext_macro_code
.
.       lines deleted
```

You can also use `er_print` interactively if you specify the experiment name and omit the command when starting `er_print`. You can type commands at an `(er_print)` prompt.

For information about the `er_print` utility, see the *Oracle Solaris Studio 12.3: Performance Analyzer* manual and the `er_print`(1) man page.

Analyze Multithreaded Application Performance With the Thread Analyzer

The Thread Analyzer is a specialized version of the Performance Analyzer for examining multithreaded programs. The Thread Analyzer can detect multithreaded programming errors that cause data races and deadlocks in code that is written using the POSIX thread API, the Solaris thread API, OpenMP directives, or a mix of these.

The Thread Analyzer detects two common threading issues in multithreaded programs:

- Data races, which occur when two threads in a single process access the same shared memory location concurrently and without holding any exclusive locks, and at least one of the accesses is a write.
- Deadlocks, which occur when two or more threads are blocked because they are waiting for each other to complete a task.

The Thread Analyzer is streamlined for multithreaded program analysis and shows only the Races, Deadlocks, Dual Source, Race Details, and Deadlock Details tabs of the Performance Analyzer. For OpenMP programs, the OpenMP Parallel Region and OpenMP Task tabs are also shown.

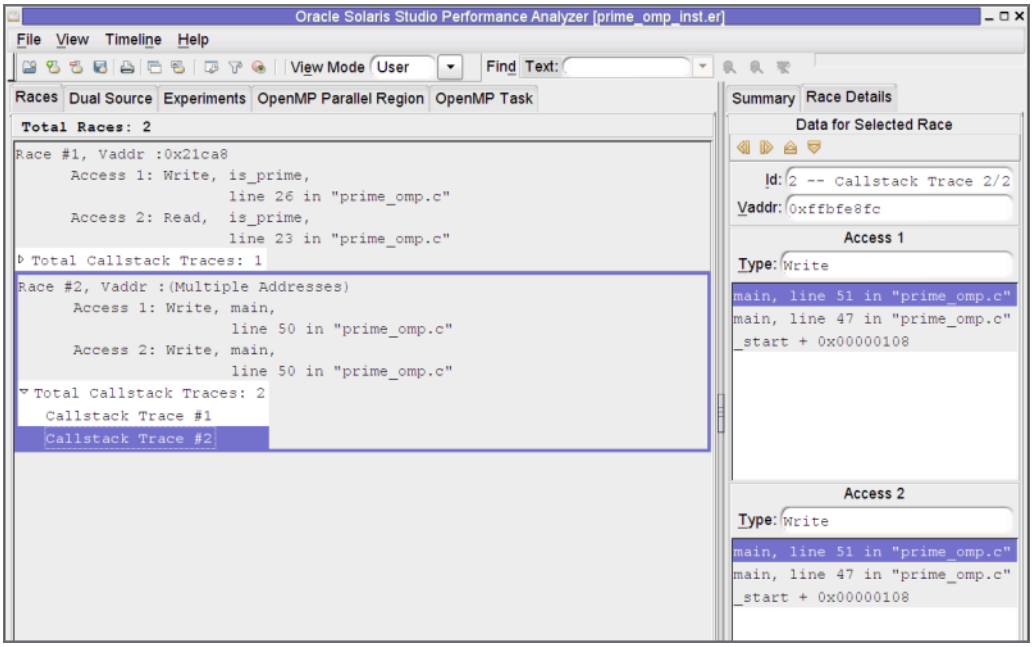
You can detect data races on source code or binary code. In both cases, you have to instrument the code to enable the necessary data to be collected.

To use the Thread Analyzer:

1. Instrument your code for analysis of data races. For source code, use the `-xinstrument=datarace` compiler option when compiling. For binary code, use the `discover -i datarace` command to create instrumented binaries.

   Deadlock detection does not require instrumentation.
2. Run the executable with the `collect` command with the `-r race` option to collect datarace data, the `-r deadlock` option to collect deadlock data, or the `-r all` option to collect both types of data.
3. Start the Thread Analyzer with the `tha` command or use the `er_print` command to display the resulting experiment.

The following figure shows the Thread Analyzer window with data races that were detected in an OpenMP program, and the call stacks that lead to the data races.



For information about using the Thread Analyzer, see the `tha`(1) man page and the *Oracle Solaris Studio 12.3: Thread Analyzer User's Guide*.

Simple Performance Optimization Tool (SPOT)

The Simple Performance Optimization Tool (SPOT) can help you diagnose performance problems in an application. SPOT runs a set of performance tools on an application and produces web pages to report the data gathered by the tools. The tools can also be run independently of SPOT.

SPOT is complementary to the Oracle Solaris Studio Performance Analyzer. The Performance Analyzer tells you where the time was spent in running your application. In certain situations, however, you may need more information to help diagnose your application's problems. SPOT can assist you in these situations.

SPOT uses the Performance Analyzer's `collect` utility as one of its tools. SPOT uses the `er_print` utility and an additional utility called `er_html` to display the profiling data as a web page.

Before you use SPOT, the application binary should be compiled with some level of optimization with the `-O` option and debugging information with the `-g` option to enable the SPOT tools to map performance information to lines of code.

SPOT can be used to gather performance data by launching an application or attaching to an already running application.

To run SPOT and launch your application:

```
% spot executable
```

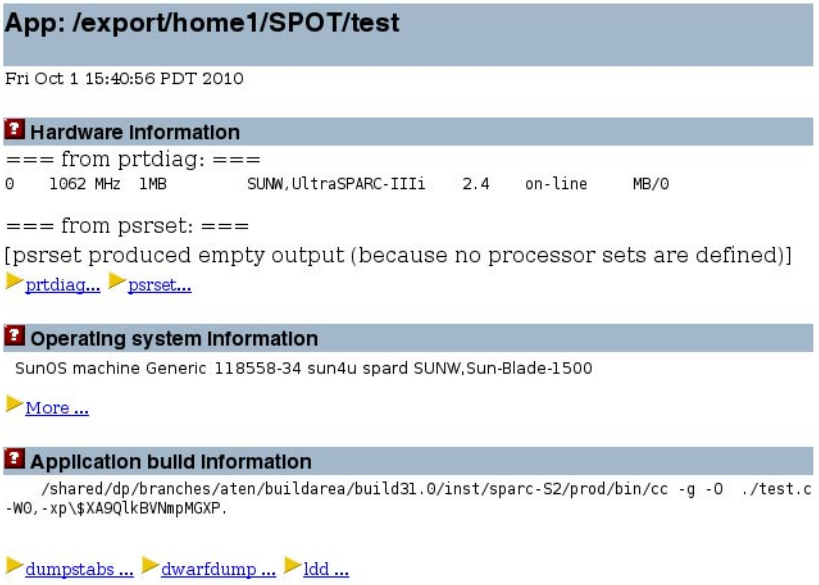
To run SPOT on an already running application:

```
% spot -P process-id
```

SPOT produces a report for each run of your application, as well as a report that compares SPOT data from different runs.

When SPOT is used on a PID, multiple tools are attached to the PID in sequence to generate the report.

The following figure shows part of the SPOT run report, which shows information about the system on which SPOT was run, and about how the application was compiled. The report includes links to other pages with more information.



The SPOT report web pages are linked together to make it easy for you to examine all the data complied.

For more information, see the *Oracle Solaris Studio 12.2: Simple Performance Optimization Tool (SPOT) User's Guide*.

Profiling Tools in DLight

DLight is an interactive graphical tool that uses the Oracle Solaris Dynamic Tracing (DTrace) technology to observe the behavior of running programs. DLight launches multiple profiling tools simultaneously, enabling you to analyze data about a running application from multiple sources in a synchronized fashion. The DLight profiling tools can help you determine the root cause of a runtime problem in an application. The tools are low impact, which enables the profiling to be done without negatively affecting the program or the system.

The DLight profiling tools require privileges that control user access to DTrace features. For this reason, you should run DLight on a system where you either have administrative privileges or can have the `dtrace_user`, `dtrace_proc`, and `dtrace_kernel` privileges granted to you by an administrator.

To start DLight:

```
% dlight
```

You choose a target application that you want to monitor, and the profiling tools that you want DLight to run. The target application can run on the local system, or on a remote networked system where you have login access and DTrace privileges.

You can run DLight on an executable that is not yet running, or attach it to a running process. You can also attach DLight to a process tree, which includes a process and any child processes that are started by the process. DLight graphically displays the data it collects as the target program runs.

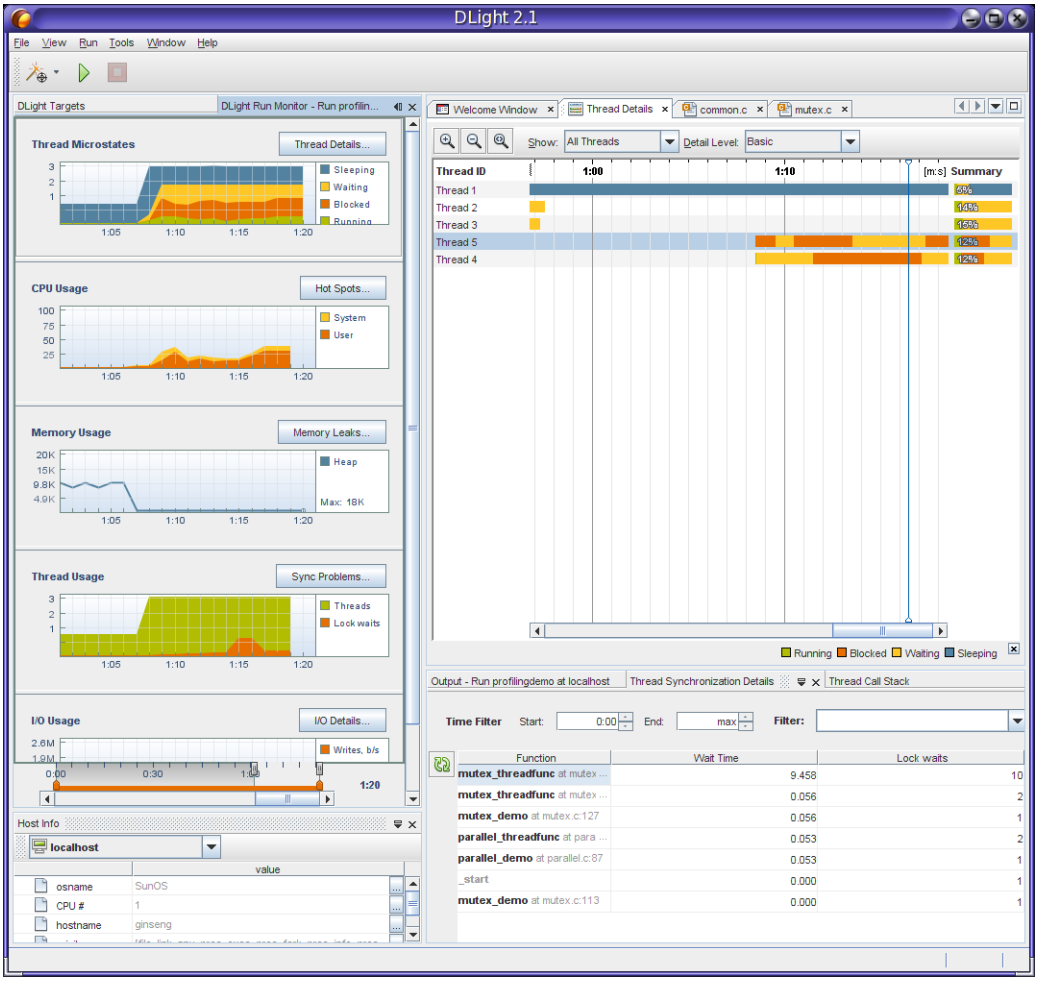
DLight includes the following profiling tools:

- Thread Microstates – Shows summary data about the states of the threads running in your program as it runs.
- CPU Usage – Shows the percentage of CPU time used by your program during its run. The CPU time is divided between user CPU time and system CPU time.
- Memory Usage – Shows how your program's memory heap changes over time.
- Thread Usage – Shows the number of threads running in your program and indicates when the threads are waiting.
- I/O Usage – Shows number of bytes read and written by your program.

Each of the tools provides a button that opens a related tool that shows more detailed information:

- Thread Details – Click the Thread Details button in the Thread Microstates graph
- CPU Time Per Function – Click the Hot Spots button in the CPU Usage graph
- Memory Leaks – Click the Memory Leaks button in the Memory Usage graph
- Thread Synchronization Details – Click the Sync Problems button in the Thread Usage graph
- I/O Details – Click the I/O Details button in the I/O Usage graph

The following figure shows the DLight profiling tools running on the ProfilingDemo sample application that is used in the Oracle Solaris Studio 12.3: DLight Tutorial.

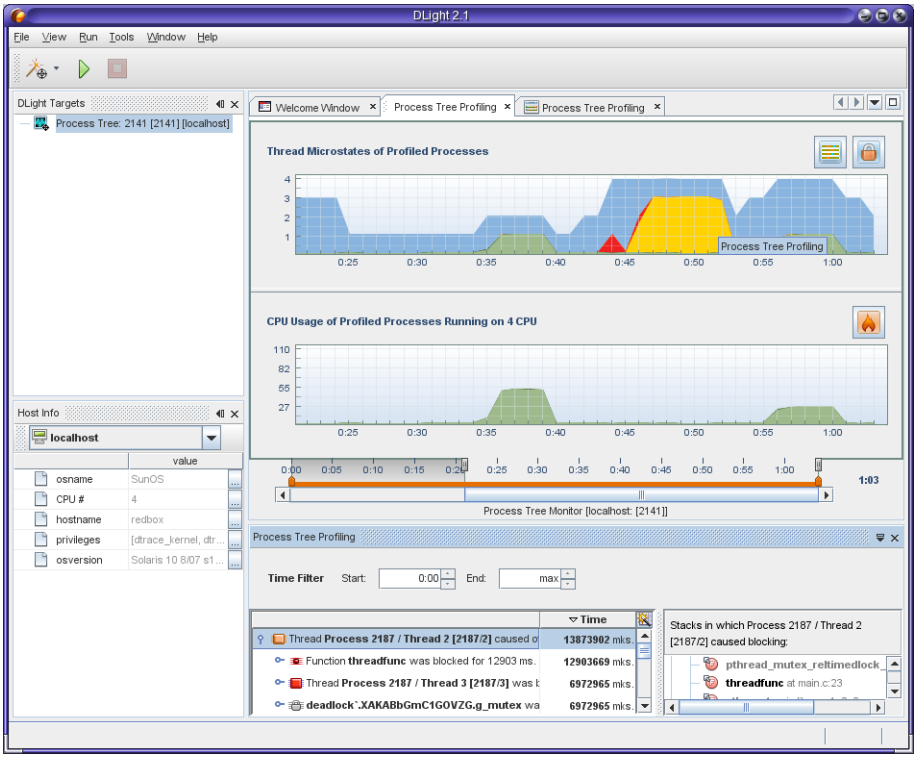


In the figure, the Thread Details window is open at the top left of the DLight window after the user clicked the Thread Details button in the Thread Microstates tool. The Thread Synchronization Details window, shown at the lower right, is open after the user clicked the Sync Problems button in the Thread Usage tool.

When you run DLight on a process tree target the following tools are displayed:

- Thread Microstates of Profiled Processes – Shows an aggregation of the microstates of all the threads of all the processes that DLight profiled with the Process Tree Target. Buttons in this graph open windows for the following details:
- Process Tree Microstate Details – Shows the microstate transitions in the form of a timeline for each thread of the target process and its child processes. Open this tool by clicking a button in the Thread Microstates of Profiled Processes tool.
- Process Tree Blocked Thread Details – Shows locking statistics for the process and its children. Open this tool by clicking a button in the Thread Microstates of Profiled Processes tool.
- CPU Usage of Profiled Processes – Shows the aggregated CPU usage of all threads in all the targeted processes profiled across all the CPUs they are running on.
- Process Tree CPU Hot Spots – Shows the CPU-intensive areas in your program's process tree by displaying the functions in your program along with the CPU time used by the function and any functions it calls. Open this tool by clicking a button in the CPU Usage of Profiled Processes tool.

The following figure shows DLight running on a process tree target.



The figure shows the Process Tree Blocked Thread Details window at the bottom, which the user opened by clicking the lock icon in the Thread Microstates of Profiled Processes tool. On the right side of this window you can see the call stacks where the locks of the selected thread occurred.

The Process Tree Profiling graphs together can be used to determine how your application's multiple processes and multiple threads are working together. You can see points where threads are blocked and the effect on CPU usage, and narrow the problems down to the lines of code where they occur.

For information about using DLight, see the Oracle Solaris Studio 12.3: DLight Tutorial and DLight's integrated help available from the Help menu.

Profiling Tools in the IDE

The IDE provides many of the same profiling tools as DLight to enable you to examine the performance of your projects from within the IDE. The tools run automatically whenever you

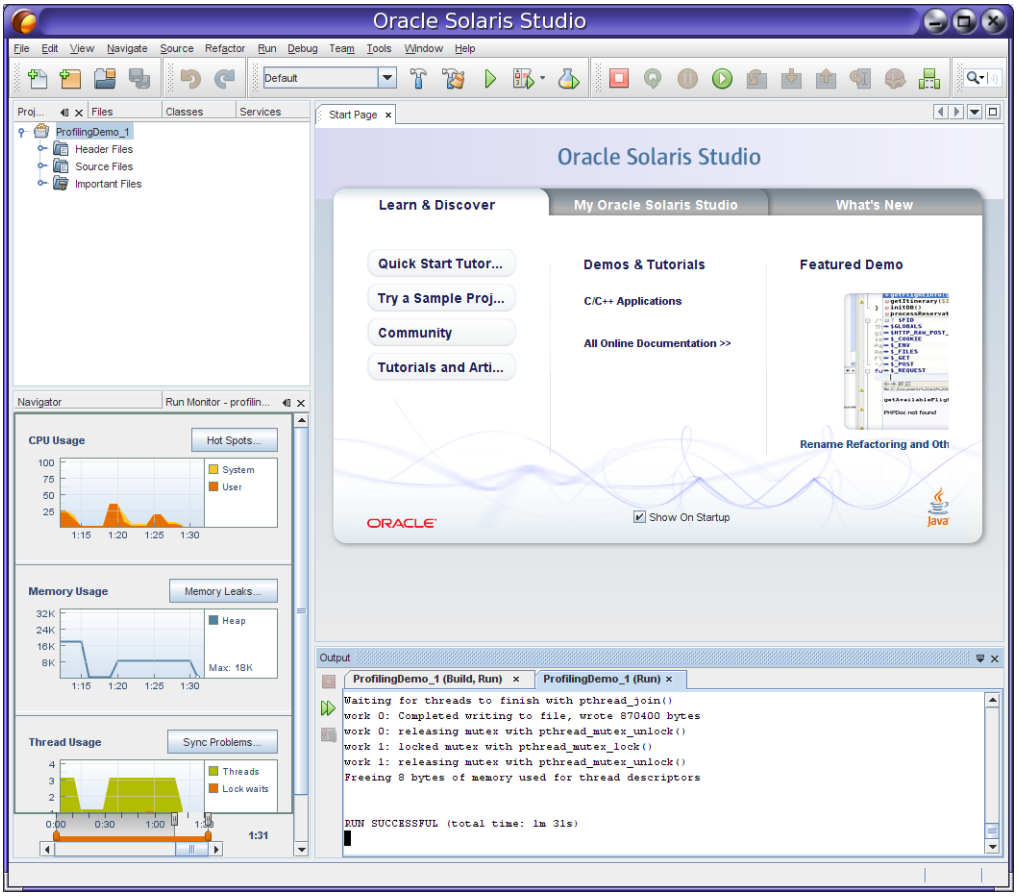
run your C, C++, or Fortran projects. The tools are low impact, which enables the profiling to be done without negatively affecting the program or the system.

The data is presented graphically so you can easily see a summary of resource usage of your program. When you run your project, the Run Monitor window automatically opens to display the output of the low-impact tools. You can disable the profiling tools if you like, or specify which tools you want to run automatically.

The default profiling tools do not use DTrace as the underlying technology. Instead, they use Studio utilities and operating system utilities to collect the data. This approach enables all users to use the tools whether they are running on Oracle Solaris or Linux. However, you can also select tools that use DTrace and provide much more detailed information if you are running the IDE on Oracle Solaris.

As in DLight, the IDE tools that use DTrace require privileges that control a user's access to DTrace features. See the instructions "Enabling DTrace for Profiling C/C++/Fortran Applications" in the IDE help for information about how to assign the privileges.

The following figure shows the IDE with the default Run Monitor tools.

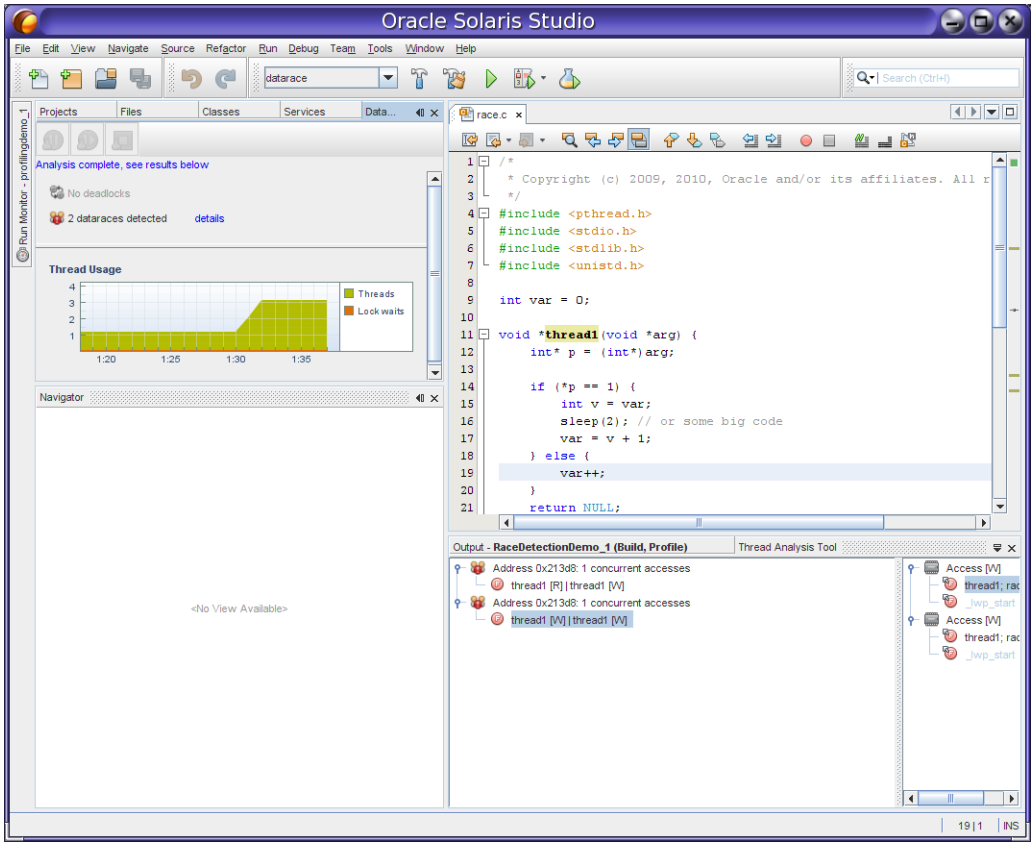


Additional tools for more detailed profiling have a greater performance impact on the system and the application, so those tools do not run automatically. The advanced tools are linked to the automatic profiling tools and can be launched easily by clicking a button.

The IDE features two additional tools that are not available in DLight: the Data Races and Deadlocks Detection tool, and the Memory Access Error tool.

The Data Races and Deadlocks Detection tool uses the same underlying technology as the Thread Analyzer, described later in this document. The tool adds instrumentation to your threaded program and then analyzes the program as it runs to detect actual and potential data races and deadlocks among the threads. To start the tool, click the Profile Project button, select Data Races and/or Deadlocks, specify options for data collection, and click Start.

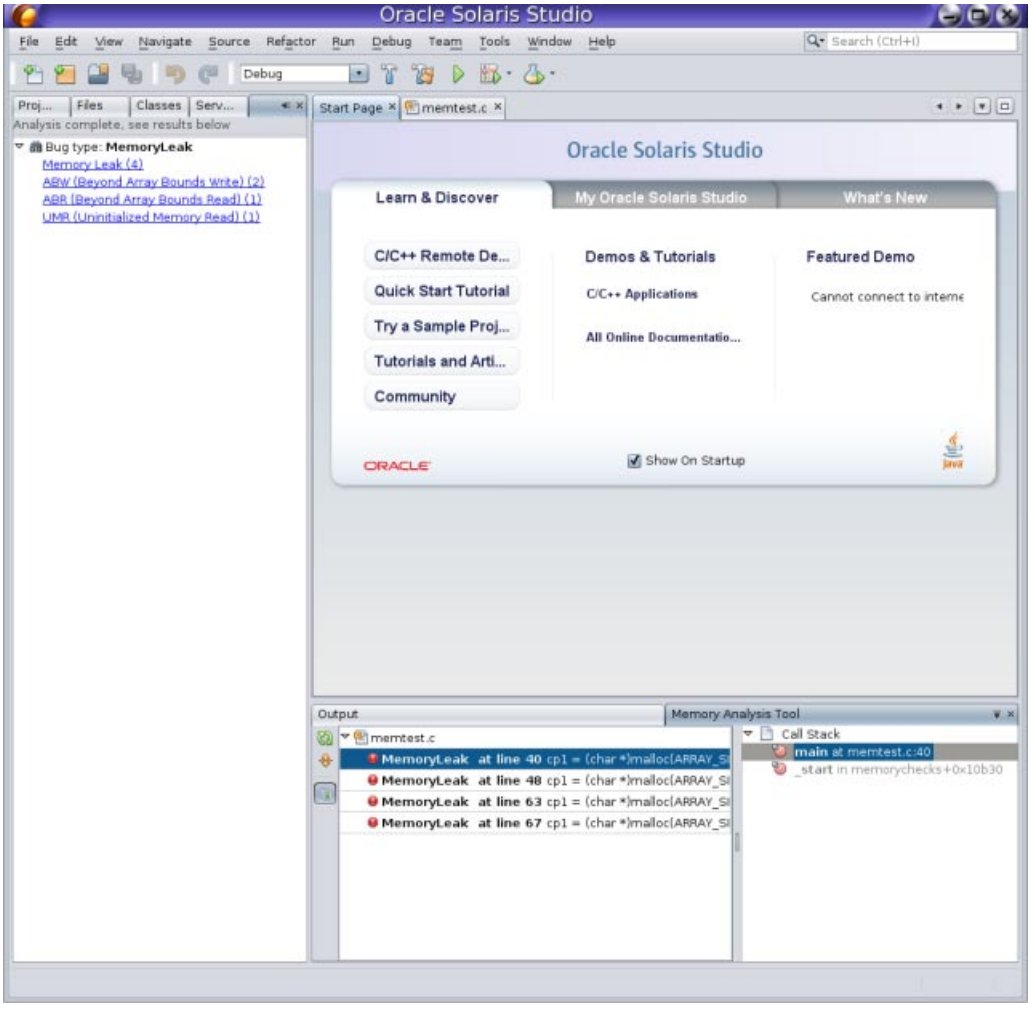
The following figure shows the Data Races and Deadlocks Detection tool after it has detected data races.



If you click the `details` link in the Data Race Detection window, the Thread Details window opens to show where the data races occur. You can double-click the threads in the Thread Details window to open the source file where the problem occurs and go to the affected line of code.

The Memory Access Error tool uses the same underlying technology as Discover, described earlier. The tool instruments your program and then analyzes the program as it runs to detect memory access errors and memory leaks. To start the tool, click the Profile Project button, select Memory Access Error, specify options for data collection, and click Start. The memory access error types are displayed in the Memory Analysis window. When you click on an error type, the errors of that type are displayed in the Memory Analysis Tool window, where you can see the call stack for each error.

The following figure shows the Memory Access Error tool after it has detected memory access errors.



For information about using the profiling tools, see the IDE integrated help, which you can access by pressing the F1 key or through the Help menu in the IDE. See "Profiling C/C++/Fortran Applications" , "Detecting Data Races and Deadlocks" and "Finding Memory Access Errors in Your Project" in the help Contents tab.

For More Information

See the Oracle Solaris Studio product pages on the Oracle Technology Network for more information. You can find white papers, technical articles, training and support information, and community forums and blogs to help you get more out of Oracle Solaris Studio.