An Oracle White Paper
June 2010

Optimizing Applications with
Oracle Solaris Studio Compilers and Tools
Introduction

Modern processors and systems provide myriad features and functionality that can dramatically accelerate application performance. The latest high-performance SPARC® and x86 processors provide special enhanced instructions, and the commonality of multicore processors and multisocket systems mean that available system resources are greatly increased. At the same time, applications must still be properly compiled and tuned to effectively exploit this functionality and performance for key applications. Selection of appropriate compiler flags, optimization techniques, and application of appropriate tools is essential to the creation of accurate and performant application code.

Oracle Solaris Studio 12 Update 1 software includes a full-featured integrated development environment (IDE) coupled with compilers and development tools required to produce applications that execute as efficiently as possible – while allowing developers visibility into key aspects of application development, debugging, and operation. Oracle Solaris Studio 12 offers key benefits to developers, allowing them to:

- Accelerate application performance, and
- Simplify multicore development

This document provides an overview of Oracle Solaris Studio software, and offers advice for selecting appropriate compiler flags, options, and development tools. For an exhaustive description of compiler flags and options, please see the complete Oracle Solaris Studio 12 product documentation at

http://docs.sun.com/source/820-3845/stdinfoctr.html
Maximizing application performance is a key goal for any optimizing compiler technology. However, modern application performance must be seen in the context of a diverse and complex mixture of heterogenous hardware and operating systems, as well as and both serial and parallel environments. For example, the latest x86 processors from both Intel® and AMD™ now implement Streaming SIMD Extensions 2 (SSE2) supplemental instructions while some SPARC processors support special instructions that can dramatically increase performance for certain kinds of operations. In addition, all major chip vendors are now producing multicore CPUs, including Intel® Xeon®, AMD Opteron™, and Oracle® SPARC processors.

Oracle Solaris Studio software is designed to allow developers to produce reliable, scalable, and high-performance Oracle Solaris and Linux enterprise applications across all of these diverse platforms. Specifically, Oracle Solaris Studio is designed to:

- Maximize application performance with optimizing compilers
- Simplify multicore development with automatic parallelization features and advanced tools
- Improve productivity with a next-generation IDE and tools with rich graphical interfaces
- Simplify development across multiple architectures (SPARC and x86) as well as multiple operating systems (Oracle Solaris and Linux)

Figure 1 illustrates the components of Oracle Solaris Studio software that allow developers to build, debug, and tune applications, all seamlessly integrated into a next-generation IDE for C, C++, and Fortran developers.
Optimizing Applications with Oracle Solaris Studio Compilers and Tools

Figure 1. Oracle Solaris Studio compilers and tools are seamlessly integrated into a next-generation IDE for C, C++, and Fortran developers.

The Oracle Solaris Studio IDE provides visual development tools, including autocomplete functionality (Figure 2).

Figure 2. Oracle Solaris Studio provides autocomplete functionality.
A screenshot of the Oracle Solaris Studio debugger is provided in Figure 3.

Figure 3. The Oracle Solaris Studio IDE provides an advanced debugger.

As of this writing, Oracle Solaris Studio 12 Update 1 is the latest production release of the Oracle Solaris Studio software. Available on Oracle Solaris and the latest Linux distributions, features of Oracle Solaris Studio include:

- **Optimizing C, C++, and Fortran compilers.** The Oracle Solaris Studio compilers generate improved application performance on Intel x86, AMD x86, UltraSPARC®, and SPARC64® based systems. With a wealth of recent industry-based benchmarks, Oracle Solaris Studio compilers take full advantage of the latest multicore architectures.

- **Full OpenMP 3.0 compiler, debugger, and tools support.** The OpenMP 3.0 specification contains new features to ease multicore development, and takes a more general approach to multithreaded programming by using tasks to support complex and dynamic control flows.

- **DLight.** System profiling tools allow developers to explore their systems, understand how they work, and identify performance problems across many software layers. DLight is a new tool that unifies application profiling and system profiling using DTrace technology on Oracle Solaris platforms.

- **dbxTool.** The dbx Debugger is fully integrated into the IDE and is available via the command line. Oracle Solaris Studio 12 Update 1 now features dbxttool, a stand-alone debugging solution with a user-friendly interface. With dbxttool, developers can quickly and easily debug an executable or core file, or they can attach to a running process.
• **Performance Analyzer support for MPI applications.** The Oracle Solaris Studio Performance Analyzer includes an MPI Timeline, MPI charts, along with zooming and filtering capabilities. With Sun HPC Clustertools, developers can show two new metrics: MPI Work Time and MPI Wait Time.

• **Updated Oracle Solaris Studio IDE.** Oracle Solaris Studio features a next-generation IDE based on NetBeans 6.5.1 software, specifically geared for C/C++ developers. New features include improved code completion, error highlighting, semantic highlighting, call graph, memory window, packaging of application as tar files, zip files, System V Release 4 (SVR4 packages), RPMs, or Debian packages, and much more.

• **Sun Performance Library.** The Sun Performance Library is a set of optimized, high-speed mathematical subroutines for solving linear algebra and other numerically-intensive problems. The library allows developers to increase application performance with enhanced and newly-added standard routines, including BLAS, LAPACK, FFTPACK, SuperLU, Sparse Solvers, and ScaLAPACK.

### Optimizing Applications for Serial Performance

Getting the best performance for SPARC or x86 applications involves using the latest compilers and selecting the best and most appropriate set of compiler options. The sections that follow detail a number of recommended options for optimizing applications for serial performance. Optimizing multithreaded or parallel applications is covered later in this document.

Oracle Solaris Studio compilers strive to provide the best out-of-the-box performance for any applications built using them. However, it is often the case that some minor refinements to the selection of compiler options can yield further gains in performance. As a result, it is key that optimization and tuning be approached on an experimental basis before the final version of the program is released. As a part of this process, it is key to understand exactly what is expected of the compiler in concert with the assumptions made in the application. In particular, two key questions must be asked when selecting appropriate compiler options:

• What is known about the platforms where the compiled application will eventually run?

• What is known about the assumptions that are made in the code?

In addition, it is helpful to consider the purpose of a particular compilation. Compiler options can present various trade-offs depending on whether a given compilation is meant to assist with debugging, testing, tuning, or final performance optimization.

### Identifying the Target Platform

Knowing where the code will eventually run is essential in order to understand what optimization options make sense. The choice of platform determines:

• A 32-bit or 64-bit instruction set

• Instruction set extensions the compiler can use to accelerate performance
• Instruction scheduling depending on instruction execution times

• Cache configuration

Generating 32-bit or 64-bit Code

The UltraSPARC and x86 processor families can run both 32-bit and 64-bit code. The principal advantage of 64-bit code is that the application can handle a larger data set than 32-bit code. However, the cost of this larger address space is a larger memory footprint for the application, since long variable types and pointers increase in size from 4 bytes to 8 bytes. The increase in memory footprint can cause a 64-bit version of an application to run more slowly than the 32-bit version.

At the same time, the x86 platform presents some architectural advantages when running 64-bit code as compared to running 32-bit code. In particular, the application can use more registers, and can use a better calling convention. On the x86 platform, these advantages will typically allow a 64-bit version of an application to run faster than a 32-bit version of the same code, unless the memory footprint of the application has significantly increased.

The UltraSPARC line of processors took a different approach, as it was architected to enable a 32-bit version of an application to use the architectural features of the 64-bit instruction set. As a result, there is no architectural performance gain going from 32-bit to 64-bit code. Consequently, 64-bit applications compiled for UltraSPARC processors will only see the additional cost of the increase in memory footprint.

Compiler flags determine whether a 32-bit or 64-bit binary is generated.

• The `-m32` flag generates a 32-bit binary

• The `-m64` flag generates a 64-bit binary

For additional details about migrating from 32-bit to 64-bit code, please refer to Converting 32-bit Applications Into 64-bit Applications: Things to Consider at http://developers.sun.com/solaris/articles/ILP32toLP64Issues.html and 64-bit x86 Migration, Debugging, and Tuning with the Sun Studio 10 Toolset at http://developers.sun.com/solaris/articles/amd64_migration.html

Specifying an Appropriate Target Processor

Oracle Solaris Studio compilers allow considerable flexibility in selecting a target processor through setting the `-xtarget` compiler flag. The default for the compiler is to produce a “generic” binary – namely a binary that will work well on all platforms (`-xtarget=generic`). In many situations, a generic binary will be the best choice. However, there are some situations where it is appropriate to select a different target, including:

• **To override a previous target setting.** The compiler evaluates options from left to right, and if the flag `-fast` has been specified on the compile line, then it may be appropriate to override the implicit setting of `-xtarget=native` with a different choice.
• To exploit the features of a particular processor. For example, newer processors tend to have more features that can be exploited for performance gains. The compiler can use these features at the expense of producing a binary that does not run on older processors that do not have these features.

The `-xtarget` flag actually sets three flags:

• The `-xarch` flag specifies the architecture of the target machine. This architecture is basically the instruction set that the compiler can use. If the processor that runs the application does not support the appropriate architecture then the application may not run.

• The `-xchip` flag tells the compiler which processor to assume is running the code. This flag tells the compiler which patterns of instructions to favor when it has a choice between multiple ways of coding the same operation. It also tells the compiler the instruction latency to use in order that the instructions are scheduled to minimize stalls.

• The `-xcache` flag tells the compiler the cache hierarchy to assume. This selection can have a significant impact on floating point codes where the compiler is able to make a choice about how to arrange loops so that the data being manipulated fits into the caches.

Target Architectures for the SPARC® Processor Family

For the SPARC processor family, the default setting `-xtarget=generic` should be appropriate for most situations. This setting will generate a 32-bit binary that uses the SPARC V8 instruction set, or a 64-bit binary that uses the SPARC V9 instruction set. The most common situation where the target architecture needs to be taken into account and a different setting may be required is compiling code that contains significant floating point computations.

For example, the SPARC64 VI and VII processors support floating point multiply-accumulate (FMA or FMAC) instructions. These instructions combine a floating point multiply and a floating point addition (or subtraction) into a single operation. A FMA operation typically takes the same number of cycles to complete as either a floating point addition or a floating point multiplication, so the performance gain from using these instructions can be significant. However, it is possible that the results from an application compiled to use FMA instructions may be different than the same application compiled not to use the instructions. In addition, code compiled to take advantage of FMA instructions will not run a platform that does not support those instructions.

As an illustration, consider the operation shown below. The use of the word ROUND in the equation indicates that the value is rounded to the nearest representable floating point number when it is stored into the result.

\[
\text{Result} = \text{ROUND}( (\text{value1} \times \text{value2}) + \text{value3})
\]

The single FMA instruction replaces the following two instructions

```plaintext
\text{tmp} = \text{ROUND}(\text{value1} \times \text{value2})
\text{Result} = \text{ROUND}(\text{tmp} + \text{value3})
```
Notice that the two-instruction version has two round operations, and it is this difference in the number of rounding operations that may result in a difference in the least significant bits of the calculated result. The FMA implemented on the SPARC64 VI processor is referred to as a fused FMA.

To generate FMA instructions, the binary needs to be compiled with the flags:

```
xarch=sparcfmaf -fma=fused
```

Alternatively the flags `-xtarget=sparc64vi -fma=fused` will enable the generation of the FMA instructions and will also tell the compiler to assume the characteristics of the SPARC64 VI processor when compiling the code. This option will produce optimal code for the SPARC64 VI platform. As mentioned, this resulting code will not run on a platform that does not support FMA instructions.

**Target Architectures for the x86 Processor Family**

By default, the Oracle Solaris Studio compiler targets a 32-bit generic x86 based processor, so that generated code will run on any x86 processor from a Pentium Pro to the latest Intel or AMD Opteron processor. While `-xtarget=generic` produces code that can run over the widest range of processors, this code will not take advantage of the SSE2 extensions offered by the latest processors. To exploit these instructions, the flag `-xarch=sse2` can be used. However, the compiler may not recognize all opportunities to use these instructions unless the vectorization flag `-xvector=simd` is also used.

Table 1 provides a summary of Oracle Solaris Studio compiler flags recommended for compilation for various SPARC and x86 target architectures.

**Table 1. Oracle Solaris Studio flags for specifying architecture and address space**

<table>
<thead>
<tr>
<th>ARCHITECTURE</th>
<th>32-BIT ADDRESS SPACE</th>
<th>64-BIT ADDRESS SPACE</th>
</tr>
</thead>
<tbody>
<tr>
<td>SPARC</td>
<td>-xtarget=generic -m32</td>
<td>-xtarget=generic -m64</td>
</tr>
<tr>
<td>SPARC64</td>
<td>-xtarget=sparc64vi -m32 -fma=fused</td>
<td>-xtarget=sparc64vi -m64 -fma=fused</td>
</tr>
<tr>
<td>x86</td>
<td>-xtarget=generic -m32</td>
<td>-xtarget=generic -m64</td>
</tr>
<tr>
<td>X86/SSE2</td>
<td>-xtarget=generic -xarch=sse2 -m32 -xvector=simd</td>
<td>-xtarget=generic -xarch=sse2 -m64 -xvector=simd</td>
</tr>
</tbody>
</table>

**Choosing Compiler Optimization Options**

Choosing compiler options presents a trade-off between compilation time, run-time, and (possibly) application behavior. The optimization flags chosen alter three important characteristics, including:

- The runtime of the compiled application
- The length of time that the compilation takes
- The amount of debug activity that is possible with the final binary.
In general, the higher the level of optimization, the faster the application runs (and the longer it takes to compile), but the less debug information that is available. Ultimately, the particular impact of optimization levels will vary from application to application. The easiest way of thinking about these tradeoffs is to consider three degrees of optimization, as outlined in Table 2.

### Table 2. Three degrees of optimization generate different implications for resulting code.

<table>
<thead>
<tr>
<th>PURPOSE</th>
<th>FLAGS</th>
<th>COMMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full debug</td>
<td>-g [no optimization flags]</td>
<td>The application will have full debug capabilities, but almost no optimization will be performed on the application, leading to lower performance.</td>
</tr>
<tr>
<td>Optimized</td>
<td>-g -O [-g0 for C++]</td>
<td>The application will have good debug capabilities, and a reasonable set of optimizations will be performed on the application, typically leading to significantly better performance.</td>
</tr>
<tr>
<td>High Optimization</td>
<td>-g -fast [-g0 for C++]</td>
<td>The application will have good debug capabilities, and a large set of optimizations will be performed on the application, typically leading to higher performance.</td>
</tr>
</tbody>
</table>

### Compiling for Debugging (-g)

The -g option is a high-fidelity debug option that lets the developer check for algorithmic error. With the flag set, code performs exactly as written and the developer can inspect variables under the debugger. For lower levels of optimization, the -g flag disables some minor optimizations (to make the generated code easier to debug). At higher levels of optimization, the presence of the flag does not alter the code generated (or its performance). However, it is important to be aware that at high levels of optimization, it is not always possible for the debugger to relate the disassembled code to the exact line of source, or for it to determine the value of local variables held in registers rather than stored to memory.

The C++ compiler will disable some of the inlining performed by the compiler when the -g compiler flag is used. For C++, the -g0 flag will tell the compiler to do all the inlining that it would normally perform, as well generating the debug information.

A very strong reason for compiling with the -g flag is that the Oracle Solaris Studio Performance Analyzer can then attribute time spent in the code directly to lines of source code – making the process of finding performance bottlenecks considerably easier.

### Basic Optimization (-O)

Basic optimization can be achieved by using the -O compiler flag. The -O flag offers decent runtime performance, without taking excessively long to compile the application. The -g flag can be added to the -O flag to get optimization with debugging information built in. Multiple possible levels of
optimization are offered with Oracle Solaris Studio compilers, including -O3, -O4, and -O5. Please see the Oracle Solaris Studio documentation for a full description of these options.

Aggressive Optimization (-fast)

The -fast option is a good starting point when optimizing code, but it may not necessarily represent the desired optimizations for the finished program. Developers should note that because the -fast option is defined as a particular selection of compiler options, it is subject to change from one release to another, as well as between compilers. In addition, some of the component options selected by -fast may not be available on some platforms. Care must also be taken if application compilation and linking are performed separately. Developers should make sure that applications are both compiled and linked with -fast to ensure proper behavior.

The -fast option implies many individual compilation optimizations. These individual options can be turned off or on at will. Ideally the -fast option should be applied objectively. For instance, if compiling with -fast yields a five-fold performance gain, it is definitely worth exploring which of the specific options included in -fast are providing the performance advantages. Those options might then be used individually in subsequent builds for a more deterministic and focused optimization.

Developers should be aware of a number of implications for using the -fast compilation flag.

- **Implications for target architecture.** Setting the -fast compiler flag sets -xtarget=native for the compilation. This option detects the native chip and instruction set of the development system, and targets the code for that system. As a result, -xtarget=native should only be used if the target platform is known to be the same as the development system. Otherwise, the -xtarget=generic should be set, or the -xtarget flag should be used to select the desired target architecture.

  For instance, a floating point multiply accumulate (FMA) instruction is implemented on SPARC64 processors, but is not currently implemented on older processors. As a result, a binary that was built on a SPARC64 based system and compiled with -xtarget=native will not run on an older system. The same issue applies to SSE instructions in the Intel x86 architecture that may not be available on older x86 processors and systems.

- **Implications for floating point arithmetic.** The -fast option also includes floating point arithmetic simplifications by setting the -fns and -fsimple flags. The use of -fns and -fsimple can result in significant performance gains. However, these flags may also result in a loss of precision, and they allow the compiler to perform some optimizations that do not comply with the IEEE-754 floating point arithmetic standard. Language standards are also relaxed regarding floating point expression reordering. Before committing to using these flags in production code, any performance gains should be evaluated along with a careful check of results.

  - When setting the -fns flag, subnormal numbers are flushed to zero. Subnormal numbers are very small numbers that are too small to be represented in normal form.
  
  - With -fsimple, the compiler can treat floating point arithmetic as a mathematics textbook might express. For example, assuming that the order in which additions are performed doesn’t matter,
and that it is safe to replace a divide operation by multiplication by the reciprocal. These kinds of assumptions and transformations seem perfectly acceptable when performed on paper, but they can result in a loss of precision when algebra becomes real numerical computation with numbers of limited precision. Also, `-fsimple` allows the compiler to make optimizations that assume that the data used in floating point calculations will not be NaNs (Not a Number). Compiling with `-fsimple` is not recommended if computation with NaNs is expected.

- **Implications for pointer aliasing.** Use of the `-fast` compiler optimization flag asserts that basic types don't alias, so coding assumptions should be checked accordingly. Aliased pointers point to the same region of memory, so an update of a value accessed through one pointer should cause an update of the value accessed through the other pointer. In the following code fraction, if `a` and `b` point to the same (initially zero) memory location then the output should be `a=2 b=2`. However, if the compiler assumes no aliasing, then it could read `a`, read `b`, increment `a`, increment `b`, store `a` back to memory, store `b` back to memory, and then print `a=1 b=1`.

```c
void function (int *a, int *b) {
    *b++;
    *a++;
    printf("a = %i b = %i\n",*a,*b)
}
```

For the compiler, aliasing means that stores to the memory addressed by one pointer may change the memory addressed by the other pointer. As a result, the compiler has to be very careful never to reorder stores and loads in expressions containing pointers, and it may also have to reload the values of memory accessed through pointers after new data is stored into memory. The compiler does not check to see if the assertion is ever violated, so if the code violates the assertion, then it might not behave in the intended fashion. The results generated by the application will be unpredictable if the source code does not adhere to the degree of aliasing allowed by the compiler flags. The following flags tell the compiler what degree of aliasing to assume in the code.

- `-xrestrict` asserts that all pointers passed into functions are restricted pointers. This means that if a function gets two pointers passed into it, under `-xrestrict` the compiler can assume that those two pointers never point at overlapping memory.

- `-xalias_level` indicates what assumptions can be made about the degree of aliasing between two different pointers. `-xalias_level` can be considered a statement about coding style. By using this flag, the developer is informing the compiler how pointers are treated in the coding style employed. For example, the compiler flag `-xalias_level=basic` informs the compiler that a pointer to an integer value will point to the same location as a pointer to a floating point value.

**Additional Optimizations**

In addition to optimization flags, a number of other flags and techniques can be used to increase performance.
Crossfile Optimization (-xipo)

The -xipo option performs interprocedural optimizations over the whole program at link time. Through this approach, object files are examined again at link time to see if there are any further optimization opportunities. The most common opportunity is to inline code from one file into code from another file. The term inlining means that the compiler replaces a call to a routine with the actual code from that routine.

Inlining can be good for two reasons, the most obvious being that it eliminates the overhead of calling another routine. A second, less obvious reason is that inlining may expose additional optimizations that can now be performed on the object code. For example, the following routine calculates the color of a particular point in an image by taking the x and y position of the point and calculating the location of the point in the block of memory containing the image.

```c
int position(int x, int y)
{
    return x + y*row_length;
}

for (x=0; x<100; x++)
{
    value +=array[position(x,y)];
}
```

By inlining that code in the routine that works over all the pixels in the image, the compiler is able to generate code to just add one to the current offset to get to the next point instead of having to do a multiplication and an addition to calculate each address of each point, resulting in a performance gain.

```c
for (x=0; x<100; x++)
{
    value += array[x + y*row_length];
}
```

This code can then be further optimized.

```c
ytmp=y*row_length;
for (x=0; x<100; x++)
{
    value += array[x+ytmp];
}
```

The downside of using -xipo is that it can significantly increase the compile time of the application and may also increase the size of the executable. It is worth compiling with -xipo to see if the increase in compile time is worth the gain in performance.
Profile Feedback (-xprofile=collect, -xprofile=use)

When compiling a program, the compiler makes a best guess at how the flow of the program might proceed – about the branches that are taken and those that not. For floating point intensive code, this approach generally gives good performance. However, for integer programs with many branching operations, taking the compiler’s approximations may not obtain the best performance. Profile feedback assists the compiler in optimizing the application by giving it real information about the paths that are actually taken based on a sample run of the program. Knowing the critical routes through the code allows the compiler to make sure these routes are optimized.

Profile feedback requires compiling a version of the application with the -xprofile=collect flag set, and then running the application with representative input data to collect a runtime performance profile. The program is then recompiled with -xprofile=use combined with the performance profile data that was collected. The downside of this approach is that the compile cycle can be significantly longer, since it comprises two compiles and a run of the application. The upside is that the compiler can produce much more optimal execution paths, yielding a faster runtime for the application.

A representative data set should be one that will exercise the code in ways similar to the actual data that the application will see in production. Additionally, the program can be run multiple times with different workloads to build up the representative data set. Of course if the representative data manages to exercise the code in ways that are not representative of the real workloads, then performance may not be optimal. However, it is often the case that the code is typically executed through similar routes, and so regardless of whether the data is representative or not, the performance will improve. For more information on determining whether a workload is representative see the article Selecting Representative Training Workloads for Profile Feedback Through Coverage and Branch Analysis at http://developers.sun.com/solaris/articles/coverage.html.

Using Large Pages for Data (-xpagesize=size)

If the program manipulates large data sets, then it may benefit from using large pages to hold the data. The idea of a “page” is a region of contiguous physical memory. The processor deals in virtual memory, allowing it the freedom to move the data around in physical memory, or even store it to and load it from disk. Using the concept of pages, the processor has to look up virtual addresses to find the physical location of that data in memory. Every time the processor needs to access a different page in memory, it has to look up the physical location of that page. This operation takes a small amount of time, but if it happens often, the time required can become significant. The default size of these pages is 8 KB for SPARC, and 4 KB for x86 architectures. However, the processor can actually use a range of page sizes. The advantage of using a large page size is that the processor will have to perform fewer lookups. The disadvantage is that the processor may not be able to find a sufficiently large chunk of contiguous memory onto which to allocate the large page (in which case a set of smaller size pages will be allocated instead).

The compiler option that controls page size is -xpagesize=size. The options for the size depend on the platform. On UltraSPARC processors, typical sizes are 8 K, 64 K, 512 K, or 4 MB. For example, changing the page size from 8 K (the default) to 64 K will reduce the number of look-ups by a factor
of eight. On the x86 platform, the default page size is 4 K, and the actual sizes that are available depend on the processor. It is possible to detect performance issues from page sizes using either \texttt{trapstat} (if it is available, and if the processor traps into Oracle Solaris to handle TLB misses), or \texttt{cpustat} (when the processor provides hardware performance counters for TLB miss events).

**Performance Analyzer**

As computer systems continue to become more powerful, application performance is emerging as a critical factor, with bad performance increasingly considered a program failure. Developers are now keenly aware that they must streamline critical sections of source code as well as locate programmatic errors and coding deficiencies without impacting application accuracy. Oracle Solaris Studio includes a Performance Analyzer that can help aid developers with these tasks.

To use the Performance Analyzer, applications can be compiled with any level of parallelization and optimization. To see source code, and to attribute time to lines of source code, the \texttt{-g} option must also be specified. Applications are then run using the \texttt{collect} command. The command can specify a PID,

\begin{verbatim}
% collect -P <pid>
\end{verbatim}

or the collect command can be used to launch the application and its parameters.

\begin{verbatim}
% collect <application> <parameters>
\end{verbatim}

The collect command gathers performance data during application execution, saving it to an experiment file to be used later during the analysis process. The \texttt{collect} command enables developers to obtain information on:

- Clock-based profiles
- Thread-synchronization delay events and wait time
- Operating system summary information
- Hardware-counter overflow profiles on systems where the hardware supports it
- Global information, including execution statistics and address-space data

Once the experiment is complete, the Performance Analyzer loads the experiment data from a file titled \texttt{test.1.er}. Experiments can be either loaded into the analyzer from the command line, or by using the \texttt{<File>} menu from the running analyzer application. To start the analyzer, the following is typed on the command line.

\begin{verbatim}
% analyzer <control-options> <experiment-list>
\end{verbatim}
To aid application analysis, the Performance Analyzer then provides several ways for developers to view collected performance data, including data display at the function or load object level. Developers can control which metrics are shown, as well as the order in which they appear.

Functions Tab

The Functions Tab (Figure 4) shows a list of functions and their metrics. The metrics are derived from the data collected in the experiment. Metrics can be either exclusive or inclusive. Exclusive metrics represent usage within the function itself while inclusive metrics represent usage within the function, and all of the functions it called.

The Callers-Callees Tab

The Callers-Callees tab shows the selected function in a pane in the center, with callers of that function in a pane above, and callees of that function in a pane below (Figure 5). For the selected function, the attributed metric represents the time attributed to that function. For the callees, the attributed metric represents the portion of the callee’s inclusive metric that is attributable to calls from the center function.
Figure 5. The Caller-Callee tab shows attributed time related to selected functions.

Disassembly Tab

The optional Disassembly Tab (Figure 6) shows the annotated source in an upper pane, and the annotated disassembly in a lower pane.

Figure 6. The Disassembly Tab can be invoked to show disassembled code.
Source Tab
If the code was compiled with the `-g` option, the source of a selected function can be viewed with annotations of performance metrics for each source line along with compiler commentary (Figure 7).

```
/* ...
#include <stdio.h>

int main() {
    int x, y;
    x = 2 * x + y;
    y = x + y;
    return (x + y);
}
*/
```

Figure 7. The Source Tab couples performance metrics with each source line.

Timeline Tab
The Timeline Tab (Figure 8) allows viewing the application timeline and call stack for selected events.
### Optimizing Applications with Oracle Solaris Studio Compilers and Tools

Figure 8. The Timeline Tab graphically illustrates the application timeline and call stack.

Other tabs are available and descriptions can be found in the Oracle Studio Performance Analyzer documentation at [http://docs.sun.com/app/docs/doc/819-5264/](http://docs.sun.com/app/docs/doc/819-5264/).

### Optimizing Parallel Applications

Most processors today – SPARC and x86 alike – are equipped with multiple cores and are capable of supporting multiple simultaneous execution threads. Many systems also employ multiple multicore processors. Taking advantage of these multiple cores and exploiting multiple threads of execution has become important as organizations seek to derive as much value and performance as possible from their selected platforms.

The Oracle Solaris operating system provides an efficient and scalable threading model as well as a smart scheduler to deliver these considerable resources to applications through a variety of application development and deployment tools.

- Virtualization systems such as Oracle VM and Oracle VM for SPARC let multiple operating system instances share a single physical system.
- Threaded Oracle Solaris Containers allow multiple execution environments within a single operating system instance.
- Threaded applications can take advantage of multiple cores on multicore processors and multisocket systems.

Independent of the execution environment, as developers seek to exploit parallelism, they must ensure that their code is correct and provides predictable results. Oracle Solaris Studio compilers support...
techniques for generating parallel applications, including automatic parallelization, support for OpenMP directives, and support for the POSIX threads API. The Oracle Solaris Studio Thread Analyzer is also provided to help analyze parallel code for correctness.

Automatic Parallelization

Many existing codes were written without the assumption of parallel threads of execution. Oracle Solaris Studio compilers provide mechanisms to let the application run multiple threads without requiring the developer to specify how. Loops in particular often represent opportunities where a previously repetitive serial operation can be divided into multiple independent execution threads. Several compiler flags are used with Oracle Solaris Studio compilers to govern automatic parallelization behavior.

- The `-xautopar` compiler flag tells the compiler to look for loops that can be safely parallelized in the code.
- The `-xreduction` compiler flag can be used to recognize and parallelize reduction operations that take a range of values and output a single value – such as summing all the values in an array.
- The `-xloopinfo` compiler flag can be specified to generate information for the developer about the loops that the compiler has parallelized.

OpenMP

Support for OpenMP in Oracle Solaris Studio means that the compilers can look for directives (pragma) in the source code in order to build a parallel version of the application. Similar to automatic parallelization, the compiler does the work so that the developer doesn’t have to manage their own threads. OpenMP represents an incremental approach to parallelization with potentially fine granularity. OpenMP allows developers to set directives around specific loops to be optimized through threading while leaving other loops untouched. The other distinct advantage of this approach is that developers can derive a serial and a parallel version of the application from the exact same code base, which can be helpful for debugging. Several compiler flags are used with Oracle Solaris Studio related to OpenMP.

- OpenMP is enabled by the `-xopenmp` compiler flag and directives are only recognized when the flag is used.
- The `-xvpara` compiler flag reports potential parallelization issues.
- The `-xloopinfo` compiler flag tells the compiler to provide the details of which loops were parallelized.
- The `OMP_NUM_THREADS` environment variable must be set by the user at runtime, and it controls the number of desired threads for codes that are parallelized using OpenMP and automatic parallelization.
POSIX Pthreads

By programming to the POSIX threads API, developers can have complete control over thread usage in their applications. POSIX Threads (or Pthreads) represents a POSIX standard for a thread API – defining a set of C programming language types, functions, and constants. Oracle Solaris Studio compilers support the POSIX threads programming model.

Thread Analyzer

While the Performance Analyzer provides an advanced tool for application optimization, the Thread Analyzer is designed to help ensure multithreaded application correctness. Specifically, the Thread Analyzer can help detect, analyze, and debug the special situations that can arise in multithreaded applications.

- **Data races** can cause incorrect or unpredictable results, and can occur arbitrarily far way from where a problem seems to occur. Data races occur under the following conditions:
  - Two or more threads in a single process concurrently access the same memory location
  - At least one of the threads is accessing the memory location for writing
  - The threads are not using any exclusive locks to control their accesses to that memory

- **Deadlock conditions** occur when one thread is blocked waiting on a resource held by a second thread, while the second thread is blocked waiting on a resource held by the first (or an equivalent situation with more threads involved).

To instrument the source code for data race and deadlock detection the code is compiled with a special flag, executed under control of the `collect -r` command, and then loaded into the Thread Analyzer.

- Applications are first compiled with the `-xinstrument=datarace` compiler flag. It is recommended that the `-g` flag also be set, and that no optimization level be used to help ensure that the line numbers and call-stacks information is returned correctly.

- Resulting application code is then executed within the `collect -r` command allowing for the collection of key runtime information. Use the `collect -r all` option to run the program and create a data race detection and deadlock detection experiment during the execution of the process. Alternately, either data races or deadlock conditions for the experiment.

  ```
  % collect -r race <app> <params>
  % collect -r deadlock <app> <params>
  ```

- Finally, the results of the experiment are loaded into the Thread Analyzer to identify data race and deadlock conditions (Figure 9).
The Thread Analyzer can also help identify individual lines of source code that are associated with race conditions (Figure 10).
Sample Optimizations in Practice

As stated, optimization is an incremental process where different optimizations are evaluated against the advantages they provide. Those optimizations that make a substantial performance difference are then noted as candidates for building the final executable application. As an example of various tuning options, this section considers a simple program that calculates the Mandelbrot set. This entire code for this application follows.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#define SIZE 4000
int ** data;

int ** setup()
{
    int i;
    int **data;
    data=(int**)malloc(sizeof(int*)*SIZE);
    for (i=0; i<SIZE; i++)
    {
        data[i]=(int*)malloc(sizeof(int)*SIZE);
    }
    return data;
}

int inset(double ix, double iy)
{
    int iterations=0;
    double x=ix, y=iy, x2=x*x, y2=y*y;
    while ((x2+y2<4) && (iterations<1000))
    {
        y = 2 * x * y + iy;
        x = x2 - y2 + ix;
        x2 = x * x;
        y2 = y * y;
        iterations++;
    }
    return iterations;
}
```
void loop()
{
    int x, y;
    double xv, yv;
    #pragma omp parallel for private(y,xv,yv) schedule(guided)
    for (x=0; x<SIZE; x++)
    {
        for (y=0; y<SIZE; y++)
        {
            xv = ((double)(x-SIZE/2))/(double)(SIZE/4);
            yv = ((double)(y-SIZE/2))/(double)(SIZE/4);
            data[x][y] = inset(xv, yv);
        }
    }
    if (data[7][7]<0) {printf("Error");}
}

void main()
{
    data = setup();
    loop();
}

To determine a baseline, the application is first compiled using the \texttt{-g}, \texttt{-O}, and \texttt{-xtarget=generic} compiler flags. Timings for the application runtime are provided below.

\begin{verbatim}
% cc -g -O -xtarget=generic mandle.c
% timex ./a.out
real  33.02
user  32.88
sys   0.09
\end{verbatim}

Because the development system in this case was based on the x86 architecture, it made sense to specify the use of SSE2 instructions to see if using those instructions can provide an additional performance advantage. Note that \texttt{-xtarget=native} would produce the same result in this case since the \texttt{-xarch=sse2} flag would be implied.

\begin{verbatim}
% cc -g -O -xarch=sse2 mandle.c
% timex ./a.out
real  12.05
user  11.92
sys   0.08
\end{verbatim}
In this case, the code runs nearly three times faster using SSE2 instructions, compared to when the compiler is told not to generate them. Fortunately, most x86 processors now support SSE2 instructions so it is relatively safe to assume that the bulk of the available hardware will support them.

Next the `-xopenmp` flag is set to trigger use of the OpenMP directive that delineates the for loop that does the computation work of the Mandelbrot computation. The `-xvpara` and `-xloopinfo` flags are specified to generate information on what loops are parallelized, and report any potential issues.

```
% cc -g -O -xopenmp -xvpara -xloopinfo mandle.c
"mandle.c", line 13: not parallelized, call may be unsafe
"mandle.c", line 25: not parallelized, loop has multiple exits
"mandle.c", line 41: PARALLELIZED, user pragma used
"mandle.c", line 43: not parallelized, loop inside OpenMP region
```

The resulting code is then run with the environment variable `OMP_NUM_THREADS` set equal to two.

```
% export OMP_NUM_THREADS=2
% timex ./a.out
real           8.72
user          11.92
sys            0.08
```

In this case, it is important to note that the user time is the same (11.92 seconds) because the same amount of work is performed. However, the real (or wall-clock) time is reduced because there are now two threads performing the work. Unfortunately, the performance doesn't double because the work is unbalanced between the two threads. One thread finishes first, so the performance improvement is limited by the slower thread. This behavior can be checked by collecting a profile using the Performance Analyzer and looking at the timeline view, as shown in Figure 11.
Figure 11. Viewing the timeline in the Performance Analyzer reveals that the thread workload is unbalanced. If the OpenMP directive is changed to include schedule(guided) and the application is recompiled, the runtime performance improves even further, dropping from a high of over 33 seconds to less than seven seconds.

```bash
% timex ./a.out
real    6.90
user    11.94
sys     0.08
```

Figure 12 illustrates the balanced workload the final completed Mandelbrot set.
Conclusion

With improvements in technology, application developers have new opportunities to optimize and tune applications. Developers need to be able to exploit technology advancements at the processor level as well as leveraging the resources provided by multicore processors and multiprocessor systems. At the same time, they must ensure that their code executes correctly across the broadest set of intended target platforms.

Oracle Solaris Studio software provides a proven set of compilers and tools that offer C, C++, and Fortran developers the flexibility and power they need to develop correct and performant applications. Oracle Solaris Studio compilers offer optimization options that allow developers to tune their applications to take advantage of specific platform advantages. Combined with key compiler options, the Oracle Solaris Studio Performance Analyzer helps collect data about how applications actually perform while providing a highly visual tuning experience while the Thread Analyzer helps developers develop correct multithreaded code.

Together these tools can harness the considerable resources available in modern processors and computing platforms, producing highly-interactive applications and the fastest available execution times for applications running on Oracle Solaris and Linux platforms.