# Oracle® Solaris Studio 12.4: Overview

Part No: E37073
December 2014

# Contents

# Using This Documentation

- **Overview** – Describes 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.
- **Audience** – Application developers, system developers, architects, support engineers
- **Required knowledge** – None

## Product Documentation Library

The product documentation library is located at `http://docs.oracle.com/cd/E37069_01`.

System requirements and known problems are included in the "Oracle Solaris Studio 12.4: Release Notes ".

## 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.

## Feedback

Provide feedback about this documentation at `http://www.oracle.com/goto/docfeedback`.

# Oracle Solaris Studio 12.4 Overview

Oracle Solaris Studio software is a set of software development tools for C, C++, and Fortran development on Oracle Solaris™ and Linux platforms, with support of multicore systems with SPARC® processors and x86 and x64 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 newer multicore CPUs including the SPARC T5, SPARC M5, SPARC M6, SPARC M10, SPARC M10+, and Intel Ivy Bridge and Haswell processors as well as the older SPARC T4, SPARC T3, 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.
- 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.

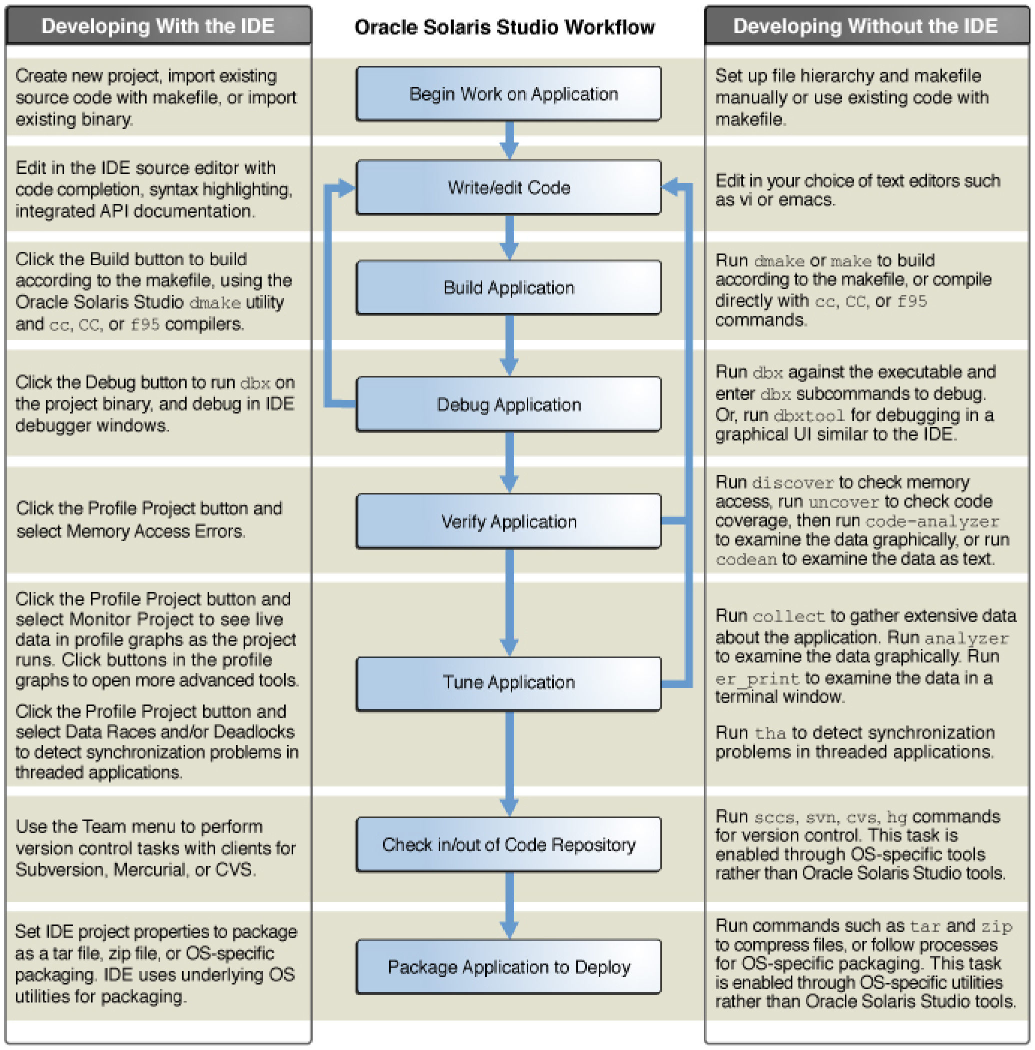
## Developer Workflow for Oracle Solaris Studio

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 Performance Analyzer 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.



| Developing With the IDE | Oracle Solaris Studio Workflow | Developing Without the IDE |
|---|---|---|
| Create new project, import existing source code with makefile, or import existing binary. | Begin Work on Application | Set up file hierarchy and makefile manually or use existing code with makefile. |
| Edit in the IDE source editor with code completion, syntax highlighting, integrated API documentation. | Write/edit Code | Edit in your choice of text editors such as vi or emacs. |
| Click the Build button to build according to the makefile, using the Oracle Solaris Studio `dmake` utility and `cc`, `CC`, or `f95` compilers. | Build Application | Run `dmake` or `make` to build according to the makefile, or compile directly with `cc`, `CC`, or `f95` commands. |
| Click the Debug button to run `dbx` on the project binary, and debug in IDE debugger windows. | Debug Application | Run `dbx` against the executable and enter `dbx` subcommands to debug. Or, run `dbxtool` for debugging in a graphical UI similar to the IDE. |
| Click the Profile Project button and select Memory Access Errors. | Verify Application | Run `discover` to check memory access, run `uncover` to check code coverage, then run `code-analyzer` to examine the data graphically, or run `codean` to examine the data as text. |
| Click the Profile Project button and select Monitor Project to see live data in profile graphs as the project runs. Click buttons in the profile graphs to open more advanced tools. Click the Profile Project button and select Data Races and/or Deadlocks to detect synchronization problems in threaded applications. | Tune Application | Run `collect` to gather extensive data about the application. Run `analyzer` to examine the data graphically. Run `er_print` to examine the data in a terminal window. Run `tha` to detect synchronization problems in threaded applications. |
| Use the Team menu to perform version control tasks with clients for Subversion, Mercurial, or CVS. | Check in/out of Code Repository | Run `sccs`, `svn`, `cvs`, `hg` commands for version control. This task is enabled through OS-specific tools rather than Oracle Solaris Studio tools. |
| Set IDE project properties to package as a tar file, zip file, or OS-specific packaging. IDE uses underlying OS utilities for packaging. | Package Application to Deploy | Run commands such as `tar` and `zip` to compress files, or follow processes for OS-specific packaging. This task is enabled through OS-specific utilities rather than Oracle Solaris Studio tools. |

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

Oracle Solaris Studio IDE is based on NetBeans IDE, an open-source integrated development environment. 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.

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 Studio performance utilities invisibly to collect data on your running project.

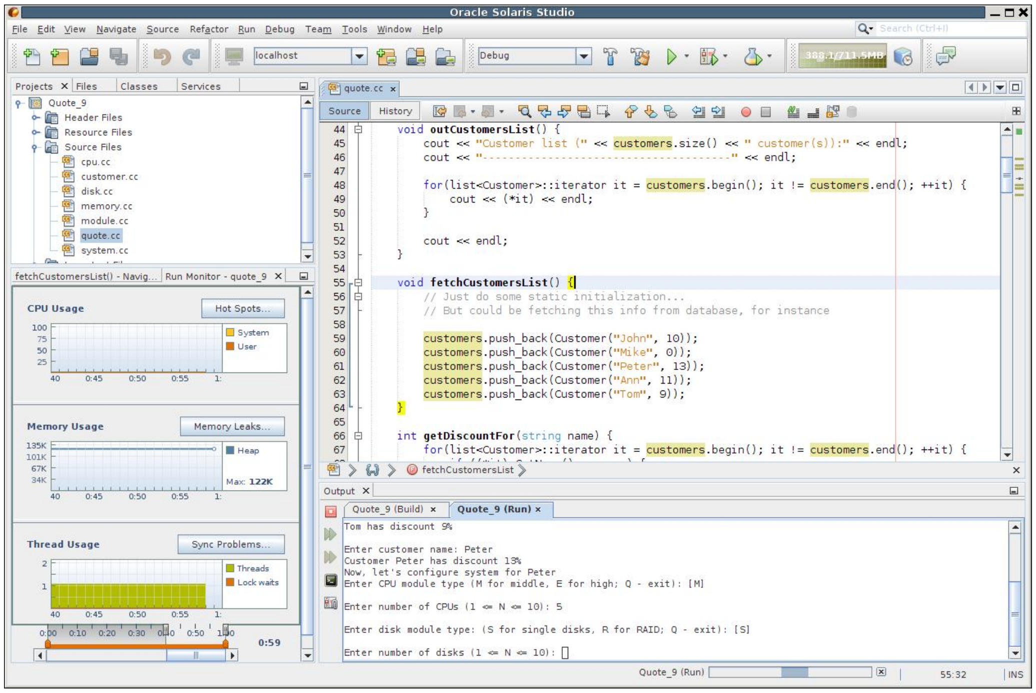
Using Oracle Solaris Studio IDE offers some advantages over development using text editors and the command line:

- **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, dbxtool, Code Analyzer, and Performance Analyzer on a desktop system while using 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 or other tool 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 with the Monitor Project profiling tools.



```
    void outCustomersList() {
        cout << "Customer list (" << customers.size() << " customer(s)):" << endl;
        cout << "-----------------------------------" << endl;

        for(list<Customer>::iterator it = customers.begin(); it != customers.end(); ++it) {
            cout << (*it) << endl;
        }

        cout << endl;
    }

    void fetchCustomersList() {
        // Just do some static initialization...
        // But could be fetching this info from database, for instance

        customers.push_back(Customer("John", 10));
        customers.push_back(Customer("Mike", 0));
        customers.push_back(Customer("Peter", 13));
        customers.push_back(Customer("Ann", 11));
        customers.push_back(Customer("Tom", 9));
    }

    int getDiscountFor(string name) {
        for(list<Customer>::iterator it = customers.begin(); it != customers.end(); ++it) {
```

```
Tom has discount 9%

Enter customer name: Peter
Customer Peter has discount 13%
Now, let's configure system for Peter
Enter CPU module type (M for middle, E for high; Q - exit): [M]

Enter number of CPUs (1 <= N <= 10): 5

Enter disk module type: (S for single disks, R for RAID; Q - exit): [S]

Enter number of disks (1 <= N <= 10): 
```

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 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 12.4: 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. |

Oracle Solaris Studio compilers provide significantly more information than other compilers to help you understand your code. 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 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, and some of the *Programming Language - C and ISO/IEC 9899:2011* standard. The C compiler also supports the OpenMP 4.0 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`*dir*`] [-l`*library*`]...`

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`*dir* option to add directories to the list that the linker searches for libraries, and the-`l`*library* option to add object libraries to the linker's list of search libraries. The `-L` option must precede the associated library on the command line.

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 12.4: C User's Guide " and the `cc`(1) man page. For information about the new and changed features, see "What's New in Oracle Solaris Studio 12.4 ". For information about problems and workarounds, and limitations and incompatibilities of the compiler, see "Oracle Solaris Studio 12.4: Release Notes ".

## C++ Compiler

The Oracle Solaris Studio C++ compiler (`CC`) supports the *ISO International Standard for C++, ISO/IEC 14882:2011, Programming Language — C++* and *ISO International Standard for C++, ISO IS 14822:2003, Programming Language — C++*. The `CC` compiler also supports the OpenMP 4.0 shared-memory parallelism API. The OpenMP 4.0 API is included with Oracle Solaris Studio 12.4.

For specific information about C++11 support, see "Support for the C++11 Standard" in "What's New in Oracle Solaris Studio 12.4 ".

The C++ compiler (`CC`) produces 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. The compiler automatically parallelizes serial source code to produce binaries with better performance on multicore systems and can also prepare binaries for enhanced debugging or analysis by other Oracle Solaris Studio tools. The compiler also supports GNU C/C++ compatibility features.

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`*dir*`] [-l` *library*`]...`

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`*dir* option to add directories to the list that the linker searches for libraries, and the-`l`*library* option to add object libraries to the linker's list of search libraries. The `-L` option must precede the associated library on the command line.

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 12.4: C++ User's Guide " and the `CC`(1) man page. For information about the new and changed features, see "What's New in Oracle Solaris Studio 12.4 ". For information about problems and workarounds, and limitations and incompatibilities of the compiler, see "Oracle Solaris Studio 12.4: Release Notes ".

## 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 4.0.

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 12.4: Fortran User's Guide " and the `f95`(1) man page. For information about the new and changed features, see "What's New in Oracle Solaris Studio 12.4 ". For information about problems and workarounds, and limitations and incompatibilities of the compiler, see "Oracle Solaris Studio 12.4: Release Notes ".

## 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.

To link a `/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.

Runtime libraries for Fortran, C, and C++ are also provided with Oracle Solaris Studio in addition to those provided in the operating systems. Some examples include the `libsunmath` and `libmopt` math libraries,

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 4.0 for Parallel Programming

OpenMP is an Application Programming Interface (API) to write shared memory parallel applications in C, C++ and Fortran. It consists of a set of compiler directives, library routines, and environment variables.

Programming in OpenMP has following advantages:

- Can dramatically improve program performance on modern multicore architectures.
- Enables programmers to easily write portable code since OpenMP is supported on a large number of compilers.
- Requires little programming effort. Programmers identify the code that can be parallelized in existing programs, and add pragmas to parallelize it.
- Enables programmers to parallelize their code incrementally.

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. See the “Oracle Solaris Studio 12.4: OpenMP API User’s Guide ” for details.

## Oracle Solaris Studio Performance Library for Programs With Intensive Computation

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

Oracle Solaris Studio Performance Library routines can increase application performance on multicore and multiprocessor (MP) Oracle systems. Many routines have SPARC and x86 specific optimizations that are not present in the base Netlib libraries, as well as parallelization using OpenMP. Besides the standard Fortran interfaces, a complete set of C interfaces is also included.

The Oracle Solaris Studio 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 Oracle Solaris Studio Performance Library, you must use the options `-library=sunperf` and `-staticlib=sunperf`.

For complete information about using the Oracle Solaris Studio Performance Library, see the “Oracle Solaris Studio 12.4: Performance Library User’s Guide” in PDF format in the `Information Library`. For man pages for each function and subroutine in the library, see section 3p of the Oracle Solaris Studio man pages. For information about the new and changed features of the Oracle Solaris Studio Performance Library, see “What’s New in Oracle Solaris Studio 12.4 ”.

## `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.

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 12.4: Distributed Make (dmake) " or the `dmake`(1) man page.

## 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 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 19
- "`dbx` in the IDE" on page 20
- "`dbx` in `dbxtool`" on page 21

### `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 12.4: 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, see "What's New in Oracle Solaris Studio 12.4 ".

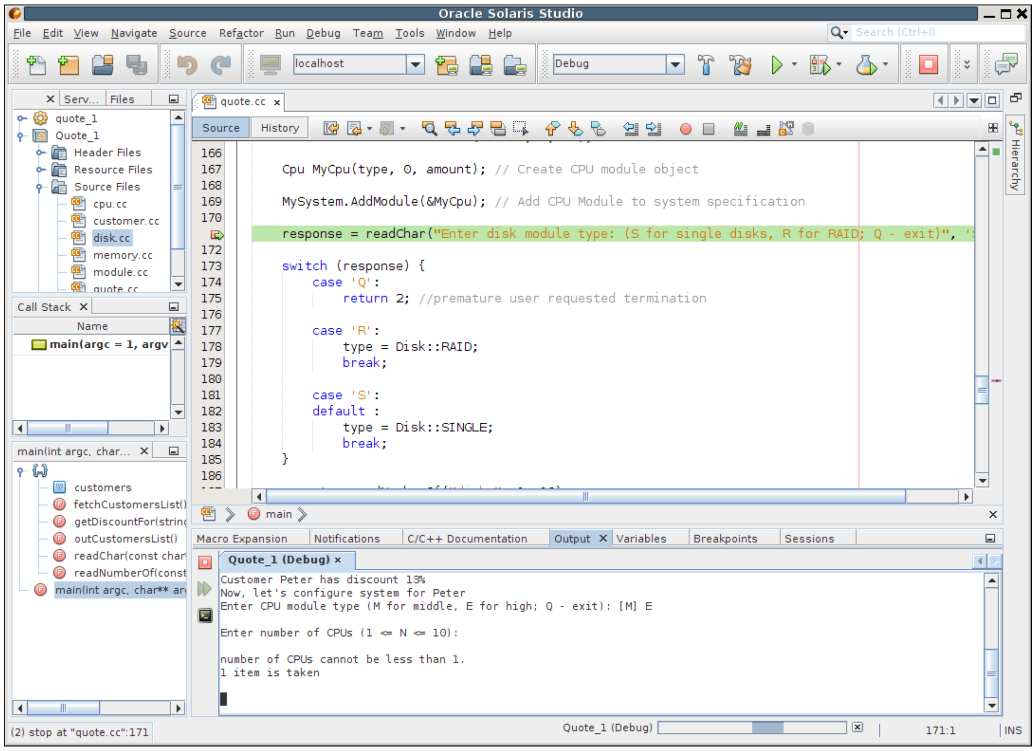
For known problems, limitations, and incompatibilities in the current release of dbx, see "Oracle Solaris Studio 12.4: Release Notes ".

## dbx in the IDE

You can use dbx in 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. You can also use the IDE to debug executables that are not associated with an IDE 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.



```
    Cpu MyCpu(type, 0, amount); // Create CPU module object

    MySystem.AddModule(&MyCpu); // Add CPU Module to system specification

    response = readChar("Enter disk module type: (S for single disks, R for RAID; Q - exit)", '

    switch (response) {
        case 'Q':
            return 2; //premature user requested termination

        case 'R':
            type = Disk::RAID;
            break;

        case 'S':
        default :
            type = Disk::SINGLE;
            break;
    }
```

```
Customer Peter has discount 13%
Now, let's configure system for Peter
Enter CPU module type (M for middle, E for high; Q - exit): [M] E

Enter number of CPUs (1 <= N <= 10):

number of CPUs cannot be less than 1.
1 item is taken
```

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 12.4: IDE Quick Start Tutorial ".

## `dbx` in `dbxtool`

You can also use `dbx` through `dbxtool`, a graphical tool 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.

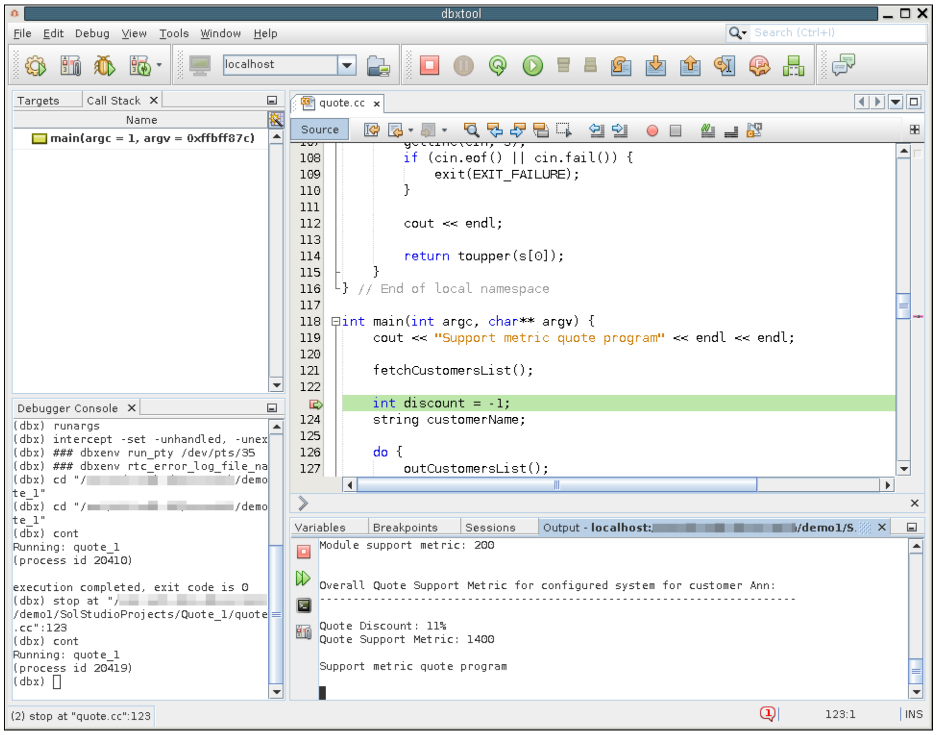
To start `dbxtool`, type:

```
% 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.


dbxtool
File Edit Debug View Tools Window Help
Search (Ctrl+I)
localhost
Targets
Call Stack
Name
main(argc = 1, argv = 0xffbff87c)
quote.cc
Source
Variables
Breakpoints
Sessions
Output - localhost:[redacted]/demo1/S.
123:1
INS

```
        if (cin.eof() || cin.fail()) {
            exit(EXIT_FAILURE);
        }

        cout << endl;

        return toupper(s[0]);
    }
} // End of local namespace

int main(int argc, char** argv) {
    cout << "Support metric quote program" << endl << endl;

    fetchCustomersList();

    int discount = -1;
    string customerName;

    do {
        outCustomersList();
```

Debugger Console

```
(dbx) runargs
(dbx) intercept -set -unhandled, -unex
(dbx) ### dbxenv run_pty /dev/pts/35
(dbx) ### dbxenv rtc_error_log_file_na
(dbx) cd "/[redacted]/demo
te_1"
(dbx) cd "/[redacted]/demo
te_1"
(dbx) cont
Running: quote_1
(process id 20410)

execution completed, exit code is 0
(dbx) stop at "/[redacted]
/demo1/SolStudioProjects/Quote_1/quote
.cc":123
(dbx) cont
Running: quote_1
(process id 20419)
(dbx)
```

```
Module support metric: 200


Overall Quote Support Metric for configured system for customer Ann:
----------------------------------------------------------------------


Quote Discount: 11%
Quote Support Metric: 1400

Support metric quote program
```

(2) stop at "quote.cc":123

For information about using `dbxtool`, see the `dbxtool`(1) man page and the integrated help in `dbxtool`. The "Oracle Solaris Studio 12.4: 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, Code Analyzer lets you find errors in your application that you would not find when using other error detection tools by themselves. |
| `codean` | A command-line utility that provides functionality similar to Code Analyzer. |

## `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.

The `discover` utility 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 `-o` 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 "Oracle Solaris Studio 12.4: Discover and Uncover User's Guide ".

## `uncover` Tool for Measuring Code Coverage

The `uncover` utility 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 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 12.4: Discover and Uncover User's Guide ".

## Code Analyzer Tool For Integrated Error Checking

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

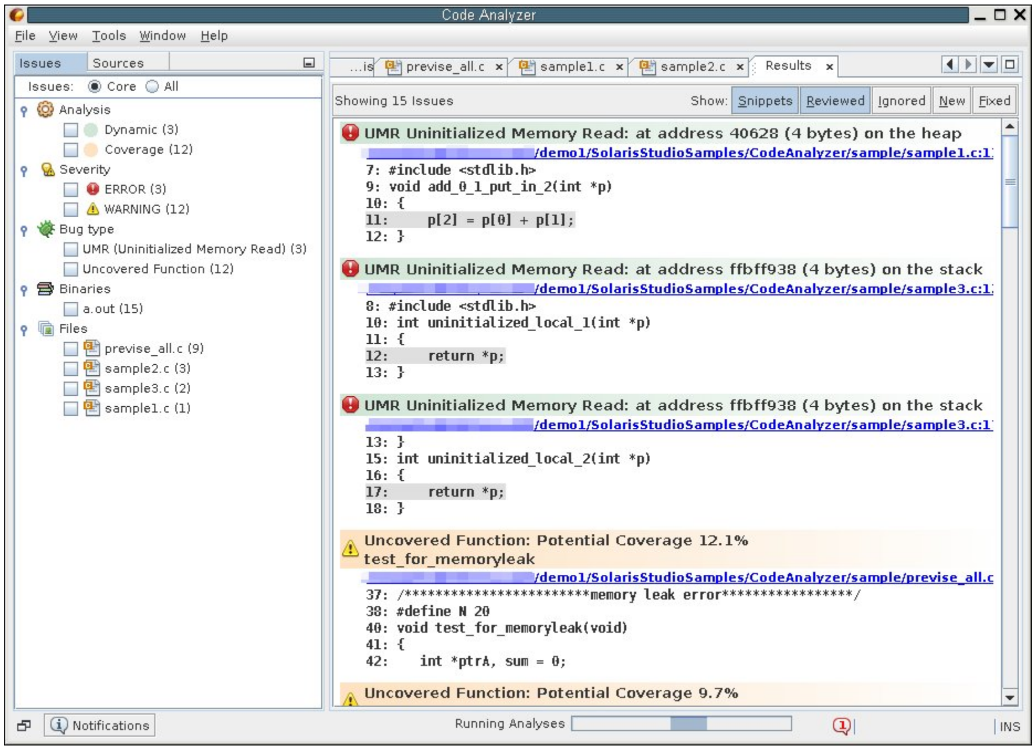
- Static code checking, which is performed when you compile your application with the Oracle Solaris Studio C or C++ compiler and specify the -xanalyze=code option.
- Dynamic memory access checking, which is performed when you instrument your binary with discover using the -a option, and then run the 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 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 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, Code Analyzer is displaying the issues found in the `a.out` binary.



UMR Uninitialized Memory Read: at address 40628 (4 bytes) on the heap
/demo1/SolarisStudioSamples/CodeAnalyzer/sample/sample1.c:1

```
#include <stdlib.h>
void add_0_1_put_in_2(int *p)
{
    p[2] = p[0] + p[1];
}
```

UMR Uninitialized Memory Read: at address ffbff938 (4 bytes) on the stack
/demo1/SolarisStudioSamples/CodeAnalyzer/sample/sample3.c:1

```
#include <stdlib.h>
int uninitialized_local_1(int *p)
{
    return *p;
}
```

UMR Uninitialized Memory Read: at address ffbff938 (4 bytes) on the stack
/demo1/SolarisStudioSamples/CodeAnalyzer/sample/sample3.c:1

```
}
int uninitialized_local_2(int *p)
{
    return *p;
}
```

Uncovered Function: Potential Coverage 12.1%
test_for_memoryleak
/demo1/SolarisStudioSamples/CodeAnalyzer/sample/previse_all.c

```
/***********************memory leak error****************/
#define N 20
void test_for_memoryleak(void)
{
   int *ptrA, sum = 0;
```

Uncovered Function: Potential Coverage 9.7%

For more information, see the integrated Code Analyzer help, "Oracle Solaris Studio 12.4: Code Analyzer User's Guide ", and "Oracle Solaris Studio 12.4: Code Analyzer Tutorial ".

## `codean` Tool for Integrated Checking

You can also generate reports from the data collected through the compilers, `discover`, and `uncover` with the `codean` command-line utility. The `codean` tool provides functionality similar to Code Analyzer, but you can use it on systems where a graphical environment is not available, or if you prefer the command line. The `codean` tool can also be used in automated scripts and has some features that are not yet available in the Code Analyzer tool.

For more information, see the `codean`(1) man page, "Oracle Solaris Studio 12.4: Code Analyzer User's Guide ", and "Oracle Solaris Studio 12.4: Code Analyzer Tutorial ".

## 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 the IDE.** Enable 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, Performance Analyzer, Thread Analyzer, and `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. 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.

Performance Analyzer tools allow you to have control over the data that is collected, inspect the data deeply, and examine your program's interaction with the hardware. 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. Profile a target application in Performance Analyzer or collect performance data from the target application with the `collect` command.
2. Examine the data with the Performance Analyzer graphical tool, or the `er_print` command line utility, or the Thread Analyzer graphical tool for detecting data races and deadlocks on multithreaded applications.

## Collect Performance Data to Profile an Application

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 Profile Application dialog in 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 `-i on` option causes the Collector to perform input/output tracing. 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 (Process ID: 11103) ...
00:00:00.000  ===== (11103) synprog run
00:00:00.005  ===== (11103) Mon  22 Sep 14  17:05:51 Stopwatch calibration
  OS release 5.11 -- enabling microstate accounting 5.11.
        0.000096 s.  (22.4 % of 0.000426 s.) -- inner
 N = 1000, avg = 0.096 us., min = 0.090, max = 0.105
        0.000312 s.  (67.0 % of 0.000466 s.) -- outer
 N = 1000, avg = 0.312 us., min = 0.307, max = 0.457
00:00:00.006  ===== (11103)  Begin commandline
 icpu.md.cpu.rec.recd.dousl.gpf.fitos.uf.ec.tco.b.nap.sig.sys.so.sx.so
```

```
00:00:00.006  ===== (11103) start of icputime
    3.003069 wall-secs.,   2.978360 CPU-secs., in icputime
00:00:03.009  ===== (11103) start of muldiv
    3.007489 wall-secs.,   2.997647 CPU-secs., in muldiv
00:00:06.017  ===== (11103) start of cputime
    3.002315 wall-secs.,   2.989407 CPU-secs., in cputime
00:00:09.019  ===== (11103) start of recurse
    3.082371 wall-secs.,   3.069782 CPU-secs., in 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`.

See the "Oracle Solaris Studio 12.4: Performance Analyzer Tutorials " for step-by-step instructions for using Performance Analyzer on sample applications you can download.

For detailed information about profiling applications and using the Collector, see the Help menu in Performance Analyzer, the "Oracle Solaris Studio 12.4: Performance Analyzer " manual, and the `collect`(1) man page.

## Examine Performance Data With Performance Analyzer

Performance Analyzer provides insight into the behavior of your application to enable you to find problem areas in your code. Performance Analyzer identifies which functions, code segments, and source lines are using the most system resources. Performance Analyzer can profile single-threaded, multithreaded, and multi-process applications, then present the profiling data to help you identify where you can improve your application's performance.

You can run Performance Analyzer with the `analyzer` command. The basic syntax of the `analyzer` command to start 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, Performance Analyzer aggregates the experiment data by default, but can also be used to compare the experiments if you specify the `-c` option on the command line before the experiment names.

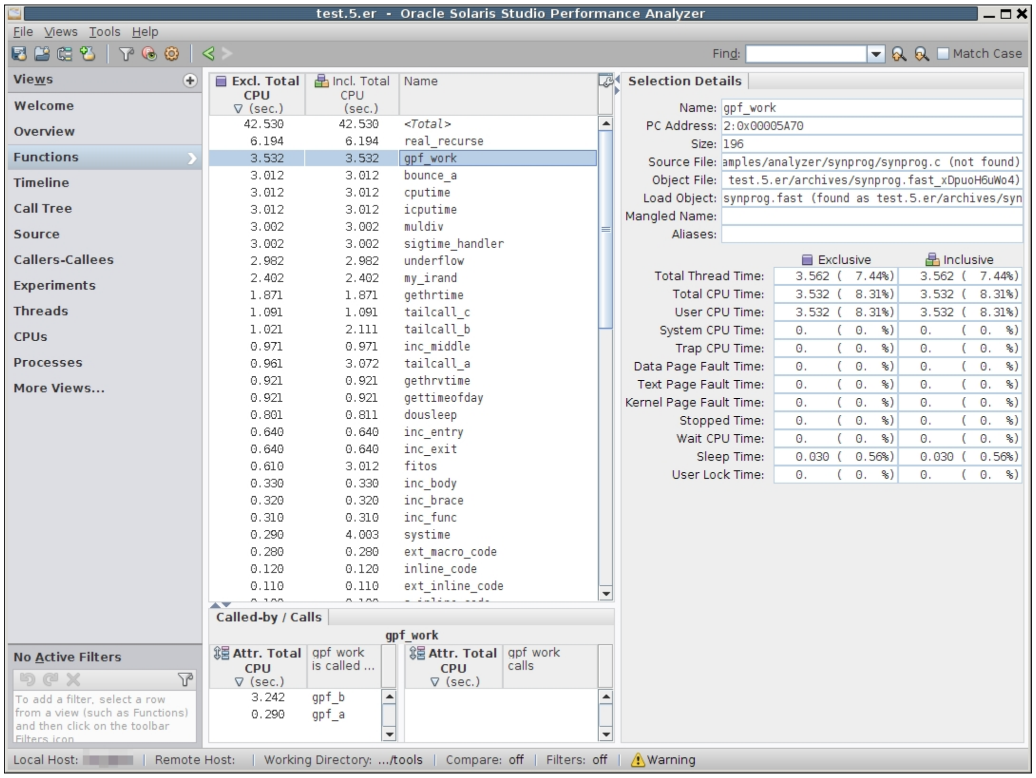
If you do not specify an experiment on the command line, Performance Analyzer displays a Welcome screen to help you get started.

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

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

The initial view of the experiment is the Overview where you can get a quick overview of time and resources used by your program and select the performance metrics you want to see in the views of performance data.

The following figure shows Performance Analyzer's Functions view for a `test.1.er` experiment that was made on the `synprog` example. The Functions view shows the CPU time used by each function of the `synprog` program. When you click the function `gpf_work` the Selection Details window on the right side shows details about the `gpf_work` function's resource usage. At the bottom of the Functions view, the Called-by/Calls area shows the functions that are called by `gpf_work` and you can double-click the calls to navigate to them in the Functions view..



| Excl. Total CPU ▽ (sec.) | Incl. Total CPU (sec.) | Name |
|---|---|---|
| 42.530 | 42.530 | <Total> |
| 6.194 | 6.194 | real_recurse |
| 3.532 | 3.532 | gpf_work |
| 3.012 | 3.012 | bounce_a |
| 3.012 | 3.012 | cputime |
| 3.012 | 3.012 | icputime |
| 3.002 | 3.002 | muldiv |
| 3.002 | 3.002 | sigtime_handler |
| 2.982 | 2.982 | underflow |
| 2.402 | 2.402 | my_irand |
| 1.871 | 1.871 | gethrtime |
| 1.091 | 1.091 | tailcall_c |
| 1.021 | 2.111 | tailcall_b |
| 0.971 | 0.971 | inc_middle |
| 0.961 | 3.072 | tailcall_a |
| 0.921 | 0.921 | gethrvtime |
| 0.921 | 0.921 | gettimeofday |
| 0.801 | 0.811 | dousleep |
| 0.640 | 0.640 | inc_entry |
| 0.640 | 0.640 | inc_exit |
| 0.610 | 3.012 | fitos |
| 0.330 | 0.330 | inc_body |
| 0.320 | 0.320 | inc_brace |
| 0.310 | 0.310 | inc_func |
| 0.290 | 4.003 | systime |
| 0.280 | 0.280 | ext_macro_code |
| 0.120 | 0.120 | inline_code |
| 0.110 | 0.110 | ext_inline_code |

Name: gpf_work
PC Address: 2:0x00005A70
Size: 196
Source File: amples/analyzer/synprog/synprog.c (not found)
Object File: test.5.er/archives/synprog.fast_xDpuoH6uWo4)
Load Object: synprog.fast (found as test.5.er/archives/syn
Mangled Name:
Aliases:

| | Exclusive | Inclusive |
|---|---|---|
| Total Thread Time: | 3.562 ( 7.44%) | 3.562 ( 7.44%) |
| Total CPU Time: | 3.532 ( 8.31%) | 3.532 ( 8.31%) |
| User CPU Time: | 3.532 ( 8.31%) | 3.532 ( 8.31%) |
| System CPU Time: | 0. ( 0. %) | 0. ( 0. %) |
| Trap CPU Time: | 0. ( 0. %) | 0. ( 0. %) |
| Data Page Fault Time: | 0. ( 0. %) | 0. ( 0. %) |
| Text Page Fault Time: | 0. ( 0. %) | 0. ( 0. %) |
| Kernel Page Fault Time: | 0. ( 0. %) | 0. ( 0. %) |
| Stopped Time: | 0. ( 0. %) | 0. ( 0. %) |
| Wait CPU Time: | 0. ( 0. %) | 0. ( 0. %) |
| Sleep Time: | 0.030 ( 0.56%) | 0.030 ( 0.56%) |
| User Lock Time: | 0. ( 0. %) | 0. ( 0. %) |

gpf_work

| Attr. Total CPU ▽ (sec.) | gpf work is called ... |
|---|---|
| 3.242 | gpf_b |
| 0.290 | gpf_a |

| Attr. Total CPU ▽ (sec.) | gpf work calls |
|---|---|

To add a filter, select a row from a view (such as Functions) and then click on the toolbar Filters icon

Local Host: [redacted] | Remote Host: | Working Directory: .../tools | Compare: off | Filters: off | Warning

For information about using Performance Analyzer, see the "Oracle Solaris Studio 12.4: Performance Analyzer " manual, Performance Analyzer integrated help, and the `analyzer`(1) man page.

See the "Oracle Solaris Studio 12.4: Performance Analyzer Tutorials " for step-by-step instructions for using Performance Analyzer on sample applications you can download.

## 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 Total CPU Time

Excl.     Incl.      Name
Total     Total
CPU sec.  CPU sec.
50.806    50.806     <Total>
 5.994     5.994     so_burncpu
 5.914     5.914     real_recurse
 3.502     3.502     gpf_work
 3.012     3.012     sigtime_handler
 3.002     3.002     bounce_a
 3.002     3.002     cputime
 3.002     3.002     icputime
 2.992     2.992     sx_burncpu
 2.992     2.992     underflow
 2.792     2.792     muldiv
 2.532     2.532     my_irand
 1.831     1.831     gethrtime
 1.031     1.991     tailcall_b
 0.961     0.961     inc_middle
 0.961     0.961     tailcall_c
 0.941     0.941     gethrvtime
 0.941     0.941     gettimeofday
 0.911     2.902     tailcall_a
 0.801     0.801     dousleep
 0.650     0.650     inc_entry
 0.640     0.640     inc_exit
```

```
0.480    3.012    fitos
0.330    0.330    inc_func
0.320    0.320    inc_body
0.320    0.320    inc_brace
0.290    4.003    systime
0.260    0.260    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 12.4: Performance Analyzer " manual and the `er_print`(1) man page.

## Analyze Multithreaded Application Performance With Thread Analyzer

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

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.

Thread Analyzer is streamlined for multithreaded program analysis and shows only the Races, Deadlocks, and Dual Source data views of Performance Analyzer. For OpenMP programs, the OpenMP Parallel Region and OpenMP Task views 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.

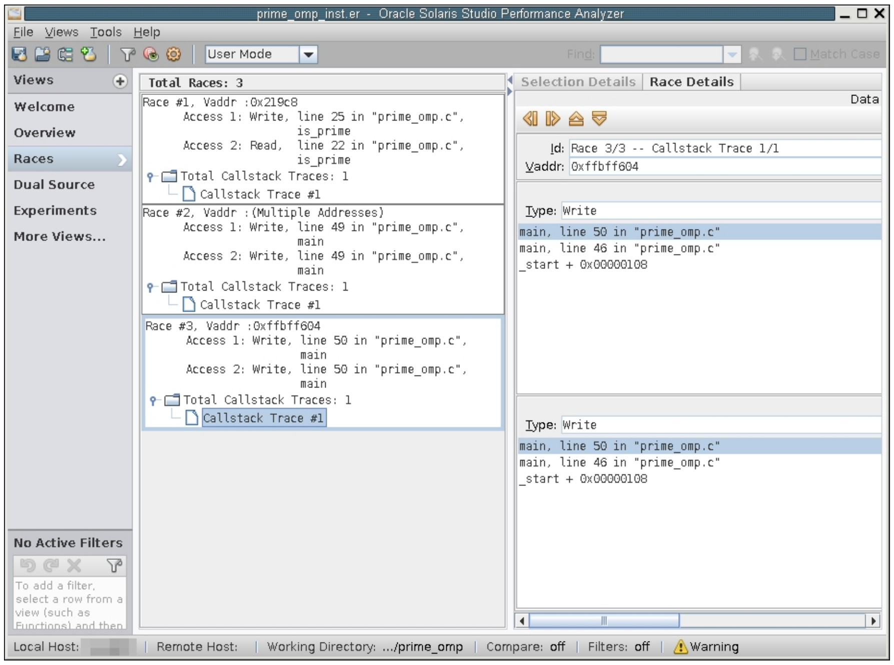
To use Thread Analyzer:

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 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.



```
Total Races: 3
Race #1, Vaddr :0x219c8
      Access 1: Write, line 25 in "prime_omp.c",
                     is_prime
      Access 2: Read,  line 22 in "prime_omp.c",
                     is_prime
  Total Callstack Traces: 1
    Callstack Trace #1
Race #2, Vaddr :(Multiple Addresses)
      Access 1: Write, line 49 in "prime_omp.c",
                     main
      Access 2: Write, line 49 in "prime_omp.c",
                     main
  Total Callstack Traces: 1
    Callstack Trace #1
Race #3, Vaddr :0xffbff604
      Access 1: Write, line 50 in "prime_omp.c",
                     main
      Access 2: Write, line 50 in "prime_omp.c",
                     main
  Total Callstack Traces: 1
    Callstack Trace #1
```

```
   Id: Race 3/3 -- Callstack Trace 1/1
Vaddr: 0xffbff604

Type: Write
main, line 50 in "prime_omp.c"
main, line 46 in "prime_omp.c"
_start + 0x00000108

Type: Write
main, line 50 in "prime_omp.c"
main, line 46 in "prime_omp.c"
_start + 0x00000108
```

For information about using Thread Analyzer, see the `tha`(1) man page and the "Oracle Solaris Studio 12.4: 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. 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 `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
```

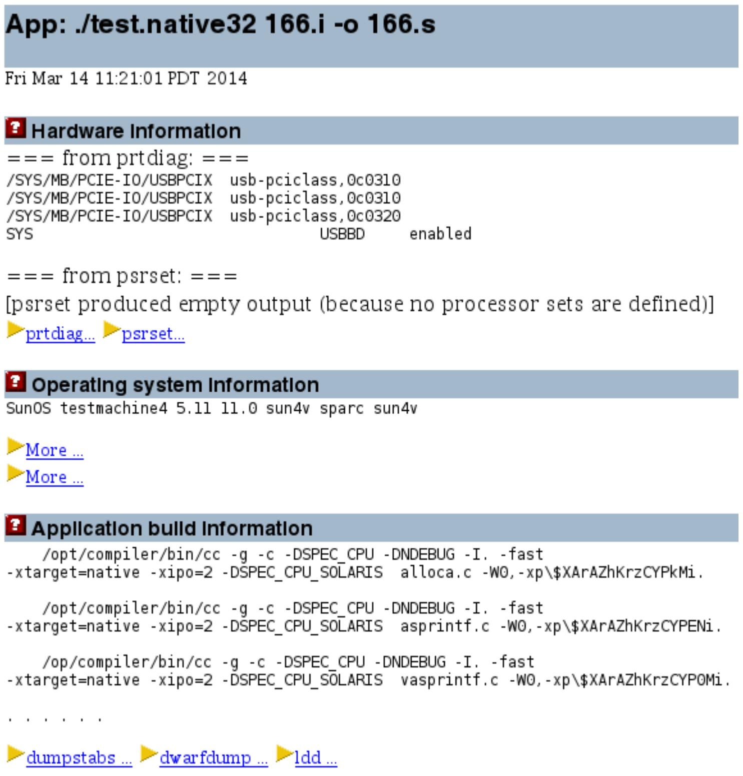
To run SPOT on an already running application:

```
% 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.



App: ./test.native32 166.i -o 166.s

Fri Mar 14 11:21:01 PDT 2014

Hardware Information

```
=== from prtdiag: ===
/SYS/MB/PCIE-IO/USBPCIX  usb-pciclass,0c0310
/SYS/MB/PCIE-IO/USBPCIX  usb-pciclass,0c0310
/SYS/MB/PCIE-IO/USBPCIX  usb-pciclass,0c0320
SYS                                USBBD     enabled

=== from psrset: ===
[psrset produced empty output (because no processor sets are defined)]
```

prtdiag... psrset...

Operating system Information

```
SunOS testmachine4 5.11 11.0 sun4v sparc sun4v
```

More ...
More ...

Application build Information

```
    /opt/compiler/bin/cc -g -c -DSPEC_CPU -DNDEBUG -I. -fast
-xtarget=native -xipo=2 -DSPEC_CPU_SOLARIS  alloca.c -W0,-xp\$XArAZhKrzCYPkMi.

    /opt/compiler/bin/cc -g -c -DSPEC_CPU -DNDEBUG -I. -fast
-xtarget=native -xipo=2 -DSPEC_CPU_SOLARIS  asprintf.c -W0,-xp\$XArAZhKrzCYPENi.

    /op/compiler/bin/cc -g -c -DSPEC_CPU -DNDEBUG -I. -fast
-xtarget=native -xipo=2 -DSPEC_CPU_SOLARIS  vasprintf.c -W0,-xp\$XArAZhKrzCYP0Mi.

. . . . . .
```

dumpstabs ... dwarfdump ... ldd ...

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” in the Oracle Solaris Studio 12.2 documentation library at .

## Profiling Tools in the IDE

Oracle Solaris Studio IDE provides interactive graphical profiling tools to enable you to examine the performance of your projects as they run within the IDE. The profiling tools use Oracle Solaris Studio utilities and operating system utilities to collect the data.

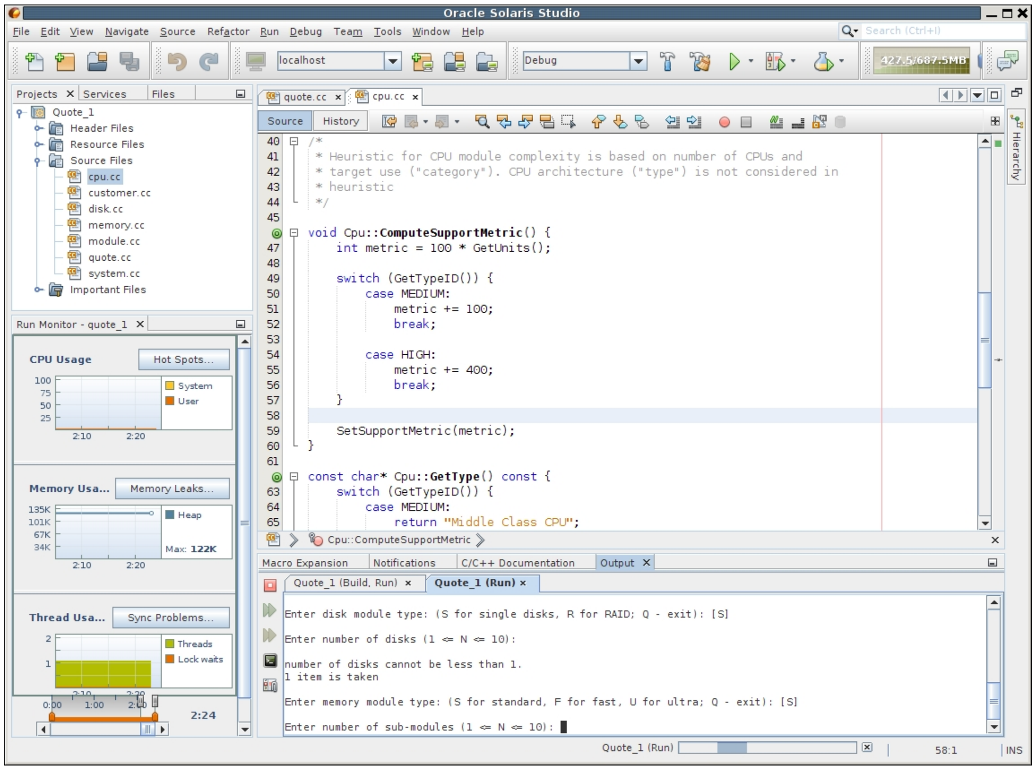
The profiling tools are available from the Profile Project button .

| | |
|---|---|
| Monitor Project | Presents graphs that enable you to see a summary of resource usage of your program. |

| | |
|---|---|
| Memory Access Errors | Analyzes the program as it runs to detect memory access errors and memory leaks. |
| Data Races and Deadlocks Detection | Analyzes the program as it runs to detect actual and potential data races and deadlocks among the threads. |

When you profile your project and choose Monitor Project, the Run Monitor window opens to display the output of the low-impact tools for CPU Usage, Memory Usage, and Thread Usage.

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



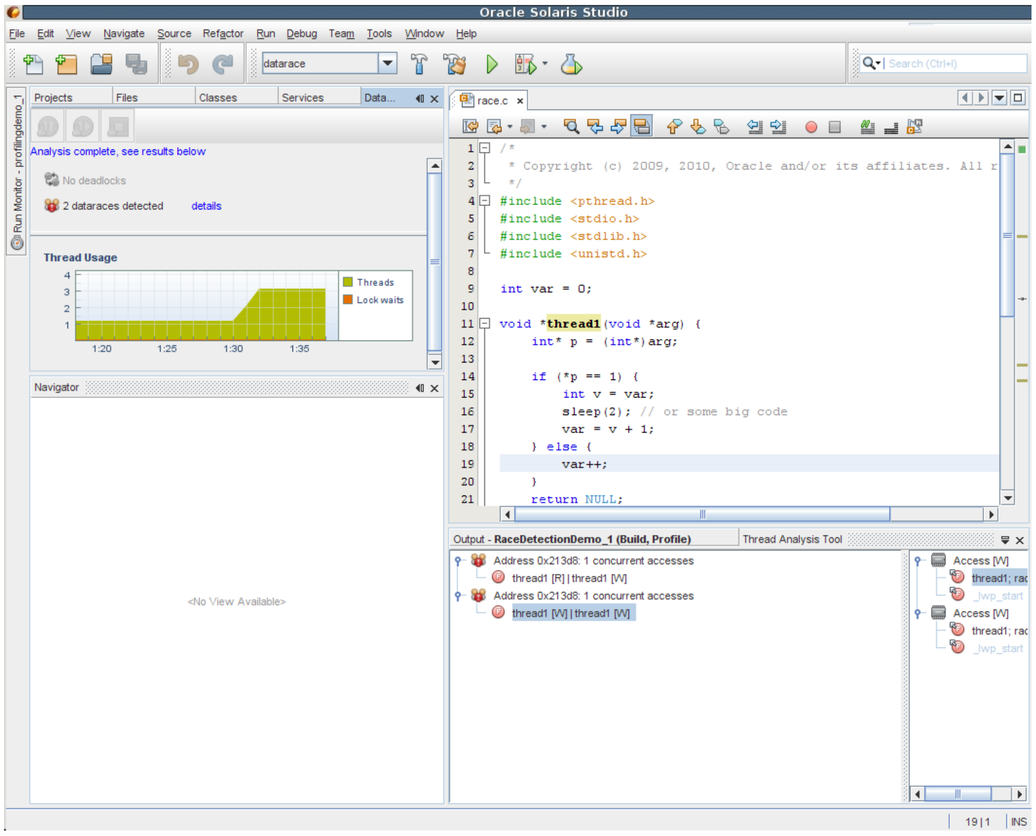
```
/*
 * Heuristic for CPU module complexity is based on number of CPUs and
 * target use ("category"). CPU architecture ("type") is not considered in
 * heuristic
 */

void Cpu::ComputeSupportMetric() {
    int metric = 100 * GetUnits();

    switch (GetTypeID()) {
        case MEDIUM:
            metric += 100;
            break;

        case HIGH:
            metric += 400;
            break;
    }

    SetSupportMetric(metric);
}

const char* Cpu::GetType() const {
    switch (GetTypeID()) {
        case MEDIUM:
            return "Middle Class CPU";
```

```
Enter disk module type: (S for single disks, R for RAID; Q - exit): [S]

Enter number of disks (1 <= N <= 10):

number of disks cannot be less than 1.
1 item is taken

Enter memory module type: (S for standard, F for fast, U for ultra; Q - exit): [S]

Enter number of sub-modules (1 <= N <= 10):
```

Additional tools for more detailed profiling have a greater performance impact on the system and the application, so those tools do not run automatically when you run Monitor Project. The advanced tools are linked to the Run Monitor tools and can be launched easily by clicking buttons to see Hot Spots, Memory Leaks, and Sync Problems.

The Data Races and Deadlocks Detection tool uses the same underlying technology as 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.


Oracle Solaris Studio
File Edit View Navigate Source Refactor Run Debug Team Tools Window Help
datarace
Search (Ctrl+I)
Run Monitor - profilingdemo_1
Projects Files Classes Services Data...
Thread Usage
Threads
Lock waits
1:20 1:25 1:30 1:35
Navigator
<No View Available>
race.c
Output - RaceDetectionDemo_1 (Build, Profile)
Thread Analysis Tool
19 | 1 INS

Analysis complete, see results below

No deadlocks

2 dataraces detected details

```
/*
 * Copyright (c) 2009, 2010, Oracle and/or its affiliates. All r
 */
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int var = 0;

void *thread1(void *arg) {
    int* p = (int*)arg;

    if (*p == 1) {
        int v = var;
        sleep(2); // or some big code
        var = v + 1;
    } else {
        var++;
    }
    return NULL;
```

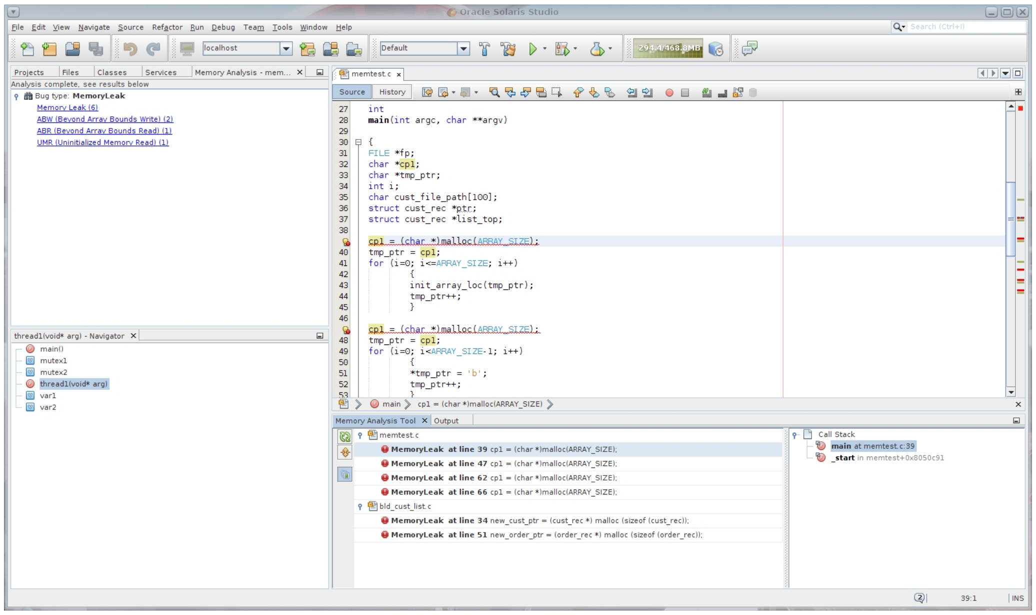
```
Address 0x213d8: 1 concurrent accesses
    thread1 [R] | thread1 [W]
Address 0x213d8: 1 concurrent accesses
    thread1 [W] | thread1 [W]
```

```
Access [W]
    thread1; rac
    _lwp_start
Access [W]
    thread1; rac
    _lwp_start
```

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.


Oracle Solaris Studio
File Edit View Navigate Source Refactor Run Debug Team Tools Window Help
localhost
Default
Projects Files Classes Services Memory Analysis - mem...
memtest.c
Source History
thread1(void* arg) - Navigator
main()
mutex1
mutex2
thread1(void* arg)
var1
var2
Memory Analysis Tool
Output
39:1 INS

Analysis complete, see results below

Bug type: **MemoryLeak**
- Memory Leak (6)
- ABW (Beyond Array Bounds Write) (2)
- ABR (Beyond Array Bounds Read) (1)
- UMR (Uninitialized Memory Read) (1)

```
int
main(int argc, char **argv)

{
FILE *fp;
char *cp1;
char *tmp_ptr;
int i;
char cust_file_path[100];
struct cust_rec *ptr;
struct cust_rec *list_top;

cp1 = (char *)malloc(ARRAY_SIZE);
tmp_ptr = cp1;
for (i=0; i<=ARRAY_SIZE; i++)
        {
        init_array_loc(tmp_ptr);
        tmp_ptr++;
        }

cp1 = (char *)malloc(ARRAY_SIZE);
tmp_ptr = cp1;
for (i=0; i<ARRAY_SIZE-1; i++)
        {
        *tmp_ptr = 'b';
        tmp_ptr++;
        }
```

main > cp1 = (char *)malloc(ARRAY_SIZE)

```
memtest.c
    MemoryLeak  at line 39 cp1 = (char *)malloc(ARRAY_SIZE);
    MemoryLeak  at line 47 cp1 = (char *)malloc(ARRAY_SIZE);
    MemoryLeak  at line 62 cp1 = (char *)malloc(ARRAY_SIZE);
    MemoryLeak  at line 66 cp1 = (char *)malloc(ARRAY_SIZE);
bld_cust_list.c
    MemoryLeak  at line 34 new_cust_ptr = (cust_rec *) malloc (sizeof (cust_rec));
    MemoryLeak  at line 51 new_order_ptr = (order_rec *) malloc (sizeof (order_rec));
```

```
Call Stack
    main at memtest.c:39
    _start in memtest+0x8050c91
```

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.