An Archived Oracle Technical Paper
April 2011

The System Developer's Edge

Selected Blog Posts and Articles

Compiled by Darryl Gove

Contents

Preface

The *System Developer's Edge* is a reprint of a book originally published in 2010 under the title *The Developer's Edge*.

Introduction to the 2011 Reprint

The Developer's Edge was envisioned as an almanac for developers, something that gathered together a set of useful resources that could be dipped into, referred to, or read cover-to-cover.

The book was never intended as the only location where the information resided, however some of the content is no longer available elsewhere, making it fortuitous that it has been captured here.

While the main body of text has not been changed, the book has been updated to the Oracle brand. The title has changed to include the word "systems", to target the intended audience more clearly.

With all that said, welcome to the *Systems Developer's Edge*.

Overview

The System Developer's Edge: Selected Blog Posts and Articles focuses on articles in the following areas:

- Native language issues
- Performance and improving performance
- Specific features of the x86 and SPARC processors
- Tools that are available in the Oracle Solaris OS

The articles provide a broad overview on a topic, or an in-depth discussion. The texts should provide insights into new tools or new methods, or perhaps the opportunity to review a known domain in a new light.

Why This Book?

As developers, we are fortunate in that the answer to most questions is a search engine away. It is usually a short trip from the compile time warning message or runtime error to a description of the problem and, perhaps more importantly, the solution. The solutions are documented in blog posts, on wikis, in the documentation, in posting to newsgroups, or in one of a myriad of channels of communication. Not only are the issues documented there is also an element of immediacy. Other people may have encountered the same issue only yesterday, but their workaround is already described. It is a huge advantage to have access to all this information.

However, the downside of this is that context is often lost. For example, finding out which compiler flag you need to accomplish a task might solve your immediate problem but it doesn't always provide the information about what differentiates that flag from others, which would help you solve similar problems in the future. Although searching online gives you a very direct path from a problem to a solution, you do not gain the additional knowledge about a topic that browsing through a book about the subject often provides.

This book is a selection from the wealth of material posted on `http://blogs.sun.com` and the developer sites on `http://www.sun.com`. The book captures some of the developer-focused information presented on these sites, and organizes the material into common themes. The objective is to both capture the content and place it into a wider context. The book provides both the critical insights that are needed to solve problems and the surrounding details that facilitate a broader awareness of the issues.

How This Book Is Organized

The book contains the following information.

Chapter 1, "General Topics," covers general topics such as the STREAM benchmark, or Amdahl's law.

Chapter 2, "Compilers," covers the traditional domain of compilers, compiler flags, and optimization, and also includes a trilogy of articles on profile feedback.

Chapter 3, "Hardware-Specific Topics," covers some hardware-specific material, such as the use of atomic operators, and the x86 calling convention.

Chapter 4, "Solaris Performance Primer," contains an impressive series of blog entries by Stefan Schneider and Thomas Bastian that cover the wealth of tools that are available for inspecting system and application performance.

Chapter 5, "Developer Tools," continues on the topic of tools by describing those that are available as part of the Sun Studio compiler collection.

Chapter 6, "Libraries and Linking," covers libraries and the linker, topics such as how to cause an application to pick up a particular library or a different library depending on the capabilities of the hardware.

Chapter 7, "Floating Point," discusses some topics around floating point, such as subnormal numbers.

Chapter 8, "Parallel Code," contains some articles on the topic of parallelization and OpenMP.

Typographic Conventions

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

TABLE P–1 Typographic Conventions

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

Acknowledgements

This book is the work of multiple authors: Chris Aoki, Thomas Bastian, Steve Clamage, Nawal Copty, Rod Evans, Richard Friedman, Alfred Huang, Yuan Lin, Giri Mandalika, Richard Marejka, Fintan Ryan, Stefan Schneider, Lawrence Spracklen, Geetha Vallabhaneni, and Douglas Walls. A number of people have contributed to producing this final text. Janice Gelb edited the text, working to produce clarity whilst maintaining the original voice of the authors. Miriam Blatt and Karsten Guthridge once again provided insightful reviews. Zee Zee Matta created the cover art. Jeff Gardiner marshalled the resources for the project.

◆ ◆ ◆ CHAPTER 1

1 General Topics

This chapter pulls together various posts about topics which are more general in nature, applying to multiple languages or multiple platforms. These range from discussions of how to identify the compiler version used to build an executable or shared object to details about how large pages can be used to improve application performance.

Why Scalar Optimization Is Important

Richard Friedman, May 3, 2006

I've heard people say that they weren't interested in learning how to optimize the scalar parts of their programs. All they were interested in was learning how to parallelize the code.

Well, that's pretty foolish. And Amdahl's law proves it.

Basically, the performance of a program distributed over a multiprocessor system is always limited by the fraction of the code that doesn't run in parallel.

Lets say we have N processors available to us. If we run the program on one processor, it runs T seconds. Theoretically if we were to run it 100% over N processors, it should take T/N seconds (wall-clock time).

But what if only a fraction f<1.0 of the executed code runs in parallel over the N processors. So the total wall clock time is (1-f)T + fT/N. The fraction of the non-parallel part plus the fraction of the parallel part. The speed up is the time with one processor over the time with N processors, or (I hope I've done this right...)

```
S = T/((1-f)T+fT/N) = 1/(1-f+f/N)
```

(Hmm. Let's check. If f=1, S=N meaning Nx speedup on N processors if it runs 100% in parallel. And as f goes towards 0, we get S=1, as expected: as we approach 100% serial code with f=0, S goes towards 1, meaning no speedup. So far so good.)

So let's try to figure out what fraction of the executable code we need to run in parallel to get a specific speedup with N processors. Solving for f, we get (after some ordinary algebraic manipulations)

```
f = (1-1/S)/(1-1/N)    for N>1
```

So let's say we have 64 processors. We'd like to a speedup of 64, but that would mean that 100% of the code has to run in parallel, theoretically. So let's be modest and say we'd like a speedup of 32. What would f be?

Well, that gives f=.984 So, theoretically 98% of the executed code has to run in parallel over the 64 processors to get a 32x speedup.

Work it the other way around: if I can get half the executable code to run in parallel, and the other half running in one processor, what speedup can I expect on a 64–processor system?

Well, that gives a speedup S = 1.97 ... not even 2! (Theoretically. We have made some really gross assumptions -- like the code runs only in 1 or 64 processors, etc.; so this is really worst case.)

Still, the moral of the story: If you can't get your code over 90% parallelized, then you'd better pay attention to the part that is running in only one processor, and make sure that (serial) part is optimized. Because it will dominate performance.

This is the kind of stuff you learn in the Performance Optimization course described in this blog entry (`http://blogs.sun.com/rchrd/entry/performance_optimization_class_may_30`). Serial performance matters, even on multiprocessor systems! So it pays to look at all the performance options the compilers offer (`http://developers.sun.com/solaris/articles/options.html`).

Lots of hand waving here: to be accurate, the real time is the sum of the time spent by the fraction that runs on one processor, plus the time spent by the fraction that runs on two processors, plus... and so on up to the time spent by the fraction that runs on all N processors.

But let's just take the case where there are only two processors. Now the speedup is:

```
S = 1/(1-f/2)
```

And if f=.5, the best you can do is a 1.33 speedup. If f=.9, you approach 1.82. So still you need to look carefully at the serial code unless you can be sure that more than 90% of the code is running concurrently on both processors.

(I'm sure someone will find some problem with this logic, but I'm using some gross simplifications. Obviously there is more to it than this. But you get the point.)

It's the law (`http://en.wikipedia.org/wiki/Amdahl%27s_law`).

`http://blogs.sun.com/rchrd/entry/why_scalar_optimization_is_important`

32-Bit Good, 64-Bit Better?

Darryl Gove, March 14, 2008

One of the questions people ask is when to develop 64-bit apps vs 32-bit apps. The answer is not totally clear cut, it depends on the application and the platform. So here's my take on it.

First, let's discuss SPARC. The SPARC V8 architecture (`http://www.sparc.org/standards/V8.pdf`) defines the 32-bit SPARC ISA (Instruction Set Architecture). This was found on a selection of SPARC processors that appeared before the UltraSPARC I, that is, quite a long time ago. Later the SPARC V9 architecture (`http://www.sparc.org/standards/SPARCV9.pdf`) appeared. (As an aside, these are open standards, so anyone can download the specs and make one, of course not everyone has the time and resources to do that, but it's nice in theory ;-))

The SPARC V9 architecture added a few instructions, but mainly added the capability to use 64-bit addressing. The ABI (Application Binary Interface) was also improved (floating-point values passed in FP registers rather than the odd V8 choice of using the integer registers). The UltraSPARC I and onwards have implemented the V9 architecture, which means that they can execute both V8 and V9 binaries.

One of the things that it's easy to get taken in by is the *Animal Farm* idea that if 32-bit is good, 64-bit must be better. The trouble with 64-bit address space is that it takes more instructions to set up an address, pointers go from 4 bytes to 8 bytes, and the memory footprint of the application increases.

A hybrid mode was also defined, which took the 32-bit address space, together with a selection of the new instructions. This was called v8plus, or more recently sparcvis (`http://blogs.sun.com/quenelle/entry/goodbye_xarch_amd64_hello_m64`). This has been the default architecture for SPARC binaries for quite a while now, and it combines the smaller memory footprint of SPARC V8 with the more recent instructions from SPARC V9. For applications that don't require 64-bit address space, v8plus or sparcv is the way to go.

Moving to the x86 side, things are slightly more complex. The high-level view is similar. You have the x86 ISA, or IA32 as it's been called. Then you have the 64-bit ISA, called AMD64 or EMT64. EMT64 gives you both 64-bit addressing, a new ABI, a number of new instructions, and perhaps most importantly a bundle of new registers. The original x86 architecture has impressively few registers, and EMT64 fixes that quite nicely.

In the same way as SPARC, moving to a 64-bit address space does cost some performance due to the increased memory footprint. However, the x64 gains a number of additional registers, which usually more than make up for this loss in performance. So the general rule for x86 is that 64 bits is better, unless the application makes extensive use of pointers.

Unlike SPARC, EMT64 does not currently provide a mode which gives the instruction set extensions and registers with a 32-bit address space.

`http://blogs.sun.com/d/entry/32_bits_good_64_bits`

What About Binary Compatibility?

Douglas Walls, December 4, 2006

- What will happen if I try to run my application on a newer Solaris OS release?
- What will happen if I try to compile my application using one version of the Sun Studio compilers and link it with libraries compiled with earlier compiler versions?

Important questions to consider, and luckily both the Solaris OS and the Sun Studio compilers guarantee a certain degree of binary compatibility between releases.

Here are some things to keep in mind:

- *Binary Compatibility among versions of the Solaris OS:*

  Executables created with a supported compiler release on an earlier Solaris OS version will run on later Solaris OS versions. This is part of the Solaris binary compatibility guarantee. Such executables might need the Sun Studio runtime libraries that are part of the compiler with which the executable was created. For example, an executable created using Forte Developer 6 update 2 on Solaris 2.6, along with the shared runtime libraries of Forte Developer 6 update 2, will run correctly without recompilation or relinking on Solaris 10.

- *Binary Compatibility among object files:*

  Libraries and object files created with an earlier release of a Sun Studio compiler can be used when linking with object files created by a later version of that compiler. When you link with a mixed set of object files and libraries created with different versions of the Sun Studio compilers, you must use the *latest* compiler that produced any of the object files or libraries being linked. For example, a shared object (`.so`) file created with a Forte Developer Update 2 compiler can be used when linking with the Sun Studio 11 version of that same compiler.

Specific Questions I Get Often

- Can I compile my application on Solaris 10 and run it on Solaris 9 and Solaris 8?

  No. Might work, but since you compiled it on Solaris 10, it might also be using system interfaces that did not exist on Solaris 8 and 9 or have changed in Solaris 10.

- Can I compile my application on Solaris 8 and run it on Solaris 9 and Solaris 10?

  Yes! This is what binary compatibility is all about. (See above)

- Can I compile and build my shared library on Solaris 10 and use it on Solaris 9 and Solaris 8?

  No. Might work, but since you compiled it on Solaris 10, it might also be using system interfaces that did not exist on Solaris 8 and 9 or have changed in Solaris 10.

- If I compile the code in my shared library using the Sun Studio 11 compilers, can my customers who are still using Forte 6 Update 1 compilers link with these shared libraries?

No. You must always link with the same compiler used to create the newest objects in your application or library. So, if Sun Studio 11 compilers are used to compile the code in a shared library, Sun Studio 11 compilers must be used when linking with that shared library.

http://blogs.sun.com/dew/entry/what_about_binary_compatibility%3F

Compiler Versions

Darryl Gove, May 19, 2008

If you need to find out which version of the compiler is installed, use:

```
$ cc -V
cc: Sun C 5.9 SunOS_sparc 2007/05/03
```

The following table shows the mappings of version numbers to recent released compilers.

TABLE 1–1 Compiler Version Numbers

Product Name	C/C++ Version Number
Sun Studio 12	5.9
Sun Studio 11	5.8
Sun Studio 10	5.7
Sun Studio 9	5.6

A more interesting question is which compiler generated an executable.

```
$ mcs -p test.o
test.o:
acomp: Sun C 5.8 2005/10/13
as: Sun Compiler Common 11 2005/10/13
```

The test file was generated by Sun Studio 11.

Finally, which flags were used to generate an executable. Use `dwarfdump` and `grep` for "command" for binaries generated with Sun Studio 12, or for C code compiled with Sun Studio 11.

```
$ dwarfdump test.o|grep command
DW_AT_SUN_command_line  /opt/SUNWspro/prod/bin/cc -xtarget=generic64 -c  test.c
<   13>   DW_AT_SUN_command_line      DW_FORM_string
```

For older compilers use `dumpstabs` and `grep` for "CMD":

```
$ dumpstabs getr.o|grep CMD
2:  .stabs "/export/home; /opt/SUNWspro/bin/../prod/bin/cc -c -O
getr.c",N_CMDLINE,0x0,0x0,0x0
```

Of course it's very unlikely that shipped binaries will contain information about compiler flags.

http://blogs.sun.com/d/entry/compiler_versions

Single-Precision Floating-Point Constants

Darryl Gove, July 19, 2007

The C standard specifies that by default, floating-point constants should be held as double precision values. Calculations where one variable is single precision and one variable is double precision need to be computed as double precision. So, it's very easy to have code which works entirely on single precision variables but ends up with lots of conversion between single and double precision because the constants are not specified as single precision. An example of this is shown in the next code snippet.

```
float f(float a)
{
  return a*1.5;
}
% cc -O -S ff.c
/* 0x0008            */         ldd     [%o5+%lo(offset)],%f4
/* 0x000c            */         ld      [%sp+68],%f2
/* 0x0010            */         fstod   %f2,%f6
/* 0x0014            */         fmuld   %f6,%f4,%f8
/* 0x0018            */         retl    ! Result =  %f0
/* 0x001c            */         fdtos   %f8,%f0
```

The `fstod` instruction converts the variable a from single precision to double precision, the `fdtos` converts the result of the multiplication back to single precision.

The option to specify that unsuffixed floating point constants are held as single precision rather than the default of double precision is -xsfpconst (http://docs.sun.com/source/819-3688/cc_ops.app.html#pgfId-1001438). This option works at file level, so it is not ideal for situations where files are a mix of single–precision and double-precision constants. This flag can cause the opposite problem where the single-precision constant needs to be converted to double precision in order to be used in a calculation with a double-precision variable.

The issue is easy to work around at a source code level. Single-precision constants can be specified with a trailing f. For example:

```
float f(float a)
{
  return a*1.5f;
}
```

Compiling this code produces the following much neater and more efficient code:

```
/* 0x0008            */         ld      [%o5+%lo(offset)],%f2
/* 0x000c            */         ld      [%sp+68],%f4
```

```
/* 0x0010           */         retl    ! Result =  %f0
/* 0x0014           */         fmuls   %f4,%f2,%f0
```

http://blogs.sun.com/d/entry/single_precision_floating_point_constants

Resolving Problems With Creating Reproducible Binaries Using Sun Studio Compilers

Douglas Walls, July 24, 2006

Certain flags (`-g, -xipo, -xcrossfile`) invoke features of the Sun Studio Compilers and tools that must make static data global. To avoid namespace collisions, the data is globalized using a unique globalization prefix. The drawback of using a unique globalization prefix is that the resulting object files will be different every time the source file is compiled, even though the sources and compilation options are identical. So, if you compare the resulting object files from two identical compilations they will differ.

If you are in an environment that requires the ability to reproduce identical object files, this becomes a problem.

Both C and C++ have an undocumented flag (`-xglobalstatic`) that can be passed to the compiler front-end which will force the use of a static globalization prefix based on the source filepath, instead of the usual algorithm that guarantees a unique prefix. To pass the flag to cc, use the `-W0` option as follows:

```
cc -W0,-xglobalstatic
```

To pass the flag to CC use the `-Qoption ccfe` option as follows:

```
CC -Qoption ccfe -xglobalstatic
```

The drawback to using the filepath to generate the globalization prefix is the increased risk of a namespace collision at link time between static data with the same name that has been globalized from two files with the same filepaths. Though rare, it does occur more often than using a randomly generated prefix.

If you do run into namespace collisions, you might need to assign the globalization prefix. For C this can be done with the wizard option `-W0,-xp<prefix>`, for example:

```
-W0,-xp\$XAqalkBBa5_D2Mo
```

For C++ this can be done with the wizard options `-Qoption ccfe -prefix -Qoption ccfe <prefix>`, for example:

```
-Qoption ccfe -prefix -Qoption ccfe \$XA0ZlkBtDTxEGkV.
```

http://blogs.sun.com/dew/entry/resolving_problems_creating_reproducible_binaries

volatile Keyword

Darryl Gove, June 12, 2007

The keyword volatile should be used in situations where an item of data is shared between multiple threads. An example of this is the code:

```
int ready;
...
  while (ready);
...
```

This code gets compiled into:

```
/* 0x0004            */         ld      [%o5+%lo(ready)],%o5
/* 0x0008            */         cmp     %o5,0
/* 0x000c            */         be,pn   %icc,.L77000022
/* 0x0010            */         nop

                         .L77000017:
/* 0x0014          5 */         ba      .L77000017
/* 0x0018            */         nop

                         .L77000022:
/* 0x001c          5 */         retl    ! Result =
/* 0x0020            */         nop
```

Compare this to the situation where the variable is declared as volatile:

```
                         .L900000106:
/* 0x0018          5 */         cmp     %o5,0
/* 0x001c            */         be,pn   %icc,.L77000022
/* 0x0020            */         nop


                         .L77000017:
/* 0x0024          5 */         ld      [%o2],%o3 ! volatile
/* 0x0028            */         cmp     %o3,0
/* 0x002c            */         bne,a,pt        %icc,.L900000106
/* 0x0030            */         ld      [%o2],%o5 ! volatile

                         .L77000022:
/* 0x0034          5 */         retl    ! Result =
/* 0x0038            */         nop
```

The endless loop has been removed, the variable is now polled.

The situation is interesting when using a pointer:

```
volatile int * v1;
int * volatile v2;

...
  while (*v1);
  while (*v2);
...
```

The two pointers are treated differently

```
                        .L900000109:
/* 0x001c          6 */         cmp     %o5,0
/* 0x0020            */         be,pn   %icc,.L900000110
/* 0x0024          0 */         sethi   %hi(v2),%g5

                        .L77000026:
/* 0x0028          6 */         ld      [%o4],%g1 ! volatile LOAD *V1
/* 0x002c            */         cmp     %g1,0
/* 0x0030            */         bne,a,pt        %icc,.L900000109
/* 0x0034            */         ld      [%o4],%o5 ! volatile LOAD *V1
...

                        .L900000108:
/* 0x0054          7 */         ld      [%o5],%o4 Load *V2
/* 0x0058            */         cmp     %o4,0
/* 0x005c            */         be,pn   %icc,.L77000039
/* 0x0060            */         nop

                        .L77000034:
/* 0x0064          7 */         ld      [%g2],%o1 ! volatile LOAD V2
/* 0x0068            */         ld      [%o1],%o0 LOAD *V2
/* 0x006c            */         cmp     %o0,0
/* 0x0070            */         bne,a,pt        %icc,.L900000108
/* 0x0074            */         ld      [%g2],%o5 ! volatile LOAD V2
...
```

v1 is a pointer to a volatile int, whereas v2 is a volatile pointer to an int. So v2 requires both the pointer and the data it points to to be reloaded, v1 only reloads the data being pointed to.

http://blogs.sun.com/d/entry/volatile

Finding the Canonical Path to an Executable

Douglas Walls, May 8, 2006

Below is a coding example of how an executable can determine the canonical path to itself on the file system. Compiler drivers, like cc, CC, f95 need to do this in order to execute the component executables that compile a program, for example, the compiler front-end, an optimizer, and a linker.

```
% cat findself.c
#include <stdio.h>
#include <errno.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>

#ifndef MAXPATHLEN
#define MAXPATHLEN      1024
#endif
```

```
/* find_run_directory - find executable file in PATH
 * PARAMETERS:
 *      cmd     filename as typed by user
 *      cwd     where to return working directory
 *      dir     where to return program's directory
 *      run     where to return final resolution name
 * RETURNS:
 *      returns zero for success,
 *      -1 for error (with errno set properly).
 */
int find_run_directory (char *cmd, char *cwd, char *dir, char **run)
{
    char                *s;
    if (!cmd || !*cmd || !cwd || !dir) {

errno = EINVAL;         /* stupid arguments! */
        return -1;
    }
    if (*cwd != '/')
        if (getcwd (cwd, MAXPATHLEN - 1) == NULL )

return -1;          /* can't get working directory */
    if (strchr (cmd, '/') != NULL) {
        if (realpath(cmd, dir) == NULL) {
            int lerrno = errno;
            if (chdir((const char *)cwd) == NULL)
                errno = lerrno;
            return -1;
        }
    } else {
#ifdef __linux__
        /* getexecname() not available on Linux */
        if (readlink("/proc/self/exe", dir, MAXPATHLEN) == -1) {
#else
        if (realpath(getexecname(), dir) == NULL) {
#endif
            int lerrno = errno;
            if (chdir((const char *)cwd) == NULL)
                errno = lerrno;
            return -1;
        }
    }
    s = strrchr (dir, '/');
    *s++ = 0;
    if (run)            /* user wants resolution name */
        *run = s;
    return 0;
}
char current_working_directory[MAXPATHLEN];
char run_directory[MAXPATHLEN];
char * run_exec_name = NULL;
int main(int argc, char **argv)
{
    if ( !find_run_directory (argv[0],
                              current_working_directory,
                             run_directory, &run_exec_name) ) {
        (void)printf("argv[0] = %s\n"
                         "cwd = %s\n"
```

```
                        "run_dir = %s\n"
                        "run_exec = %s\n",
                    argv[0],
                    current_working_directory,
                    run_directory,
                    run_exec_name);
    } else {
        (void) printf("%s\n", "Unaable to find run directory.");
    }
    exit (0);
}

% cc findself.c -O -o prod/bin/findself
% ls
bin         findself.c  prod
% ls bin
findself
% ls -laF bin
total 6
drwxr-xr-x   2 me      staff        512 Mar  5 10:37 ./
drwxr-xr-x   4 me      staff        512 Mar  5 10:37 ../
lrwxrwxrwx   1 me      staff         20 Mar  5 10:37 findself -> ../prod/bin/findself*
% ls -laF prod/bin
total 22
drwxr-xr-x   2 me      staff        512 Mar  5 10:37 ./
drwxr-xr-x   3 me      staff        512 Mar  5 10:37 ../
-rwxr-xr-x   1 me      staff       8992 Mar  5 10:37 findself*
% bin/findself
argv[0] = bin/findself
cwd = /home/me/blog
run_dir = /home/me/blog/prod/bin
run_exec = findself
% prod/bin/findself
argv[0] = prod/bin/findself
cwd = /home/me/blog
run_dir = /home/me/blog/prod/bin
run_exec = findself
% setenv PATH /home/me/bin:${PATH}
% findself
argv[0] = findself
cwd = /home/me/blog
run_dir = /home/me/blog/prod/bin
run_exec = findself
```

http://blogs.sun.com/dew/entry/finding_the_canonical_path_to

STREAM and the Performance Impact of Compiler Optimization

Fintan Ryan, September 13, 2007

Introduction

A couple of weeks back there was a discussion on the perf-discuss (http://www.opensolaris.org/jive/thread.jspa?threadID=36191[amp]tstart=0) alias over on opensolaris.org (http://www.opensolaris.org/os/)around memory bandwidth/throughput benchmarks, and as is customary in such a discussion STREAM (http://www.cs.virginia.edu/stream/) got a mention. During the discussion, a couple of suggestions were made regarding various compiler options to use with STREAM. Seeing as we gather some of this data on an ongoing basis as part of Sun's Performance Lifestyle (http://blogs.sun.com/fintanr/entry/enabling_suns_performance_lifestyle), I decided at the time to throw in a couple of extra experiments so that we can demonstrate the kind of impact that various compiler options can have.

Now in order to keep the test in the realms of reproducibility for most people (and of course within the bounds of not annoying other people by taking up valuable test cycles on some of our larger machines – I would have liked to have given one of our SunFire X4600's a run with all of these experiments, but they are incredibly highly utilized by the various engineering groups we work with) the numbers which I based this post on were generated on a SunFire X2100 M2 server, which is a single CPU, dual-core AMD64 box. The ARRAY_SIZE for our problems was set using the l2 cache size script (http://blogs.sun.com/fintanr/entry/quick_grab_of_l2_cache) mentioned previously.

Secondly, as mentioned before, my group does not generate benchmark numbers for publication. In general we run STREAM with `-xopenmp -fast` as a more out-of-the-box type compilation. However, here we are taking a baseline with no compiler options passed in, which consists of twenty iterations of stream, calling that our 100% point, and finally expressing our results as a percentage of our baseline. Now with all of that aside, let's move onto the experiments.

Experiment Details

The rig, as mentioned above is a SunFire X2100 , 1 x 2400Mhz M2 chip, 1Gb of ram. The OS used is the latest version of Solaris Developer Express (Nevada 70b for those following from OpenSolaris). The compiler is Sun Studio 12 , 2007/05.

The compiler options used here do not represent an exhaustive comparison of the various compiler options, but rather a more general indications of the kind of optimization that Studio 12 can do, and the impact that your compiler can have on the performance of your application. It should be noted though that STREAM is a benchmark which is highly suited to being optimized. To further improve performance, we use OpenMP. We have two tables of results below, one with `OMP_NUM_THREADS` set to the core count (two in this case) and one with `OMP_NUM_THREADS` set to the physical processor count (one in this case).

The Results

The data here is pretty self explanatory, the higher the number the better. Of the experiments we did here, the most optimal options were `-fast -xopenmp -xvector=simd -xprefetch -xprefetch_level=3` and running with our environment set to include `OMP_NUM_THREADS=2`.

TABLE 1–2 One OpenMP Thread

metric	no options	`-fast`	`-fast -xopenmp`	`-fast -xopenmp -xvector=simd`	`-fast -xopenmp -xvector=simd -xprefetch -xprefetch_level=3`	`-fast -xvector=simd -xprefetch -xprefetch_level=3`
add	100.00%	118.50%	117.04%	130.08%	188.93%	187.18%
copy	100.00%	126.39%	124.94%	217.72%	214.05%	214.68%
scale	100.00%	123.97%	122.48%	214.20%	210.21%	212.20%
triad	100.00%	118.34%	116.67%	129.84%	186.03%	188.86%

TABLE 1–3 Two OpenMP Threads

metric	no options	`-fast`	`-fast -xopenmp`	`-fast -xopenmp -xvector=simd`	`-fast -xopenmp -xvector=simd -xprefetch -xprefetch_level=3`	`-fast -xvector=simd -xprefetch -xprefetch_level=3`
add	100.00%	118.50%	177.71%	245.01%	301.24%	187.33%
copy	100.00%	126.44%	181.02%	323.97%	324.35%	214.55%
scale	100.00%	124.10%	175.78%	319.18%	319.12%	212.20%
triad	100.00%	118.41%	177.07%	245.63%	298.91%	188.94%

Compiler Options

The compiler options used here represent a small subset of what you can do with Sun Studio 12, and are covered in a lot more detail in the extensive documentation (`http://developers.sun.com/sunstudio/documentation/product/compiler.jsp`). Most of the options used above are self explanatory, but the two that may be of interest are `-xvector` and `-xprefetch`.

TABLE 1–4 Compiler Flags

Flag	Comment
`-xvector=simd`	Instructs the compiler to use SIMD (Single Instruction Markup Data). Basically this allows us to deal with several chunks of data in one operation rather than multiple ones. STREAM is heavily vector oriented, so this gives us a sizeable performance gain.
`-xprefetch -xprefetch_level=3`	This option enables prefetching, at the highest level the compiler supports. Prefetching is a mechanism by which data is speculatively fetched from memory into the CPU cache. Certain processor architectures (for example, SPARC, and in this case AMD64) will do an amount of prefetching, but you can instruct the compiler to insert even more prefetch instructions. In the case of STREAM we are processing large arrays which lends itself very well to this kind of optimization, but it's one that should be used with some caution.

Further Reading

The compiler folks are continuously publishing new articles (`http://developers.sun.com/sunstudio/documentation/techart/index.jsp`) which contain various tips and suggestions on how to get the most out of your compiler which are well worth reading It's also worth signing up to the Sun Developer Network (`https://reg.sun.com/register?program=sdn`) to get the free downloads of Studio 12.

`http://blogs.sun.com/fintanr/entry/stream_and_the_performance_impact`

On Sun Studio and gcc Style

Alfred Huang, May 22, 2006

A question has been raised on the optimization effect of Sun Studio's inline assembly mechanism and that of `gcc` style enhanced inline assembly. Let's start with a brief introduction of what they are first and then discuss the optimization effect they may have.

At the simplest level, both compilers support the `asm()` statement, which is the insert-as-is non-optimized inline assembly. This form of inline assembly simply inserts the enclosed assembly strings as-is without any facility of argument and optimization.

Sun Studio supports the `.il` inline template in a form similar to an include file. An `.il` file may contain multiple inline templates where each template is of the following form:

```
.inline "template_name",0
   "assembly code"
.end
```

In a sense, Sun Studio treats each inline template as a function definition and adheres argument passing and return values according to the calling convention in the ABI of the corresponding platform.

For example, let's add 8 numbers together and return its result. The inline template in the `foo.il` file may be as follows:

```
.inline multi_add,0
movq    %rdi, %rax
addq    %rsi, %rax
addq    %rdx, %rax
addq    %rcx, %rax
addq    %r8,  %rax
addq    %r9,  %rax
addq    (%rsp),%rax
addq    8(%rsp), %rax
.end
```

The `foo.c` file may look like:

```
int foo(int,int,int,int,int,int,int,int);
int multi_add()
{
   return foo(1,2,3,4,5,6,7,8);
}
```

Compile the `foo.c` file with `foo.il`:

```
cc -S -O -xarch=amd64 foo.il foo.c
```

The resulting `foo.s` will contain an optimized result:

```
foo:  ...
/ ASM INLINE BEGIN:   multi_add
      movq   $36, %rax
/ ASM INLINE END
```

Based on the calling convention of the AMD64 ABI, the first six integral arguments are passed in `%rdi`, `%rsi`, `%rdx`, `%rcx`, `%r8` and `%r9`, any extras are to be passed on the stack, hence (`%rsp`) and 8(`%rsp`) in this case. If optimization is not desired, a `-Wu,-no_a2lf` option can be used:

```
cc -S -O -xarch=amd64 -Wu,-no_a2lf foo.il foo.c
```

Thus the resulting `foo.s` will contain the following:

```
    push       $8
    push       $7
    movq       $1,%rdi
    movq       $2,%rsi
    movq       $3,%rdx
    movq       $4,%rcx
    movq       $5,%r8
    movq       $6,%r9
```

```
        / INLINE: multi_add
        movq    %rdi, %rax
        addq    %rsi, %rax
        addq    %rdx, %rax
        addq    %rcx, %rax
        addq    %r8, %rax
        addq    %r9, %rax
        addq    (%rsp), %rax
        addq    0x8(%rsp), %rax
        / INLINE_END
```

Sun Studio treats an inline template as a function call, therefore function argument loading will take place before the template body is inserted. If `-Wu,-no_a2lf` is not used, all these assembly instructions are inserted into Sun Studio's intermediate representation stream and all specified optimization will take place, result in a single "`movq $36, %rax`".

In this manner, the user-specified inline template will be assimilated into Sun Studio's optimization and code generation mechanism. Further note that all registers specified in the inline template will be "virtualized" and reallocated by the code generator. Therefore, the final appearance of the inline template may be drastically different from its original. The main catch of the approach is that users must understand the calling convention of the underlying platform.

GCC style extended `asm` inline assembly takes a different approach. Basically it has the form:

```
    asm("template",
        "input arguments",
        "output arguments",
        "clobber list");
```

The approach provides flexibility for the user to specify the input and output arguments and allows the user to inform the compiler of the resource used within the template, thus allowing the compiler to avoid resource conflict. Other than that, the template is treated as a black box with its content unexamined. Hence, the argument-substituted template body will be inserted asis and only limited optimizations will be performed.

For example, let's add 3 constants and a static variable together with the following `asm()`:

```
int mem;
int foo()
{
        int res;

        asm ("addl %1, %0\n"
             "\taddl %2, %0\n"
             "\taddl %3, %0\n"
             "\taddl %4, %0\n"
             : "=r" (res)
             :  "g" (1),
                "g" (2),
                "g" (3),
                "m" (mem)
```

```
            );

        return res;
}
```

And the result will simply be:

```
.globl foo
        .type   foo, @function
foo:
/APP
        addl $1, %edx
        addl $2, %edx
        addl $3, %edx
        addl mem, %edx
/NO_APP
        movl    %edx, %eax
        ret
```

Note the template body will be inserted as-is after the argument substitution into the final assemblyonly. But some forms of optimization may still be performed. For example the entire template may be moved out of a loop if it turns out to be a constant invariant when the "clobber list" indicates no change in memory and register.

Both styles of inline assembly mechanism have their pros and cons, it depends on one's needs. It is possible to convert them from one style to the other.

http://blogs.sun.com/alblog/entry/on_studio_and_gcc_style

Atomic Operations in the Solaris 10 OS

Darryl Gove, January 11, 2007

Atomic operations (http://docs.sun.com/app/docs/doc/816-5168/6mbb3hr46?a=view) are available in Solaris 10 through the <atomic.h> header file (and libc). A short example is as follows.

```
% more atom.c
#include <atomic.h>
volatile unsigned int test;

void main()
{
  test=0;
  for (int i=0; i<10000; i++){atomic_add_int(&test,i);}
}
% cc -O atom.c
% a.out
```

Here's the disassembly for atomic_add_int (http://docs.sun.com/app/docs/doc/816-5168/6mbb3hr1k?a=view) from libc:

```
atomic_add_int()
    2dc18:    ld            [%o0], %o2
    2dc1c:    add           %o2, %o1, %o3  <---|
    2dc20:    cas           [%o0] , %o2, %o3   |
    2dc24:    cmp           %o2, %o3           |
    2dc28:    bne,a,pn      %icc, 0x2dc1c  ----|
    2dc2c:    mov           %o3, %o2       ----| [delay slot]
    2dc30:    retl
    2dc34:    add           %o2, %o1, %o0
```

The idea of atomic functions is that they complete without anything else being able to change the variable during the operation - as if the operation were a single step. They are extremely useful when data is shared between threads

The basic idea, as can be seen in the add code, is that the variable is loaded, the add is performed, the value stored back (using the atomic instruction 'cas' - compare and swap). Then the result is checked to see whether it worked or not. If it didn't work the operation is repeated.

The `cas` instruction performs the test that if the value held at [%o0] is equal to the value %o2, then replace it with the value %o3. %o3 returns the value that was in [%o0] when the operation was tried.

In the SPARC assembly language, the instruction in the delay slot of the branch gets executed together with the branch instruction.

`http://blogs.sun.com/d/entry/atomic_operations_in_solaris_10`

Atomic Operations

Darryl Gove, July 29, 2008

One of the questions from the presentation last week was how to use atomic operations in Solaris 9 if they are only available on Solaris 10. The answer is to write your own. The following code snippet demonstrates how to write an atomic increment operation:

```
.inline atomic_add,8
  ld [%o0],%o2         /* Load existing value            */
1:
  add %o2, %o1, %o3    /* Generate new value             */
  cas [%o0],%o2,%o3    /* Swap memory contents           */
  cmp %o2, %o3         /* Compare old value with return  */
  bne 1b               /* Fail if the old value does not */
                       /* equal the value returned from  */
                       /* memory                         */
  mov %o3,%o2          /* Retry using latest value from  */
                       /* memory                         */
.end
```

It's probably a good idea to read "Atomic SPARC: Using the SPARC Atomic Instructions" on page 122, and it may also be useful to read "Using Inline Templates to Improve Application Performance" on page 74.

http://blogs.sun.com/d/entry/atomic_operations

The Cost of Mutexes

Darryl Gove, July 29, 2008

Another question that came up last week was about the performance of mutex locks compared to the performance of atomic operations. The basic cost is acquiring and freeing a mutex takes two operations whereas an atomic operation is a single call. There's also a bit more code required when using mutexes. The following code demonstrates the cost of calling mutexes, atomic operations, and also an inline template (http://blogs.sun.com/d/entry/atomic_operations).

```
#include <pthread.h>
#include <atomic.h>
#include "timing.h"

#define SIZE 10000000

pthread_mutex_t mutex;
pthread_t thread;
volatile unsigned int counter;

void atomic_add(volatile unsigned int *,int);

void * count(void* value)
{
  counter=0;
  starttime();
  while (counter<SIZE)
  {
    pthread_mutex_lock(&mutex);
    counter++;
    pthread_mutex_unlock(&mutex);
  }
  endtime(SIZE);
  counter=0;
  starttime();
  while (counter<SIZE)
  {
    atomic_add_int(&counter,1);
  }
  endtime(SIZE);
  counter=0;
  starttime();
  while (counter<SIZE)
  {
    atomic_add(&counter,1);
  }
  endtime(SIZE);
}

void main()
```

```
{
  pthread_mutex_init(&mutex,0);
  counter=0;
  pthread_create(&thread,0,count,0);

  pthread_join(thread,0);
  pthread_mutex_destroy(&mutex);
}
```

Compiling and running an UltraSPARC T1 gives results like:

```
% cc test.c add.il
% a.out
Time per iteration 250.61 ns
Time per iteration 75.85 ns
Time per iteration 65.21 ns
```

So the mutex calls are about 3x slower than atomic operations. Calling `libc` is about 10ns slower than using an inline template (not a bad difference in return for not having to write the inline template code).

It's interesting to see where the time goes. So here's the profile of the application:

```
Excl.     Incl.      Name
User CPU  User CPU
 sec.      sec.
3.973     3.973      <total>

1.341     3.973      count
1.331     1.331      mutex_unlock
0.781     0.781      mutex_lock_impl
0.490     0.490      atomic_add_32
0.030     0.030      mutex_lock
```

The routine `mutex_lock` tail calls `mutex_lock_impl`, which does the work of locking the mutex. The heart of `mutex_unlock` looks like:

```
   0.        0.                [?]    beff8:  mov         %o1, %o3
   0.020     0.020             [?]    beffc:  cas         [%o0] , %o2, %o3
## 0.560     0.560             [?]    bf000:  cmp         %o2, %o3
   0.        0.                [?]    bf004:  bne,a       0xbeff8
   0.        0.                [?]    bf008:  mov         %o3, %o2
```

The core of `mutex_lock_impl` is not too dissimilar, so basically the mutex lock code contains two atomic operation loops, plus a bundle of other instructions that make up the rest of the cost of the calls.

Looking at where the time is spent for the `atomic_add_32` call:

```
   0.010     0.010             [?]    2ecb8:  ld          [%o0], %o2
   0.040     0.040             [?]    2ecbc:  add         %o2, %o1, %o3
   0.010     0.010             [?]    2ecc0:  cas         [%o0] , %o2, %o3
## 0.370     0.370             [?]    2ecc4:  cmp         %o2, %o3
   0.        0.                [?]    2ecc8:  bne,a,pn    %icc,0x2ecbc
```

```
   0.        0.                [?]    2eccc:  mov         %o3, %o2
   0.050     0.050             [?]    2ecd0:  retl
   0.010     0.010             [?]    2ecd4:  add         %o2, %o1, %o0
```

Which is again, a very similar loop, but with little overhead around it. And it pretty much matches the code that came from the inline template:

```
   0.040     0.040             [?]    110ec:  add         %o2, %o1, %o3
   0.010     0.010             [?]    110f0:  cas         [%o0] , %o2, %o3
## 0.360     0.360             [?]    110f4:  cmp         %o2, %o3
   0.        0.                [?]    110f8:  bne         0x110ec
   0.040     0.040             [?]    110fc:  mov         %o3, %o2
```

http://blogs.sun.com/d/entry/the_cost_of_mutexes

Using Large DTLB Page Sizes

Darryl Gove, April 18, 2007

The TLB is a structure on the chip that handles the mapping of virtual memory addresses (used by the application) into physical memory addresses (used by the hardware). It is a list of such mappings, with each mapping describing a range of memory (called the page size). The default on SPARC is 8KB page sizes, but it can be configured up to impressively large sizes (for example, 256MB for UltraSPARC T1). The command to display what page sizes the hardware supports is `pagesize`(1):

```
pagesize -a
```

If the application requests a virtual to physical translation that is not mapped in the TLB, then there's a TLB miss. On UltraSPARC III/IV the process of fetching a TLB entry takes about a hundred cycles.

Using a larger page size will reduce the number of TLB misses. Of course a large page size requires a large chunk of contiguous physical memory, and it's not always possible to get this.

An application can request large pages in one of three ways:

- Using the `ppgsz`(1) command to set the preferred page sizes.
- Using the compiler flag `-xpagesize` to set the preferred page size at compile time.
- Preloading the `mpss.so.1`(1) library and using the `MPSSHEAP` and `MPSSSTACK` environment variables to describe the page size.

When an application is running, it is possible to inspect the page sizes of the allocated memory using the command:

```
pmap -xs <pid>
```

http://blogs.sun.com/d/entry/using_large_dtlb_page_sizes

Page Size and Memory Layout

Darryl Gove, February 7, 2008

Support for large pages has been available since Solaris 9. I've previously talked about the various ways that an application can be coaxed into using large pages (http://blogs.sun.com/d/entry/using_large_dtlb_page_sizes). However, I wanted to quickly write up how the large pages are laid out in memory. Take the following code that allocates a large chunk of memory, and then iterates over it for enough time to run `pmap -xs` on it:

```
#include <stdlib.h>

void main()
{
  int x,y;
  char *c;
  c=(char*)malloc(sizeof(char)*300000000);
  for (y=0; y<; y++)
  for (x=0; x<300000000; x++) { c[x]=c[x]+y;}
}
```

Compiling this code to use 4MB pages and then running the resulting executable produces a `pmap` output like:

```
% cc -xpagesize=4M t.c
% a.out&
[1] 15501
% pmap -xs 15501
15501:  a.out
 Address  Kbytes     RSS    Anon  Locked Pgsz Mode   Mapped File
00010000       8       8       -       -   8K r-x--  a.out
00020000       8       8       8       -   8K rwx--  a.out
00022000    3960    3960    3960       -   8K rwx--    [ heap ]
00400000  290816  290816  290816       -   4M rwx--    [ heap ]
...
```

Notice that the heap starts on 8KB pages, and uses these up until the memory reaches a 4MB boundary and then starts using 4MB pages. In this case it means that nearly 4MB of the memory is not using 4MB pages - if this happens to be where the majority of the program's active data resides, then there will still be plenty of TLB misses.

Fortunately, it is possible to tell the linker where to start the heap. There are some mapfiles provided in `/usr/lib/ld/` for various scenarios. The one that we need is `map.bssalign`. Recompiling with this produces the following memory layout:

```
% cc -M /usr/lib/ld/map.bssalign -xpagesize=4M t.c
% a.out&
[1] 19077
% pmap -xs 19077
19077:  a.out
 Address  Kbytes     RSS    Anon  Locked Pgsz Mode   Mapped File
00010000       8       8       -       -   8K r-x--  a.out
```

```
00020000       8       8       8       -   8K rwx--  a.out
00400000  294912  294912  294912       -   4M rwx--    [ heap ]
```

With this change the heap now starts on a 4MB boundary and is entirely mapped with 4MB pages.

http://blogs.sun.com/d/entry/page_size_and_memory_layout

◆ ◆ ◆ CHAPTER 2

Compilers

This chapter is focused on compiler flags. Discussions include guidance on how to select the best compiler flags, checking code for security issues, and a collection of articles on the best practices for using profile feedback.

Selecting the Best Compiler Options

Darryl Gove, June 2008

This article suggests how to get the best performance from an UltraSPARC or x86/EMT64 (x64) processor running on the latest Solaris systems by compiling with the best set of compiler options and the latest compilers. These are suggestions of things you should try, but before you release the final version of your program, you should understand exactly what you have asked the compiler to do.

The Fundamental Questions

There are two questions that you need to ask when compiling your program:

- What do I know about the platforms that this program will run on?
- What do I know about the assumptions that are made in the code?

The answers to these two questions determine what compiler options you should use.

The Target Platform

What platforms do you expect your code to run on? The choice of platform determines:

- 32-bit or 64-bit instruction set
- Instruction set extensions the compiler can use

- Instruction scheduling depending on instruction latency
- Cache configuration

The first three are often the most important ones.

32-bit Versus 64-bit Code

The UltraSPARC and x64 families of processors can run both 32-bit and 64-bit code. The main 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; long variable types and pointers increase in size from 4 bytes to 8 bytes. The increase in footprint will cause the 64-bit application to run more slowly than the 32-bit version.

However, the x86/x64 platform has some architectural advantages when running 64-bit code compared to running 32-bit code. In particular, the application can use more registers, and can use a better calling convention. These advantages will typically enable 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 was architected to enable the 32-bit version of the application to already use the architectural features of the 64-bit instruction set. So there is no architectural performance gain going from 32-bit to 64-bit code. Consequently the UltraSPARC processors will only see the additional cost of the increase in memory footprint.

The compiler flags that determine whether a 32-bit or 64-bit binary is generated are the flags `-m32` and `-m64`.

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

Specifying an Appropriate Target Processor

The default for the compiler is to produce a “generic” binary; a binary that will work well on all platforms. In many situations this will be the best choice. However, there are some situations where it is appropriate to select a different target.

- To override a previous target setting. The compiler evaluates options from left to right, 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 take advantage of features of a particular processor. For example, newer processors tend to have more features. The compiler can use these features at the expense of producing a binary that does not run on the older processors that do not have these features.

The `-xtarget` flag actually sets three flags:

- The `-xarch` flag which specifies the architecture of the machine. This 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 which tells the compiler which processor to assume is running the code. This 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 so that the instructions are scheduled to minimize stalls.
- The `-xcache` flag tells the compiler which cache hierarchy to assume. This 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 SPARC Processors

The default setting `-xtarget=generic` should be appropriate for many situations. This 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 a different setting may be required is compiling a code containing significant floating point computations so that the resulting binary uses the floating point multiply-accumulate (FMA or FMAC) instructions.

The SPARC64 VI processors support FMA instructions. These instructions combine a floating point multiply and a floating point addition (or subtraction) into a single operation. A FMA 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.

An FMAC instruction performs the following operation, 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.

```
Result = ROUND( (value1 * value2) + value3)
```

The single instruction replaces the following two instructions

```
tmp = ROUND(value1 * value2)
Result = ROUND(tmp + 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. It is possible to have an unfused FMA, which implements the multiply accumulate operation in a single instruction but produces a result which is identical to the one that would be produced by the two separate instructions.

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 instruction and will also tell the compiler to assume the characteristics of the SPARC64 VI processor when compiling the code. This will produce optimal code for the SPARC64 VI platform.

Specifying the Target Processor for the x64 Processor Family

By default the compiler targets a 32-bit generic x86 based processor, so the code will run on any x86 processor from a Pentium Pro up to an AMD Opteron architecture. Whilst this produces code that can run over the widest range of processors, this does not take advantage of the extensions offered by the latest processors. Most currently available x86 processors have the SSE2 instruction set extensions. To take advantage of 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.

Summary of Target Settings for Various Address Spaces and Architectures

The following table contains a list of settings to use for the various processors and architectures.

TABLE 2–1 Architectures and Compiler Flags

Address Space	SPARC	SPARC64	x86	x64/sse2
32-bit	`-xtarget=generic -m32`	`-xtarget=sparc64vi -m32 -fma=fused`	`-xtarget=generic -m32`	`-xtarget=generic -xarch=sse2 -m32 -xvector=simd`
64-bit	`-xtarget=generic -m64`	`-xtarget=sparc64vi -m64 -fma=fused`	`-xtarget=generic -m64`	`-xtarget=generic -xarch=sse2 -m64 -xvector=simd`

Optimization and Debug

The optimization flags chosen alter three important characteristics: the runtime of the compiled application, the length of time that the compilation takes, and the amount of debug 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; but the particular impact of optimization levels will vary from application to application.

The easiest way of thinking about this is to consider three degrees of optimization, as outlined in the following table.

TABLE 2–2 Debug Flags

Purpose	Flags	Comments
Full debug	[no optimization flags] `-g`	The application will have full debug capabilities, but almost no optimization will be performed on the application, leading to lower performance.
Optimized	`-g -O` [`-g0` for C++]	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.
High optimization	`-g -fast` [`-g0` for C++]	The application will have good debug capabilities, and a large set of optimizations will be performed on the application, typically leading to higher performance.

Note: For C++ the debug flag `-g` will inhibit some of the inlining of methods, while the flag `-g0` will provide debug information without inhibiting the inlining of these methods. Consequently, it is recommend that for higher levels of optimization `-g0` be used instead of `-g`.

Suggestion: In general an optimization level of at least `-O` is suggested. However, the two situations where lower levels might be considered are (i) where more detailed debug information is required and (ii) the semantics of the program require that all variables be treated as volatile, in which case the optimization level should be lowered to `-xO2`.

More Details on Debug Information

The compiler will generate information for the debugger if the `-g` flag is present. 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) -- but 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.

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

A very strong reason for compiling with the `-g` flag is that the Sun 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.

Suggestion

- Always compile with `-g`/`-g0` since it should not make much (if any) difference to performance. Your program will be easier to debug and analyze.
- On x86 platforms, the `-xregs=frameptr` allows the compiler to use the framepointer as an unallocated callee-saves register, which can result in increased runtime performance. This option is included in `-fast` for C. Use of the flag may mean that some tools are unable to correctly generate callstack information.
- `-fast` is a good starting point when optimizing code. However, it may not necessarily be the set of optimizations you want for the finished program. It is a good idea to use the `-#`, `-xdryrun`, or `-V` options to print out the options that `-fast` includes, and to select the appropriate ones for your application from this list.

Refer to Comparing the -fast Option Expansion on x86 Platforms and SPARC Platforms (`http://developers.sun.com/solaris/articles/amd64_migration.html#fast`) for the expansion of `-fast` by Sun Studio 10 C, C++, and Fortran compilers, `cc`, `CC`, and `f95`, respectively.

The Implications for Floating-Point Arithmetic When Using the `-fast` Option

One issue to be aware of is the inclusion of floating-point arithmetic simplifications in `-fast`. In particular, the options `-fns` and `-fsimple=2` allow the compiler to do some optimizations that do not comply with the IEEE-754 floating-point arithmetic standard, and also allow the compiler to relax language standards regarding floating-point expression reordering.

With the flag `-fns`, subnormal numbers (that is, very small numbers that are too small to be represented in normal form) are flushed to zero.

With `-fsimple=2`, the compiler can treat floating-point arithmetic as a mathematics textbook might express it. For example, the order additions are performed doesn't matter, and it is safe to replace a divide operation by multiplication by the reciprocal. These kinds of 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 you expect computation with *NaNs*.

Notes

- The use of the flags `-fns` and `-fsimple=2` can result in significant performance gains. However, they may also result in a loss of precision. Before committing to using them in production code, it is best to evaluate the performance gain you get from using the flags, and whether there is any difference in the results of the application.
- Avoid using `-fsimple=2` with applications that perform calculations on *NaNs*.
- For more information on floating-point computation, see the Sun Studio 12: Numerical Computation Guide (`http://docs.sun.com/doc/819-5269`).

Crossfile Optimization

The `-xipo` option performs interprocedural optimizations over the whole program at link time. This means that the object files are examined again at link time to see if there are any further optimization opportunities. The most common opportunity is to inline a routine 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 is 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, imagine that a 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 (`image_offset = y * row_length + x`). By inlining that code in the routine that iterates over all the pixels in the image, the compiler is able generate code to just increment 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.

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.

Suggestion:

- Try compiling with `-xipo` to see if the increase in compile time is worth the gain in performance.

Profile Feedback

When compiling a program, the compiler takes a best guess at how the flow of the program might go -- which branches are taken and which branches are not taken. For floating-point intensive code, this generally gives good performance. But programs with many branching operations might not obtain the best performance.

Profile feedback assists the compiler in optimizing your application by giving it real information about the paths actually taken by your program. Knowing the critical routes through the code allows the compiler to make sure these are the optimized ones.

Profile feedback requires that you compile and execute a version of your application built with `-xprofile=collect` and then run the application with representative input data to collect a runtime performance profile. You then recompile with `-xprofile=use` and the performance profile data collected. The downside of doing this is that the compile cycle can be significantly longer (you are doing two compiles and a run of your application), but the compiler can produce much more optimal execution paths, which means a faster runtime.

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; 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 which are not representative of the real workloads, then performance may not be optimal. However, it is often the case that the code is always 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, read the article Selecting Representative Training Workloads for Profile Feedback Through Coverage and Branch Analysis (`http://developers.sun.com/solaris/articles/coverage.html`).

Suggestion:

- Try compiling with profile feedback and see whether the performance gain is worth the additional compile time.
- Try compiling with profile feedback and `-xipo`, because the profile information will also help the compiler make better choices about inlining.

Using Large Pages for Data

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

The compiler option which controls page size is `-xpagesize=<size>`. The options for the size depend on the platform. On UltraSPARC processors, typical sizes are 8K, 64K, 512K, or 4M. For example, changing the page size from 8K (the default) to 64K will reduce the number of look

ups by a factor of 8. On the x86 platform, the default page size is 4K, and the actual sizes that are available depend on the processor. It is possible to detect performance issues from page sizes using either `trapstat`, if it is available, and if the processor traps into Solaris to handle TLB misses, or `cpustat` when the processor provides hardware performance counters that could TLB-miss events.

Advanced Compiler Options: C/C++ Pointer Aliasing

There are two flags that you can use to make assertions about the use of pointers in your program. These flags will tell the compiler something that it can assume about the use of pointers in your source. It does not check to see if the assertion is ever violated, so if your code violates the assertion, then your program might not behave in the way you intended it to. Note that `lint` can help you do some validity checking of the code at a particular `-xalias_level`. (See Chapter 4, "lint Source Code Checker," in *Sun Studio 12: C User's Guide* (`http://docs.sun.com/doc/819-5265/bjafs?a=view`).)

The two assertions are:

- `-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 - you are telling the compiler how you treat pointers in the coding style you use (for example, you can tell the compiler that an `int*` will never point to the same memory location as a `float*`).

A useful piece of terminology is the expression "alias". Two pointers alias if they point to the same location in memory. The flags `-xrestrict` and `-xalias_level` tell the compiler what degree of aliasing to assume in the code. For the compiler, aliasing means that stores to the memory addressed by one pointer may change the memory addressed by the other pointer -- this means that 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 following table summarizes the options for `-xalias_level` for C (cc).

TABLE 2–3 Alias Level Settings for C

cc -xalias_level=	Comment
any	Any pointers can alias (default)
basic	Basic types do not alias each other (for example, int* and float*)
weak	Structure pointers alias by offset. Structure members of the same type at the same offset (in bytes) from the structure pointer, may alias.
layout	Structure pointers alias by common fields. If the first few fields of two structure pointers have identical types, then they may potentially alias.
strict	Pointers to structures with different variable types in them do not alias
std	Pointers to differently named structures do not alias (so even if all the elements in the structures have the same types, if they have different names, then the structures do not alias).
strong	There are no pointers to the interiors of structures and char* is considered a basic type. (At lower levels char* is considered as potentially aliasing with any other pointers.)

The following table summarizes the options for -xalias_level for C++ (CC).

TABLE 2–4 Alias Level Settings for C++

CC -xalias_level=	Comment
any	Any pointers can alias (default)
simple	Basic types do not alias (same as basic for C)
compatible	Corresponds to layout for C

Notes

- Specifying -xrestrict and -xalias_level can lead to significant performance gains. But if your code does not conform to the requirements of the flags, then the results of running the application may be unpredictable.
- For C, -xalias_level=std means that pointers behave in the same way as the 1999 ISO C standard suggests. Specified for standard-conforming codes.

A Set of Flags to Try

The final thing to do is to pull all these points together to make a suggestion for a good set of flags. Remember that this set of flags may not actually be appropriate for your application, but it is hoped that they will give you a good starting point. (Use of the flags in square brackets, [..] depends on special circumstances.)

TABLE 2–5 Suggested Flags

Flags	Comment
-g	Generate debugging information (may use `-g0` for C++)
`-fast`	Aggressive optimization
`-xtarget=generic [-xtarget=sparc64vi -fma=fused] [-xarch=sse -xvector=simd]`	Specify target platform
`-xipo`	Enable interprocedural optimization
`-xprofile=[collect|use]`	Compile with profile feedback
`[-fsimple=0 -fns=no]`	No floating-point arithmetic optimizations. Use if IEEE-754 compliance is important
`[-xalias_level=<level>]`	Set level of pointer aliasing (for C and C++). Use only if you know the option to be safe for your program.
`[-xrestrict]`	Uses restricted pointers (for C). Use only if you know the option to be safe for your program.

Final Remarks

There are many other options that the compilers recognize. The ones presented here probably give the most noticeable performance gains for most programs and are relatively easy to use. When selecting the compiler options for your program:

- It is important to be aware of just what you are telling the compiler to do. A program may have unpredictable results if it does not conform to the requirements of the flags.
- When using optimization you will often be trading increased compile time for improved runtime performance.

This leads to the final suggestion that you should only use the flags which both give you a performance benefit and make acceptable assertions about the code.

For details on all these options, see the Sun Studio compiler user guides and man pages (`http://docs.sun.com/doc/820-3845`).

Further Reading

- *Solaris Application Programming* (http://www.sun.com/books/catalog/solaris_app_programming.xml) by Darryl Gove covers the use of the compiler as well as information about the use of many other tools provided as part of the Solaris OS.
- *Memory Hierarchy In Cache-Based Systems* (http://www.sun.com/blueprints/1102/817-0742.pdf) by Ruud van der Pas

  This Sun BluePrints online article helps the reader understand the architecture of modern microprocessors. The article introduces and explains the most common terminology and addresses some of the performance-related aspects. (PDF)
- *Application Performance Optimization* (http://www.sun.com/blueprints/0302/optimize.pdf) by Börje Lindh

  This Sun BluePrints online article provides a brief introduction to optimization on the Solaris operating environment. (PDF)
- C/C++/Fortran Compiler Documentation (http://docs.sun.com/doc/820-3845)

http://docs.sun.com/source/820-5242/index.html

The Much-Maligned - fast

Darryl Gove, March 20, 2008

The compiler flag -fast gets an unfair rap. Even the compiler reports:

```
cc: Warning: -xarch=native has been explicitly specified, or
implicitly specified by a macro option, -xarch=native on this
architecture implies -xarch=sparcvis2 which generates code that
does not run on pre UltraSPARC III processors
```

which is hardly fair given that the UltraSPARC III line came out about 8 years ago! So I want to quickly discuss what's good about the option, and what reasons there are to be cautious.

The first thing to talk about is the warning message. -xtarget=native is a good option to use when the target platform is also the deployment platform. For me, this is the common case, but for people producing applications that are more generally deployed, it's not the common case. The best thing to do to avoid the warning and produce binaries that work with the widest range of hardware is to add the flag -xtarget=generic *after* -fast. (Compiler flags are parsed from left to right, so the rightmost flag is the one that gets obeyed.) The generic target represents a mix of all the important processors, and the mix produces code that should work well on all of them.

The next option which is in -fast for C that might cause some concern is -xalias_level=basic. This tells the compiler to assume that pointers of different basic types (for example, integers, floats, etc.) don't alias. Most people code to this, and the C standard

actually has higher demands on the level of aliasing the compiler can assume. So code that conforms to the C standard will work correctly with this option. Of course, it's still worth being aware that the compiler is making the assumption.

The final area is floating point simplification. That's the flags `-fsimple=2` which allows the compiler to reorder floating point expressions; `-fns` which allows the processor to flush subnormal numbers to zero; and some other flags that use faster floating-point libraries or inline templates. I've previously written about my rather odd views on floating point math (`http://blogs.sun.com/d/entry/ieee-754_a_skewed_view`). Basically it comes down to: *If these options make a difference to the performance of your code, then you should investigate why they make a difference.*

Since `-fast` contains a number of flags which impact performance, it's probably a good plan to identify exactly those flags that do make a difference, and use only those. A tool like ATS (`http://cooltools.sunsource.net/ats/`) can really help here.

`http://blogs.sun.com/d/entry/the_much_maligned_fast`

Improving Code Layout Can Improve Application Performance

Darryl Gove, June 22, 2005

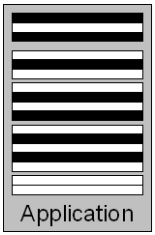
Introduction

Large applications have a particular problem: they have a lot of instructions, and the processor does not have the capacity to hold the entire application on-chip at any one time. As a consequence, larger applications spend some of their run time stalled, with the processor waiting to fetch new instructions from memory. This paper discusses several techniques that help the processor to hold more useful instructions on-chip, consequently reducing the time that is wasted fetching data from memory.

Not All Instructions Are Equal

An application will have many instructions, code has to be written to cover all eventualities—even those that rarely (and perhaps never) happen. A consequence of this is that most applications end up with a set of instructions that do the work, and a lot of other instructions which have to be there but are never used. The following figure shows a way of visualizing this. The grey rectangle represents the whole application. Within this application there are a number of routines. Within each routine there are instructions that are frequently executed, which are colored white, and instructions that are rarely executed, which are colored black.

FIGURE 2–1 Hot and Cold Regions of Code



The rarely executed instructions take up space in memory, and also in the caches, and often in the on-chip memory. For example, a single cacheline may contain a mix of hot and cold instructions. The cold instructions will just take up space, and consequently the application will have to use more cachelines to hold the code. It is also possible that due to the layout of the code in memory, some of the useful code may try to occupy the same place in the cache as some other useful code. This is known as "thrashing in the cache," which results in only a limited set of the critical instructions being available at any one time.

The symptoms of problems with code layout are that the application has a high number of Instruction Cache miss events, Instruction TLB miss events, or branch misprediction events. All these can be identified using the performance counters on the UltraSPARC-III derived processors. (See the article Using UltraSPARC-IIICu Performance Counters to Improve Application Performance (`http://developers.sun.com/solaris/articles/pcounters.html`).)

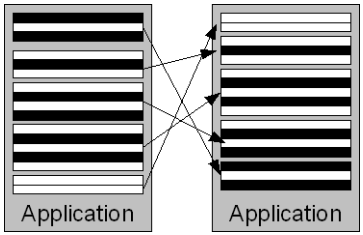
The next step is to look at some of the techniques for improving the layout of code in memory, but before doing that, it is important to realize that this doesn't just happen at the level of instructions. Whole routines are often either heavily used, or rarely used. Similarly, libraries might be full of frequently used routines, or might be required only because of a single library call which almost never happens.

Since the compiler has the ability to change the way the code is laid out in memory, it is possible for the compiler to use memory more efficiently, but it will need more information to do this. The remainder of this article covers three different approaches that can be taken to improve the layout of the application in memory.

Using Mapfiles to Reorder Routines

One approach to improve the situation is to use mapfiles. Mapfiles are a facility that tell the linker how to lay out routines in memory. To use these to improve the layout of the code, it is necessary to order the routines from the most frequently used to the least frequently used. The following figure shows our original program from Figure 2–1 laid out from hot routines to cold using a mapfile.

FIGURE 2–2 Routines Reordered Using Mapfiles



It is possible to manually generate mapfiles, but an easier approach is to use the Performance Analyzer:

1. Build the program using the flag `-xF`
2. Run the program with a representative workload under collect
3. Generate the mapfile using `er_print -mapfile <app> <mapfilename> <experiment>`
4. Rebuild the application with the flags `-xF -M <mapfile>`

Once a mapfile is generated for an application, the same mapfile can be used on subsequent compiles until either the profile of the application changes, routines are renamed, or additional routines are added.

EXAMPLE 2–1 Creating a Mapfile Using the Performance Analyzer Tools

```
$ cc -O -xF -o app *.c
$ collect app < test_data
   Creating experiment test.1.er ...
$ er_print -mapfile app app.map test.1.er
$ cc -O -xF -M app.map -o app *.c
```

Improving the Layout of Instructions By Using Profile Feedback

Mapfiles work very well at the routine level to separate frequently executed routines from infrequently executed routines. However, much of the time is spent at the instruction level, where the processor has to jump over blocks of unexecuted code. Profile feedback is a compiler technique for improving this situation.

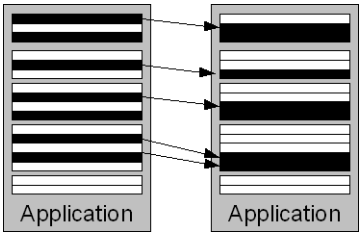
The idea with profile feedback is to give the compiler information about how the code is typically run. Based on this information it can do optimizations of the following types:

- Arrange code so that the frequently executed code in a routine is grouped together.
- Inline routines that are frequently called, to both remove the cost of calling the routine, and potentially to enable further optimization of the inlined code.

Profile feedback works best with crossfile optimization (controlled by the flag `-xipo`) since this allows the compiler to look at potentially optimizations between all source files.

The following figure shows how profile feedback can rearrange code within a routine to put the frequently executed code together.

FIGURE 2–3 Application Showing Rendering of Code Using Profile Feedback



Profile feedback is relatively straightforward to use:

1. Build the application with `-xprofile=collect -xipo`
2. Run the application with one or more representative workloads
3. Rebuild the application with `-xprofile=use -xipo`

Notice the inclusion of the `-xipo` flag to enable the compiler to do optimisations across the source files.

EXAMPLE 2–2 Using Profile Feedback to Optimize an Application

```
$ cc -O -xprofile=collect:app.profile -xipo -o app *.c
$ app < test_data
$ cc -O -xprofile=use:app.profile -xipo -o app *.c
```

Link-Time Optimization

Mapfiles work at the routine level, and profile feedback works within routines; it would seem to be a simple progression to do both optimizations at the same time. This is possible with link-time optimization (also called post-optimization).

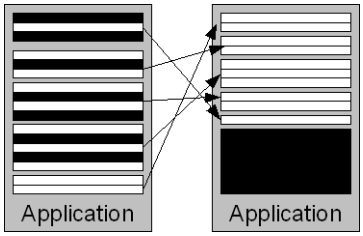
The principal of link-time optimization is that the compiler has done its work, the code exists, and all that is necessary is to lay it out appropriately. In laying the code out appropriately, the link-time optimizer will sort the routines so that hot routines are placed together (in a similar way to mapfiles), and also lay out the code within those routines so that hot instructions are placed together. However, it is possible at link-time to go beyond this:

- Since the hot code has been identified, it is possible to place all the hot code together, and then place all the cold code together. The idea being to remove all cold code from the hot region, placing code from different routines into the same region of memory.

- It is also possible to do further optimizations since the addresses of variables and routines can be calculated exactly. Hence the link-time optimizer can simplify expressions which calculate the address of variables or routines—this further reduces the instruction count.

The following figure shows what an application will look like after it has been link-time optimized. The hot code is grouped together in one part of the binary, and the cold code in a separate part.

FIGURE 2–4 Application Built Using Link-Time Optimization



The link-time optimization step requires profile feedback data to work, so the necessary steps are as follows:

1. Build the application with the flags `-xprofile=collect -xipo`
2. Run the application with one or more representative workloads
3. Rebuild the application with `-xprofile=use -xipo -xlinkopt`

EXAMPLE 2–3 Combining Link-Time Optimization With Profile Feedback

```
$ cc -O -xprofile=collect:app.profile -xipo -o app *.c
$ app < test_data
$ cc -O -xprofile=use:app.profile -xipo -o app *.c -xlinkopt
```

Concluding Remarks

Using these techniques on larger applications can yield significant performance gains. It should be noted that there is a cost in terms of increased build times, and increased build complexity; consequently, the techniques should be evaluated as to whether the gain is worth the additional effort in the build. It should also be observed that not all builds of the application need to go through the process of optimizing the code layout. Development builds can be performed without this process, and the process only applied to the final product build.

http://developers.sun.com/solaris/articles/codelayout.html

Using Profile Feedback to Improve Performance

Darryl Gove, Chris Aoki, September 2005

Summary

Profile feedback is a useful mechanism for providing the compiler with information about how code behaves at runtime. Having this information can lead to significant improvements in the performance of the application. As with all optimizations, it is only worth using profile feedback if it does produce a gain in performance.

Some degree of care is required in selecting representative workloads for providing training data to the compiler. The representativeness of the workloads can be examined by comparing the profiles gathered by tools such as `tcov` or the Performance Analyzer.

Introduction

When an application is compiled, the compiler will do its best to select the optimal instructions to use and the optimal layout for the code. It has to make decisions based on the source code, but the source code contains no information about the dynamic behavior of the code so the compiler has to use heuristics to provide a best guess.

The heuristics are used to determine how to structure the code, which routines should be inlined, which bits of code are executed frequently, and many other details.

An example of one of the problems that the compiler faces is the code shown in the following example:

```
void calculate(...)
{
  if (...some condition...)
  {
     // do calculation        CODE REGION A
     ...
  }
  else
  {
     // do calculation        CODE REGION B
     ...
  }
  // do more work
  ...
}
```

In the code shown above, the compiler has an interesting decision to make because there are different ways of structuring the code. Should the compiler make A or B the default (and therefore make that path faster), or should it structure the code so that both branches of the code have equal performance?

The question of how best to arrange IF statements is one of the many decisions affecting code layout that the compiler has to make. Examples of other decisions are:

- Is a routine executed sufficiently often that inlining it will improve performance?
- Can the code be laid out in memory so it uses the instruction cache more effectively?
- Are there loops which are iterated over sufficient numbers of times that it is worth unrolling them?

Most of these decisions can only be governed by heuristics since the compiler has no information about what happens to the program at runtime. However, there is a mechanism, called profile feedback, which enables the compiler to gather information about what happens to the program at runtime using data from a run of a representative training dataset.

Using profile feedback on the benchmark suite SPEC CPU2000 leads to an average of about 7% performance gain for the floating point suite and a 16% performance gain for the integer suite. For individual codes within the suite, the performance gains vary from no gain for some codes, to significant gains for others.

Building With Profile Feedback

The idea with profile feedback is to run the program for a short time and gather data about what happens to the program during that run. The compiler then uses that data to refine its optimization decisions.

The process of using profile feedback is:

1. The binary is built with the flag `-xprofile=collect`. This flag produces a special version of the application (called an instrumented binary), which when run will gather data about the run.
2. The application is then run with a "training" workload. A training workload is a workload that is representative of the real work that the application will do, but does not need to last as long as a real workload.
3. The binary is rebuilt using the flag `-xprofile=use`. This flag uses the previously collected data to optimize the binary.

This means that, using profile feedback, the build process will take about twice as long compared to building without it. This is because the build involves two passes through the compiler, plus a short run of the application. It is therefore important that the gains in performance seen at runtime are worth the extra build complexity.

Selecting a Representative Workload

Building with profile feedback requires that a representative training workload is used to inform the compiler about the runtime behavior of the application. The key points about this workload are:

- It should take little time to run. Running for a long time does not necessarily improve the data being fed to the compiler.
- The workload should exercise all the critical parts of the application. Tools such as the Sun Studio Performance Analyzer (`http://developers.sun.com/solaris/articles/analyzer_qs.html`) or `tcov`(1) can be used to assess whether the training workload covers the critical sections of code, and whether the profile of the training workload is similar to that of the real workload.
- Several training workloads can be used if this improves the coverage of the code.

A concern that is sometimes raised is whether using the wrong training workload might lead to worse performance for some cases. This is possible, but it typically comes about for one of two reasons:

- The training workload did not cover the entire application. The problem code happens to use part of the application for which the compiler had inadequate information.
- The behavior of the problem workload is significantly different from the training workload.

In both cases it may be that adding another training workload will improve the performance for the problem workload. It is also worth looking at the code coverage or time spent in the various routines so that the reason for the difference in performance can be identified. It is rare that training for one workload will force another workload to run slower. It is more likely that the training data has indicated to the compiler that a particular optimization is unnecessary, and using additional training data which provides evidence that the optimization is necessary will improve performance for the problem workload whilst not impacting performance for other workloads.

The Benefits of Profile Feedback

The more information that the compiler has, the better job it can do at optimizing the application. As with all optimizations, some code will greatly benefit, while other code will see no gains. It is strongly dependent on the type of code.

The type of code that is likely to benefit from profile feedback is code which has a large number of conditional statements (IF statements). The largest benefit will be code which has very predictable behavior, but the behavior is not obvious to the compiler.

A simple example of this kind of code is where there are checks for correct values. The compiler cannot easily determine whether the programmer expects the checks to pass or fail, so it will

typically make the null assumption that passing and failing are equally likely. However, if the test is for valid data and most of the time the values in the code are valid, then profile feedback will enable the compiler to identify this, and optimize the code appropriately.

Another situation where profile feedback can lead to performance gains is when the profile can be used to select the best set of routines for the compiler to inline. There are two benefits from inlining. The first is to eliminate the cost of the call to the routine. The second is to expose further opportunities for optimization. The downside of inlining is that it can lead to an increase in code size. If the inlined code turns out not to be useful, then this increase in code size may actually reduce performance. Profile feedback enables the compiler to correctly select the routines which are frequently called and are therefore candidates for inlining, whilst rejecting routines which are rarely called.

Profile Feedback Compiler Flags

The flag that tells the compiler to either build the application and collect a profile, or build the application and use an existing profile is `-xprofile`. The use of the flag has some subtleties which require a bit more explanation.

- `-xprofile=collect` can take an optional parameter which tells the compiler where to place the profile information. For example:

  `-xprofile=collect:myapp` will place the profile data in a directory called `myapp.profile` in the current directory at the time that the program is executed. Similarly, `-xprofile=collect:/tmp/myapp` will place the profile data in the directory `/tmp/myapp.profile`. In the event that the location is not specified, then the profile is placed in the directory *prog*`.profile` where *prog* is the name of the executable at the time the executable is run.

- `-xprofile=use` can also take an optional parameter telling the compiler where the profile data is located.

  `-xprofile=use:/tmp/myapp` will use the profile data located in `/tmp/myapp.profile`. If no location is specified, then the compiler will look for data in `a.out.profile` in the current directory. Notice that this is different behavior from the `-xprofile=collect` phase. The reason for the difference in behavior is that the profile collector can determine the name of the executable when collecting the data, but when the compiler is building the new application using profile data, it does not know the name of the application that was used to generate the profile.

  Note: It is a good practice to always specify the full location of the profile data when building the executable.

Specifying Other Compiler Flags With Profile Feedback

When the application is compiled with `-xprofile=collect` to collect profile information, the binary is produced with a lower level of optimization than would otherwise occur. This is so that the data gathered is more detailed than the data that would be gathered using an optimized binary. The instrumented binary produced will have a particular layout of the code depending on both the source code and the flags used to build it. If the flags are changed, the layout of the code may change.

Note: Apart from the arguments to `-xprofile`, it is best to specify the same flags for both the collection and use phases.

Running the Executable to Collect Profile Information

When the executable is run, the profile data is written into the file system. The write takes place at the end of the run, so if the application fails to run to completion, then there may well be no profile data written. If the application is run multiple times, then the profile data accumulates the results from all the runs.

If the source code is modified, it is not a good idea to reuse old profile data. Although the compiler might not complain or report an error, it is unlikely that the compiler is taking the optimal decisions.

Note: It is a good practice to remove the old profile data whenever a new `-xprofile=collect` binary is built, and for new profile data to be collected every time the source is changed.

Compiler Options That Use Data Collected by Profile Feedback

There are several compiler options which use profile feedback information:

- At optimization level `-x05`, profile feedback enables the compiler to generate speculative instructions in some frequently generated regions of code. In the absence of profile feedback, speculative instructions are still generated at `-x05` but much more sparingly.
- The compiler flags `-xipo` and `-xcrossfile` perform crossfile optimization, meaning optimizations that are across multiple source files. One example of this kind of optimization is inlining a routine from one source file into code from another source file. In the presence of profile feedback, the compiler has a much better model of the set of routines that are worth inlining

- The compiler flag `-xlinkopt` causes the compiler to perform link time optimization. This final phase of compilation uses all the knowledge of the generated code in order to do some final tweaking of the code layout. This is useful for large blocks of code where performance can be gained by laying out the code to keep all the frequently executed code together.

Example Code Using Profile Feedback

The code shown in the following example has opportunities for improvement to code layout from profile feedback. From inspection of the code it is obvious that the time is spent calling function f. This function sums up the six values passed into it but before performing the sum, it checks that each of the pointers to the values is valid. In the example, all the values are valid, and for most checks of this kind found in programs, it is usual for the data to be valid. However, the compiler cannot identify that the tests will usually be valid, so has to make the assumption that both of the two conditions in the IF statement are equally likely.

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

static unsigned f( unsigned *a0, unsigned *a1, unsigned *a2,
                   unsigned *a3, unsigned *a4, unsigned *a5)
{
 unsigned result = 0;
 if (a0 == NULL) { printf("a0 == NULL"); } else { result += (*a0); }
 if (a1 == NULL) { printf("a1 == NULL"); } else { result += (*a1); }
 if (a2 == NULL) { printf("a2 == NULL"); } else { result += (*a2); }
 if (a3 == NULL) { printf("a3 == NULL"); } else { result += (*a3); }
 if (a4 == NULL) { printf("a4 == NULL"); } else { result += (*a4); }
 if (a5 == NULL) { printf("a5 == NULL"); } else { result += (*a5); }
 return result;
}

void main(int argc,const char *argv[])
{
  int i, j, niters = 1, n=6;
  unsigned sum, answer = 0, a[6];

  niters = 1000000000;
  if (argc == 2) { niters = atoi(argv[1]); }

  for(j=0; j<n; j++)
  {
   a[j] = rand();
   answer += a[j];
  }

  for(i=0; i<niters; i++) { sum=f(a+0, a+1, a+2, a+3, a+4, a+5); }

  if (sum == answer) { printf("answer = %u\n", answer); }
  else { printf("error sum=%u, answer=%u", sum, answer); }
}
```

The output below shows the results of compiling and running this program without profile feedback.

```
$ cc -O -o example example.c
$ timex example 1000000000
answer = 86902

real 43.87
user 43.28
sys 0.00
```

The next output shows the process of compiling this code with profile feedback. Notice that there is a training run of the program using far fewer iterations of the main loop.

```
$ cc -O -xprofile=collect:./example -o example example.c
$ example 100
answer = 86902

$ cc -O -xprofile=use:./example -o example example.c
$ timex example 1000000000
answer = 86902

real 34.52
user 33.93
sys 0.01
```

The 10–second difference in runtime between the two codes represents about a 25% improvement. Obviously this particular example has been put together to demonstrate profile feedback optimizations, but the principles that it shows appear in most codes.

http://developers.sun.com/solaris/articles/profeedback.html

Selecting Representative Training Workloads for Profile Feedback Through Coverage and Branch Analysis

Darryl Gove, September 29, 2006

Introduction

Profile feedback (http://docs.sun.com/source/819-3688/cc_ops.app.html#38456) is an optimization technique that uses a short training run of the application to provide the compiler with more detailed information about the runtime behavior of the program. This information enables the compiler to make better optimization decisions (as described in "Using Profile Feedback to Improve Performance" on page 58), for example, which routines are appropriate to inline, or which branches are the frequently taken path.

However, there are two reasons why profile feedback is not more widely adopted. The first is that compiling for profile feedback does increase the complexity and time for the build process. Typically doing two passes through the compiler will, unavoidably, take about twice as long as doing a single pass.

The second reason why profile feedback is not more widely used is that there is a concern that a performance gain for one workload may be at the expense of the performance of another workload. In fact this appears not to be true. The consensus appears to be that the behavior of branches is generally invariant over different workloads. (A more detailed literature survey is included in the paper (PDF) (http://www.spec.org/workshops/2006/papers/10_Darryl_Gove.pdf) presented at the SPEC Workshop (http://www.spec.org/workshops/2006/) in Austin during January 2006.) The problem is that whilst it is generally true that branches behave in the same way, it does not mean that this true for a specific application.

This paper presents two ways of viewing the correspondence between the behavior of the training and reference workloads. The methods presented here are necessary conditions for the training workload to be representative of the reference workload.

Using the Tools

The Binary Instrumentation Tool (http://cooltools.sunsource.net/bit/) (BIT) has been released as part of the Cool Tools (http://cooltools.sunsource.net/) effort on SPARC systems. These tools are add-ons to Sun Studio (http://developers.sun.com/sunstudio). BIT can gather data on the number of times basic blocks are executed, or the probability that a given branch is taken.

In order to gather data using BIT, the binary needs to be built with an optimization level of at least `-xO1`, and the flag `-xbinopt=prepare`. The `-xbinopt` flag tells the compiler to produce an annotated application, suitable for further analysis at a later time. More details on binary optimization with `binopt` can be found in a recent article (http://developers.sun.com/prodtech/cc/articles/binopt.html).

A good example of what can be achieved is to run the BIT tool on the test program shown below.

```
$ more test.c

void main ()
{
  int i;
  int j;
  j=0;
  for (i=0; i<1000; i++)
  {
    if (i==j) {j--;}
  }
}
```

Here there is an inner loop that will get executed 1000 times. On the first iteration of the loop the variable `j` will be decremented.

The application is then built as described above. In this case low optimization is used because at higher levels of optimization the entire program will be eliminated. The command to build an executable for binary optimization is:

```
$ /opt/SUNWSpro/bin/cc -O -xbinopt=prepare -o test test.c
```

The man page (`http://cooltools.sunsource.net/bit/Docs/bit.1.html`) for BIT describes the options that are available for the tool. The information that is most useful for determining whether the training workload is appropriate are the basic block counts, and the branch taken probabilities.

The following command will run an instrumented application and output basic block counts and branch probabilities.

```
$ /opt/SUNWspro/extra/bin/bit collect -R -o bbc.txt -a bbc -o branch.txt -a branch a.out
```

The basic block count output is as follows:

```
$ more bbc.txt

Basic Block Counts for whole program:
Count PC      #Instrs Function name
1     0x11794 3       main
1000  0x117a0 2
1     0x117a8 1
1000  0x117ac 3
1     0x117b8 2
```

It is also possible to get disassembly from BIT which includes information about the execution counts for the assembly language instructions.

```
$ /opt/SUNWspro/extra/bin/bit analyze -a dis test

Disassembly for routine main
 ROUTINE: main FREQUENCY: 1.0

BLOCK: main: FREQUENCY: 1.0 PC: 0x11794
 [ 1.0]     0x11794: or %g0, #sint=0, %o5
 [ 1.0]     0x11798: or %g0, #sint=0, %o4
 [ 1.0]     0x1179c: subcc %o4, %o5, %g0

BLOCK: $LABEL_main_3_3_117a0: FREQUENCY: 1000.0 PC: 0x117a0
 [ 1000.0]  0x117a0: br,pn@(ne),%icc $LABEL_main_1_1_117ac
 [ 1000.0]  0x117a4: add %o4, #sint=1, %o4

BLOCK: $LABEL_main_2_2_117a8: FREQUENCY: 1.0 PC: 0x117a8
 [ 1.0]     0x117a8: add %o5, #sint=-1, %o5

BLOCK: $LABEL_main_1_1_117ac: FREQUENCY: 1000.0 PC: 0x117ac
 [ 1000.0]  0x117ac: subcc %o4, #sint=999, %g0
 [ 1000.0]  0x117b0: br,pt@(le),%icc $LABEL_main_3_3_117a0
 [ 1000.0]  0x117b4: subcc %o4, %o5, %g0

BLOCK: $LABEL_main_4_4_117b8: FREQUENCY: 1.0 PC: 0x117b8
 [ 1.0]     0x117b8: jmpl [%o7, #sint=8], %g0
 [ 1.0]     0x117bc: nop
```

It is possible to see from the disassembly how the routine is structured, with three basic blocks in the loop, and one of these only gets executed when the variables `i` and `j` have the same value.

There are two branches in the code, the first (at `0x117a0`) to branch around the `j--` statement when `i` and `j` are not equal (this branch will be taken 999 times out of 1000). The second branch (at `0x117b0`) is the branch back to the top of the loop, which will also get taken 999 times out of 1000.

The report on branch probabilities confirms this. The report shown has been trimmed to reduce the number of columns shown.

```
$ more branch.txt

Branch taken/not taken report for whole program:
PC     Trip Cnt Taken Not Taken Instruction
117a0  1000     999   1         br,pn@(ne),%icc $LABEL_main_1_1_117ac
117b0  1000     999   1         br,pt@(le),%icc $LABEL_main_3_3_117a0
...
```

Methodology

The methodology proposed here is to gather both basic block count and branch probability data for the training and reference workload. If there are multiple training workloads, the aggregate data for all of them should be gathered. The following is a step-by-step walkthrough of the process of gathering the data for either the training or reference workload. The middle step where the application is run can be repeated multiple times if there are multiple training (or reference) workloads.

```
$ /opt/SUNWspro/extra/bin/bit instrument -R test
$ test
$ /opt/SUNWspro/extra/bin/bit analyze -R -a dis test

Disassembly for routine main
ROUTINE: main FREQUENCY: 1.0
BLOCK: main: FREQUENCY: 1.0 PC: 0x11794
.....
```

Coverage as a Measure of Training Workload Quality

For a workload to adequately train an application for a reference workload it should *at least* execute the same parts of the code that the reference workload does. If the training workload does not execute the same code as the reference workload, then at best it is leaving some opportunities for performance gain behind; at worst it is misleading the compiler by indicating that an important part of the code for the reference workload is not important.

Coverage can be estimated from basic code block counts. If a basic block is executed more than once, it has been covered. It is not possible to draw strong conclusions from the magnitude of

the count attributed to a basic block—the magnitude can be a function of the duration of the run, or even a fixed count which depends on the size of a data structure.

It is possible to calculate coverage as a single value. Assume that T_i is the number of times that basic block i is executed during the run with the training workload. Similarly assume that R_i is the number of times that basic block i is executed during the run with the reference workload. The coverage can be defined as follows:

$$coverage = \frac{\sum_i \begin{bmatrix} R_i & T_i > 0 \\ 0 & otherwise \end{bmatrix}}{\sum_i R_i}$$

The coverage ranges from 0 to 100% (when all of the basic blocks that are critical to the reference workload are covered by the training workload).

Whilst coverage is a minimum criteria for acceptance of a training workload, having good coverage does not necessarily mean that the training workload is ideal. But we have a visual way of examining the coverage information for a particular pair of training and reference workloads.

Visually Comparing Reference and Training Coverage

One of the problems with basic block count data is that the counts depend on the runtime of the application. So a longer running workload will have higher counts. Therefore it does not make sense to try and compare the absolute count numbers. To present the data visually, the basic blocks are ordered according to their counts, basic blocks with higher counts are plotted further from the origin. For each basic block the ordering from the reference workload is used to determine position on the x-axis, and the ordering from the training workload is used to determine position on the y-axis. As a further refinement, the size of the marker used to plot each point depends on the frequency of execution (relative to the most frequently executed basic block) of the reference workload.

In an ideal situation, the most frequently executed basic blocks in the training workload will also be the most frequently executed basic blocks in the reference workload. As a graph, this should look something like a line of points going up at 45° with small markers near the origin, and large markers at the top right. This shape is probably best described as a "lollipop" shape.

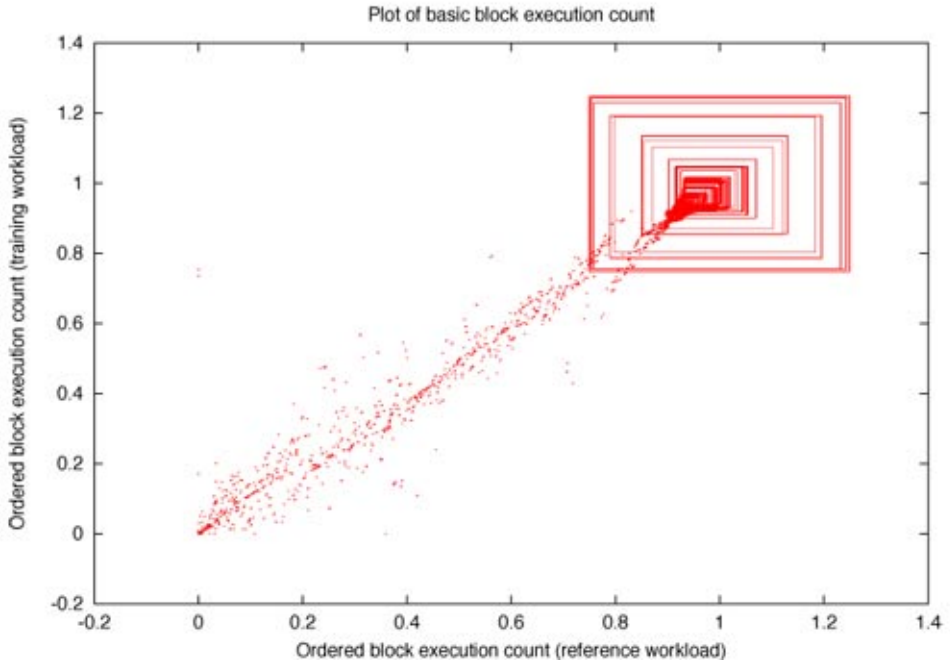
Basic blocks that are frequently executed by the reference workload but not adequately covered by the training workload will appear as large markers below the diagonal line.

Coverage Results from SPEC CPU2000

The paper presented at the SPEC workshop had results for all of the CPU2000 suite. In this paper, we'll just look at an example of good behavior, and an example of bad behavior.

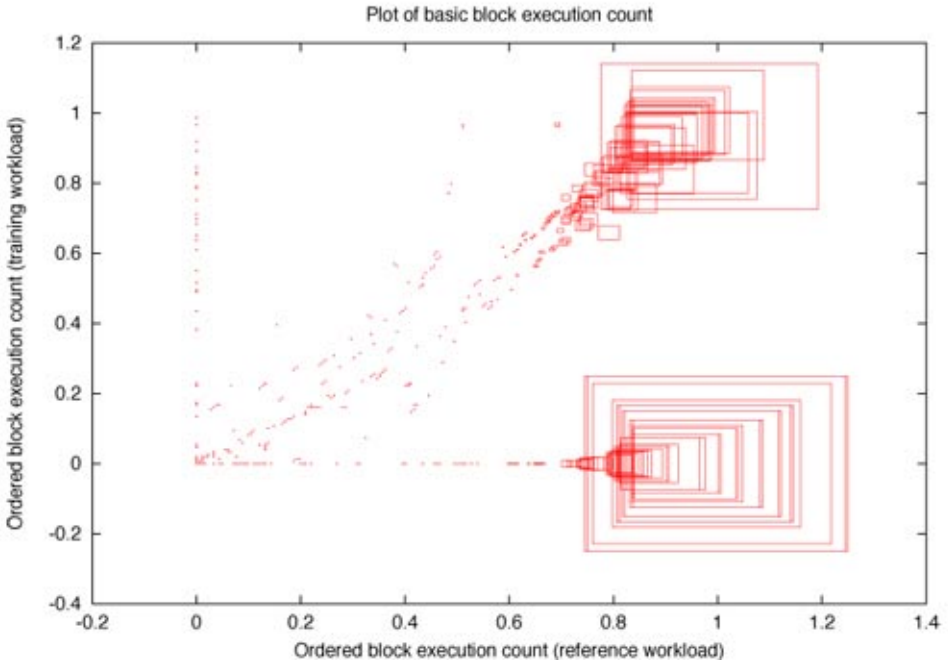
The benchmark `300.twolf` got a coverage of 100% using this methodology. The coverage graph for this benchmark is shown in the following figure.

FIGURE 2–5 Basic Block Coverage for Benchmark `300.twolf`



However, the benchmark `301.apsi` got a coverage of only 37%. The coverage plot for `301.apsi` is shown in the following figure. The coverage plot readily identifies the reason for the very poor coverage results: the training workload does not exercise a large part of the code that is frequently executed by the reference workload.

FIGURE 2–6 Basic Block Coverage for Benchmark `301.apsi`



Branch Probabilities

The next step beyond coverage of the frequently executed basic blocks is to determine whether the branches in the code behave in the same way. At the simplest level, a branch can be usually taken or usually not taken. If a given branch is usually taken (or usually not taken) in both the reference and training workloads, the training workload is appropriate.

A branch is declared to be usually taken if its taken more than half of the number of times that it is encountered in the instruction stream; otherwise it is declared to be usually not taken. It is possible to calculate a single value, which we will call a Correspondence Value, which denotes how well a training workload represents the branch behavior of the reference workload. The formulation of the Correspondence Value is very similar to that of the Coverage calculated above. Assume that the execution frequency for a branch instruction, i, in the reference workload is denoted F_i. Assume that R_i is given the value 1 if branch instruction i is usually taken in the reference workload. Similarly, assume that T_i is given the value 1 if branch instruction i is usually taken in the training workload. Then the Correspondence Value can be calculated as follows:

$$Correspondence\ Value = \frac{\sum_i \begin{bmatrix} F_i & R_i = T_i \\ 0 & R_i \neq T_i \end{bmatrix}}{\sum_i F_i}$$

The Correspondence Value ranges from zero to 100%, meaning that all branches behave the same way in both training and reference workloads.

A high Correspondence Value indicates a strong agreement between the behavior of the branches in the training and reference workloads. However, a low agreement does not necessarily mean that the training workload is inappropriate for the reference workload. There are two situations which may lead to a lower than expected Correspondence Value. The first situation is where the branches are taken about half the time, and the branches in the reference and training workloads happen to fall on different sides of the halfway mark. The second situation is where the branch behavior of the application is unpredictable. It is possible to better distinguish the branch behaviors by plotting the branches graphically.

Visually Comparing Branch Probabilities for the Training and Reference Workloads

Branch probability data is always bound between zero (never taken) and 1 (always taken). This makes it simple to plot each branch instruction on a chart, where the location on the x-axis is determined by the probability of the branch being taken in the reference workload, and the location on the y-axis being determined by the probability of the branch being taken in the training workload. To convey the importance of the various branch instructions, the size of the marker depends on the frequency of execution of the branch instruction in the reference code (as a proportion of the execution frequency of the most frequently executed branch).

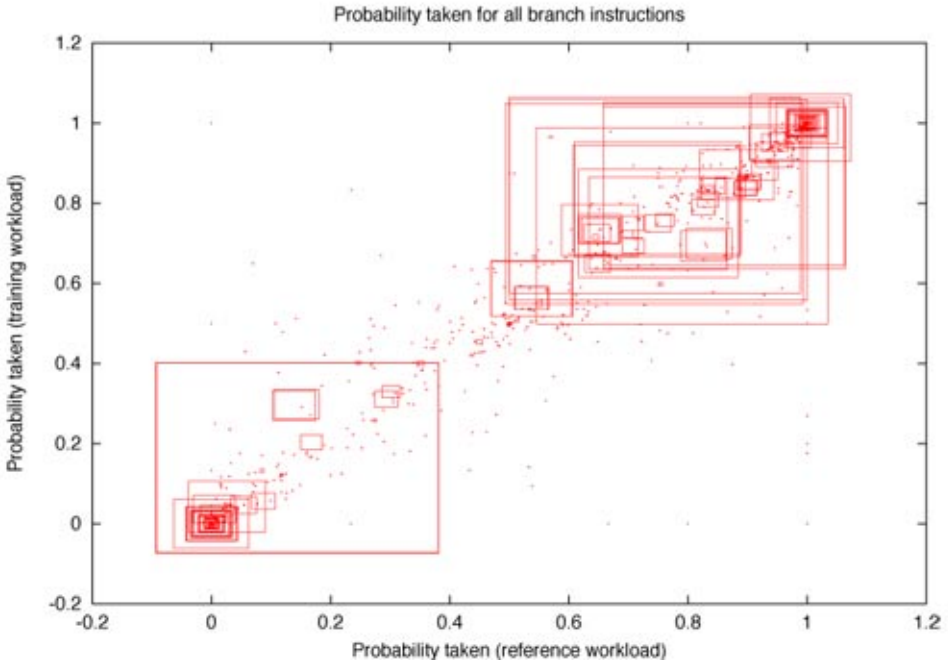
In an ideal version of this graph, the branches would all lie on the diagonal line running from the origin, at (0,0), to the top right hand corner at (1,1). Most workloads will not conform to this ideal. A more general template is to imagine the graph split into the four quadrants. The upper-right and lower-left quadrants indicate that the branch behavior is similar in the reference and training workloads. Branches that appear in the upper-left or lower-right quadrants are being mistrained.

Workloads which have a level of uncertainty (or randomness) about branch behavior will typically be indicated by a smear of branches around the diagonal.

Branch Probability Results From SPEC CPU2000

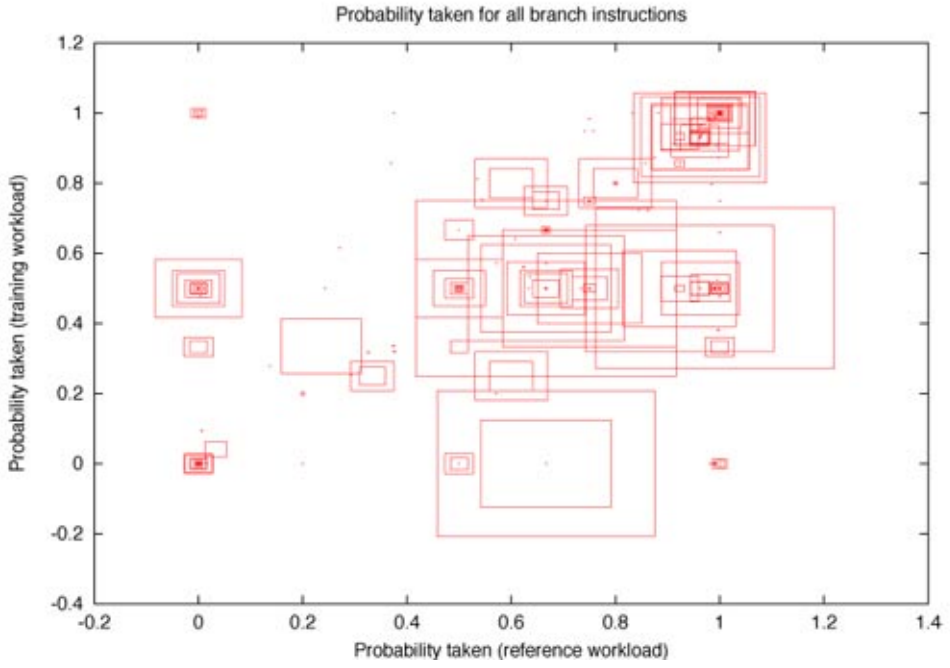
The first benchmark to examine is `300.twolf`, which has a Correspondence Value of 100% and shows good correspondence between the branch behavior in the training and reference workloads. In the following figure, it is apparent that all the branches that are frequently executed by the reference workload appear in either the upper-right or lower-left quadrant, indicating good agreement between the training and reference workloads.

FIGURE 2–7 Branch Probabilities for `300.twolf`



Once again, the benchmark `301.apsi` scores a low Correspondence Value of 72%. The following figure shows this in a graphical format. It is apparent that one reason for the low score is that there are a number of branches which have a probability of being taken in the training workload of about 50%, but are more strongly taken in the reference workload. Another reason for the low score is that there are several branches which are usually taken in the reference workload, but usually not taken in the training workload.

FIGURE 2–8 Branch Probabilities for `301.apsi`



Concluding Remarks

This paper has presented several methodologies for determining if a particular training workload is appropriate for a particular reference workload. The same approach can be extended to determine if the behavior of an application is similar over a range of reference workloads.

The cumulative effect is that given a situation where building an application with profile feedback leads to a faster runtime for a given reference workload, it is possible to look at the behavior of the branches and basic blocks in an application and determine whether the training workload is appropriate for the range of reference workloads that the application actually has to deal with. If, as a result of this analysis, it is determined that the training workloads do not adequately train for a particular reference workload, then it is possible to identify additional training workloads to be used.

Supporting Scripts

Two Perl scripts were written to support this analysis:

- blcompare.pl (`http://developers.sun.com/solaris/articles/src/coverage_blcompare`), which compares two basic block data files and determines the Coverage of the reference workload by the training workload.
- brcompare.pl (`http://developers.sun.com/solaris/articles/src/coverage_brcompare`), which compares two branch probability files and calculates the Correspondence Value.

Both scripts will generate the appropriate graphs if they are able to locate `gnuplot` on the path.

`http://developers.sun.com/sunstudio/articles/coverage.html`

Using Inline Templates to Improve Application Performance

Darryl Gove, August 2003

Summary

Inline templates are a mechanism for directly inserting assembly code into an executable. Typically, this approach is used to obtain the best performance for a given function, or to implement an algorithm in a specific way.

Introduction

In general, you should never need to use inline templates. It is normally possible to do all the coding in a high-level language, and the compiler is able to do an excellent job of optimizing this. However, in some cases you may either know more about the target hardware, more about the behavior of the code, or perhaps want to do something that the compiler doesn't readily support. In these rare situations you will find inline templates to be helpful.

The following are examples where inline templates are particularly useful:

- **User-coded mutex locks.** If you want to code a mutex lock, then you will probably want to use the atomic instructions.
- **Hardware-level access.** If you are coding for a hardware device, or perhaps just accessing the registers already present on the system, then you may end up wanting to use inline templates.
- **Precise implementation of algorithms.** If you have a short algorithm which can be implemented optimally using hand-coding tricks that the compiler is unable to replicate, then you may wish to use inline templates.

To use inline templates, a regular function call is placed in the source code, then an inline template is written with the appropriate name, and at compile time both the source file and the file containing the inline template are compiled together. The compiler will then insert the code from the inline template into the code generated from the source code.

The documentation for inlining using `.il` files can be found under `man inline`(1). This paper is based on that data.

The `inline` man page is located at

```
man -M /opt/SUNWspro/man inline
```

The `inline` man page is also available in HTML (`http://docs.sun.com/source/820-4180/man1/inline.1.html`).

Compiling With Inline Templates

You compile inline templates by placing them on the same compile line as the file which uses them. The code is inlined by the code-generator stage of compilation.

The following example shows compiling with an inline template file.

```
cc -O prog.c code.il
```

This example will compile `prog.c` and inline the code from `code.il` into the appropriate points.

Layout of Code in Inline Templates

The inline template file can contain a number of inline templates. Each template starts with a declaration, and ends with an end statement, as shown in the following example of the layout of an inline template.

```
.inline identifier,argument_size
  ...instructions...
.end
```

The identifier is the name of the template, and `argument_size` is the size of the arguments in bytes (this is not required for the latest compiler versions). Multiple templates of the same name can be placed in the file, but the compiler will pick the first one.

There is no need for a return instruction since your template will be inlined directly into your code without a call.

Note that you must include a prototype for the template in your high-level source code to ensure that the compiler assigns correct types for all the parameters.

The following example shows a prototype for an inline template

```
void do_nothing();
```

The following example shows a template

```
/* The following template does nothing*/
.inline do_nothing,0
  nop
.end
```

The first line shows the prototype as it might end up in `code.h`. The next example shows the inline template code as it might end up in a separate `code.il` file. Inline templates are always in files with the suffix `.il`. In order to increase readability, the examples in this article show the prototype included with the inline template code, but in reality they must go into different files.

Guidelines for Coding Inline Templates

The inline code can only use integer registers `%o0` to `%o5` and floating-point registers `%f0` to `%f31` for temporary values. Other registers should not be used. These registers are referred to as the "caller-saved" registers. Calls can be made to other routines from the inline template, but these calls are subject to the same constraint.

The compiler will handle most of the SPARC instruction set. If the template contains only instructions which the compiler normally generates, then it will be early inlined (see below), and the code will be scheduled optimally. If the template contains instructions that the compiler understands but does not typically generate (such as VIS instructions or atomics), then the code may be late inlined, and consequently the code may not be optimally scheduled, resulting in a slight loss of performance.

Parameter Passing

Parameter passing obeys the parameter passing defined in the target architecture, so it is different for 32-bit and 64-bit codes. It is described by the SPARC ABI, which can be downloaded at `http://www.sparc.org/standards/SCD.2.4.ps.Z`. SCD 2.3 describes v8 (32-bit code) and SCD 2.4.1 describes v9 (64-bit code).

On entering the template, arguments will be passed in `%o0`-`%o5`, and will continue on the stack. For 32-bit code, the offset is `[%sp+0x5c]` and `%sp` is guaranteed to be 64-byte aligned. For 64-bit code, the offset is `[%sp+0x8af]`. (Note that `%sp+2037` is aligned to a 16-byte boundary.)

The following example shows 32-bit parameter passing using the stack

EXAMPLE 2–4 32–bit Parameter Passing Using the Stack

```
int add_up(int v1,int v2, int v3, int v4, int v5, int v6, int v7);

/*Add up 7 integer parameters - last one will be passed on stack*/
.inline add_up,28
  add %o0, %o1, %o0
  ld [%sp+0x5c],%o1
  add %o2, %o3, %o2
  add %o4, %o5, %o4
  add %o0, %o1, %o0
  add %o2, %o4, %o2
  add %o0, %o2, %o0
.end
```

In the following example for 64-bit code, note that when a 32-bit int register is passed on the stack, the full 64-bits of the register are saved:

EXAMPLE 2–5 64–bit Parameter Passing Using the Stack

```
int add_up(int v1,int v2, int v3, int v4, int v5, int v6, int v7);

/*Add up 7 integer parameters - last one will be passed on stack*/
.inline add_up,28
  add %o0, %o1, %o0
  ldx [%sp+0x8af],%o1
  add %o2, %o3, %o2
  add %o4, %o5, %o4
  add %o0, %o1, %o0
  add %o2, %o4, %o2
  add %o0, %o2, %o0
.end
```

For 32-bit code, floating-point values will be passed in the integer registers; for 64-bit code, they will be passed in the floating point registers.

EXAMPLE 2–6 32–bit Parameter Passing by Value

```
double sum_val(double a, double b);

/*sum of two doubles by value*/
.inline sum_val,16
  st   %o0, [%sp+0x48]
  st   %o1, [%sp+0x4c]
  ldd  [%sp+0x48], %f0
  st   %o2, [%sp+0x48]
  st   %o3, [%sp+0x4c]
  ldd  [%sp+0x48], %f2
  faddd %f0, %f2, %f0
.end
```

EXAMPLE 2–7 64–bit Floating-Point Parameter Passing

```
double sum(double a, double b);

/*sum of two doubles 64-bit calling convention*/
```

EXAMPLE 2–7 64–bit Floating-Point Parameter Passing *(Continued)*

```
.inline sum,16
  faddd %f0,%f2,%f0
.end
```

For values passed in memory, single-precision floating-point values and integers, are guaranteed to be 4-byte aligned. Double-precision floating-point values will be 8-byte aligned if their offset in the parameters is a multiple of 8-bytes.

Integer return values are passed in `%o0`. Floating-point return values are passed in `%f0`/`%f1` (single-precision values in `%f0`, double-precision values in the register pair `%f0`,`%f1`).

For 32-bit code there are two ways of passing the floating point registers. The first way is to pass them by value, and the second is to pass them by reference. Either way, the compiler will do its best to optimize out the load and store instructions. It is often more successful at doing this if the floating-point parameters are passed by reference.

EXAMPLE 2–8 32–bit Parameter Passing by Value

```
double sum_ref(double *a, double *b);

/*sum of two doubles by reference*/
.inline sum_ref,16
  ldd [%o0], %f0
  ldd [%o1], %f2
  faddd %f0, %f2, %f0
.end
```

Stack Space

Sometimes it is necessary to store variables to the stack in order to load them back later, for example, for moving between the int and fp registers. The best way of doing this is to use the space which is already set aside for the parameters which are passed into the function.

For example in the v8 code shown in Example 2–6, the location `%sp+0x48` is 8-byte aligned (`%sp` is 8-byte aligned), and it corresponds to the place where the 2nd and 3rd 4-byte integer parameters would be stored if they were passed on the stack. (Note that the first parameter would be stored at a non-8-byte boundary.)

Branches and Calls

There is support for branching and calls available. Every branch or call must be followed by a `nop` instruction to fill the branch delay slot. It is possible to put instructions in the delay slot of branches. This can be useful if you wish to use the processor support for annulled instructions, but doing so will cause the code to be late-inlined (described below), and may result in sub-optimal performance.

Call instructions must have an extra last argument which indicates the number of registers used to pass arguments in the call parameters. In general you should avoid inlining call instructions.

The destinations of branches must be indicated with a number, and the branch instructions should use this number to indicate the appropriate destination together with an `f` for a forward branch or a `b` for a backward branch.

EXAMPLE 2–9 Using Branches in an Inline Template

```
int is_true(int i);

/*return whether true*/
.inline is_true,4
   cmp  %o0, %g0
   bne  1f
   nop
   mov  1, %o0
   ba   2f
   nop
1:
   mov  0, %o0
2:
.end
```

Late and Early Inlining

Inlining of templates is done by the code generator part of the compiler. There are two opportunities for inlining: before and after optimisation. If the inline template is complex then it will end up being inlined after optimization (“late inlined”), which means that the code will more or less appear exactly as it appears in the template. If the code is inlined before optimisation (“early inlining”), then it will be merged with the other code around the call site.

Early inlining will lead to better performance.

Things that will cause late inlining are:

- Use of instructions that the compiler cannot generate
- Instructions in the delay slots of branches
- Call instructions

You will get information in the compiler commentary on inlining when the code is compiled with `-g`. This information will tell you if a routine is late inlined. If there is no comment, then the routine will have been early inlined. An example of this is attempting to inline the following (incorrect) template:

```
.inline sum_val,16
  st   %o0, [%fp+0x48]
  st   %o1, [%fp+0x4c]
  ldd  [%fp+0x48], %f0
  st   %o2, [%fp+0x48]
```

```
  st   %o3, [%fp+0x4c]
  ldd  [%fp+0x48], %f2
  faddd %f0, %f2, %f0
.end
```

The template is incorrect because the code uses the frame pointer (`%fp`) rather than the stack pointer (`%sp`). The compiler will still inline the code, but because of this error it is unable to early-inline the code, and will have to late-inline the code.

The following example shows the compile line used to generate a 32-bit executable with debug information.

EXAMPLE 2–10 Compiling With `-g` to Generate Debug Information

```
cc -g -O inline32.il driver32.c
```

The utility `er_src` can be used to examine the compiler commentary for a particular file. It takes two parameters: the name of the executable and the name of the function which you wish to examine. In this case, the template which cannot be early inlined is `sum_val`. Each time the compiler comes across the `%fp` register it inserts a debug message, so you can tell that there are six instances of references to `%fp` in the template.

EXAMPLE 2–11 Using `er_src` to Output Compiler Commentary

```
er_src a.out main
Source file: /home/dg83945/book_code/inline/driver32.c
Object file: /home/dg83945/book_code/inline/driver32.o
Load Object: a.out

     1. #include <stdio.h>
     2.
     3. void do_nothing();
     4. int add_up(int v1,int v2, int v3, int v4, int v5, int v6, int v7);
     5. double sum_val(double a, double b);
     6. double sum_ref(double *a, double *b);
     7. int is_true(int i);
     8.
     9.
    10. void main()
    11. {
    12.   double a=3.11,b=7.22;
    13.   do_nothing();
    14.   printf("add_up  %i\n",add_up(1,2,3,4,5,6,7));

   Template could not be early inlined because it references the register %fp
   Template could not be early inlined because it references the register %fp
   Template could not be early inlined because it references the register %fp
   Template could not be early inlined because it references the register %fp
   Template could not be early inlined because it references the register %fp
   Template could not be early inlined because it references the register %fp
    15.   printf("sum_val %f\n",sum_val(a,b));
    16.   printf("sum_ref %f\n",sum_ref(&a,&b));
    17.   printf("is_true 0=%i,1=%i\n", is_true(0),is_true(1));
    18. }
```

Decoding the Calling Convention

The calling convention for the architecture can be a bit tricky to master. The easiest way of dealing with this is to write a test function, and see how that gets converted into assembly language.

EXAMPLE 2–12 Examining the 32–bit Calling Convention

```
# more fptest.c

double sum(double d1,double d2, double d3, double d4)
{
  return d1 + d2 + d3 + d4;
}

#cc -O -xarch=v8plusa -S fptest.c

# more fptest.s
....
                        .global sum
                       sum:
/* 000000          2 */         st      %o0,[%sp+68]
/* 0x0004            */         st      %o2,[%sp+76]
/* 0x0008            */         st      %o1,[%sp+72]
/* 0x000c            */         st      %o3,[%sp+80]
/* 0x0010            */         st      %o4,[%sp+84]
/* 0x0014            */         st      %o5,[%sp+88]

!    3                !  return d1 + d2 + d3 + d4;

/* 0x0018          3 */         ld      [%sp+68],%f2
/* 0x001c            */         ld      [%sp+72],%f3
/* 0x0020            */         ld      [%sp+76],%f10
/* 0x0024            */         ld      [%sp+80],%f11
/* 0x0028            */         ld      [%sp+84],%f4
/* 0x002c            */         faddd   %f2,%f10,%f12
/* 0x0030            */         ld      [%sp+88],%f5
/* 0x0034            */         ld      [%sp+92],%f6
/* 0x0038            */         ld      [%sp+96],%f7
/* 0x003c            */         faddd   %f12,%f4,%f14
/* 0x0040            */         retl    ! Result =  %f0
/* 0x0044            */         faddd   %f14,%f6,%f0
....
```

In the example code you can see that the first three `fp` parameters are passed in `%o0-%o5`, and that the fourth `fp` parameter is passed on the stack at locations `%sp+92` and `%sp+96`. Note that this location is 4-byte aligned, so it is not possible to use a single floating-point load double instruction to load it.

In the following example code, you can see that the first action is to load the seventh integer parameter from the stack.

EXAMPLE 2–13 Examining the 64–bit Calling Convention

```
#more inttest.c
long sum(long v1,long v2, long v3, long v4, long v5, long v6, long v7)
{
return v1 + v2 + v3 + v4 + v5 + v6 + v7;
}

# cc -O -xarch=v9 -S inttest.c

# more inttest.s
...
/* 000000          2 */         ldx     [%sp+2223],%g2
/* 0x0004          3 */         add     %o0,%o1,%g1
/* 0x0008            */         add     %o3,%o2,%g3
/* 0x000c            */         add     %g3,%g1,%g4
/* 0x0010            */         add     %o5,%o4,%g5
/* 0x0014            */         add     %g5,%g4,%o1
/* 0x0018            */         retl    ! Result =  %o0
/* 0x001c            */         add     %o1,%g2,%o0
...
```

Other Examples of Templates

Templates are used in `libm.il` (the inline math library) and in `vis.il` (the Visual Instruction Set inline library). These two files can be found in `/opt/SUNWspro/prod/lib/`. They are linked in by the compiler when flags `-xlibmil` (for the math templates) or `-xvis` (for the VIS templates) are specified. The include files which prototype the functions in the template libraries are `math.h` and `vis.h`.

Complete Source Code for 32-Bit Examples

EXAMPLE 2–14 `inline32.il` File for 32–bit Inline Template Examples

```
/* The following template does nothing*/
.inline do_nothing,0
  nop
.end

/*Add up 7 integer parameters - last one will be passed on stack*/
.inline add_up,28
  add %o0,%o1,%o0
  ld [%sp+0x5c],%o1
  add %o2,%o3,%o2
  add %o4,%o5,%o4
  add %o0,%o1,%o0
  add %o2,%o4,%o2
  add %o0,%o2,%o0
.end

/*sum of two doubles by value*/
.inline sum_val,16
```

EXAMPLE 2–14 `inline32.il` File for 32–bit Inline Template Examples *(Continued)*

```
  st   %o0,[%sp+0x48]
  st   %o1,[%sp+0x4c]
  ldd  [%sp+0x48],%f0
  st   %o2,[%sp+0x48]
  st   %o3,[%sp+0x4c]
  ldd  [%sp+0x48],%f2
  faddd %f0,%f2,%f0
.end



/*sum of two doubles by reference*/
.inline sum_ref,16
  ldd [%o0],%f0
  ldd [%o1],%f2
  faddd %f0,%f2,%f0
.end

/*return whether true*/
.inline is_true,4
   cmp  %o0,%g0
   bne  1f
   nop
   mov  1,%o0
   ba   2f
   nop
1:
   mov  0,%o0
2:
.end
```

EXAMPLE 2–15 `driver32.c` Source File for 32–bit Examples

```
#include <stdio.h>

void do_nothing();
int add_up(int v1,int v2, int v3, int v4, int v5, int v6, int v7);
double sum_val(double a, double b);
double sum_ref(double *a, double *b);
int is_true(int i);


void main()
{
  double a=3.11,b=7.22;
  do_nothing();
  printf("add_up  %i\n",add_up(1,2,3,4,5,6,7));
  printf("sum_val %f\n",sum_val(a,b));
  printf("sum_ref %f\n",sum_ref(&a,&b));
  printf("is_true 0=%i,1=%i\n", is_true(0),is_true(1));
}
```

Complete Source Code for 64-Bit Examples

EXAMPLE 2–16 `inline64.il` Template File for 64–bit Template Examples

```
/* The following template does nothing*/
.inline do_nothing,0
  nop
.end

/*Add up 7 integer parameters - last one will be passed on stack*/
.inline add_up,56
  add %o0,%o1,%o0
  ldx [%sp+0x8af],%o1
  add %o2,%o3,%o2
  add %o4,%o5,%o4
  add %o0,%o1,%o0
  add %o2,%o4,%o2
  add %o0,%o2,%o0
.end

/*sum of two doubles 64-bit calling convention*/
.inline sum,16
  faddd %f0,%f2,%f0
.end

/*return whether true*/
.inline is_true,4
   cmp  %o0,%g0
   bne  1f
   nop
   mov  1,%o0
   ba   2f
   nop
1:
   mov  0,%o0
2:
.end
```

EXAMPLE 2–17 `driver64.c` Source File for 64–bit Examples

```
#include <stdio.h>

void do_nothing();
int add_up(int v1,int v2, int v3, int v4, int v5, int v6, int v7);
double sum(double a, double b);
int is_true(int i);


void main()
{
  double a=3.11,b=7.22;
  int v1=1, v2=2, v3=3, v4=4, v5=5, v6=6, v7=7;
  do_nothing();
  printf("add_up  %i\n",add_up(v1,v2,v3,v4,v5,v6,v7));
  printf("sum    %f\n",sum(a,b));
  printf("is_true 0=%i,1=%i\n", is_true(0),is_true(1));
}
```

Running Examples

EXAMPLE 2–18 Compile and Run Sequence for the Examples

```
%cc -O driver32.c inline32.il
% a.out
add_up  28
sum_val 10.330000
sum_ref 10.330000
is_true 0=1,1=0

% cc -O -xarch=v9 driver64.c inline64.il
% a.out
add_up  28
sum    10.330000
is_true 0=1,1=0
```

http://developers.sun.com/solaris/articles/inlining.html

Crossfile Inlining and Inline Templates

Darryl Gove, May 15, 2008

Found an interesting "feature" of using crossfile (`-xipo`) optimization together with inline templates (`http://developers.sun.com/solaris/articles/inlining.html`). Suppose you have a library routine which is defined in one file and uses an inline template. This library routine is used all over the code. Here's an example of such a routine:

```
int T(int);
int W(int i)
{
 return T(i);
}
```

The routine `W` relies on an inline template (`T`) to do the work. The inline template contains some code like:

```
.inline T,0
  add  %o0,%o0,%o0
.end
```

The `main` routine resides in another file, and uses the routine `W`:

```
#include <stdio.h>

int W(int);
void main()
{
  printf("%i\n",W(9));
}
```

To use inline templates, you compile the file that contains the call to the inline template together with the inline template that it calls - like this:

```
$ cc -c -xO4 m.c
$ cc -c -xO4 w.c t.il
$ cc -xO4 m.o w.o
```

However, when crossfile optimization (`-xipo`) is used, the routine `W` is inlined into `main`, and now `main` has a dependence on the inline template. But when `m.o` is recompiled after `W` has been inlined into `main`, the compiler cannot see the inline template for `T` because it was not present on the initial compile line for `m.c`. The result of this is an error like:

```
$ cc -c -xO4 -xipo m.c
$ cc -c -xO4 -xipo w.c t.il
$ cc -xO4 -xipo m.o w.o
Undefined                       first referenced
 symbol                             in file
T                                   m.o
ld: fatal: Symbol referencing errors. No output written to a.out
```

As you might guess from the above description, the workaround is not intuitive. You need to add the inline template to the initial compile of the file `m.c` :

```
$ cc -c -xO4 -xipo m.c t.il
$ cc -c -xO4 -xipo w.c t.il
$ cc -xO4 -xipo m.o w.o
```

It is not sufficient to add the inline template to the final compile line.

Looking beyond the simple test case shown above, the problem really is that when crossfile optimization is used, the developer is no longer aware of the places in the code where inlining has happened (which is as it should be). So the developer can't know which initial compile lines to add the inline template to.

Hence, the conclusion is that whenever you are compiling code that relies on inline templates with crossfile optimization, it is necessary to include the inline template on the compile line of every file.

http://blogs.sun.com/d/entry/crossfile_inlining_and_inline_templates

Static and Inline Functions

Darryl Gove, April 28, 2008

Hit a problem when compiling a library. The problem is with mixing static and inline functions, which is not allowed by the standard, but is allowed by GCC. Example code looks like:

```
char * c;

static void foo(char *);
```

```
inline void work()
{
  foo(c);
}

void foo(char* c)
{
}
```

When this code is compiled it generates the following error:

```
% cc s.c
"s.c", line 7: reference to static identifier "foo" in extern inline function
cc: acomp failed for s.c
```

It turns out that there is a workaround for this problem, which is the flag `-features=no%extinl`. Douglas Walls describes the issue in much more detail (`http://blogs.sun.com/dew/entry/c99_inline_function`).

`http://blogs.sun.com/d/entry/static_and_inline_functions`

C99 Inline Function and the Sun C Compiler

Douglas Walls, May 22, 2006

The C standard says that inline is only a suggestion to the C compiler. The C compiler can choose not to inline anything, and attempt to call the actual function.

The Sun C compiler does not inline C function calls unless optimizing at `-x03` or above. And that inlining is done by the backends. And then only if the backend's heuristics decide it is profitable to do so. The Sun C compiler gives no way to force a function to be inlined.

For static inline functions it is simple. Either a function defined with the inline function specifier is inlined at a reference or a call is made to the actual function. The compiler can choose which to do at each reference. The Sun C compiler decides if it is profitable to inline at `-x03` and above. When not profitable to inline, or at an optimization of less than `-x03`, a reference to the actual function will be generated. If any reference to the actual function is generated, the function definition will be generated in the object code. Note if the address of the function is taken, the actual function will be generated in the object code.

Extern inline functions are more complicated. There are two types of extern inline functions: an inline definition which never provides an extern (global) definition of the function, and an extern inline function which always provide a global definition of the function. The C99 standard describes the situation in ISO/IEC 9899, Programming Languages -- C, Subclause 6.7.4, paragraphs 6 thru 8.

For an inline definition, the programmer is required to supply an extern definition of the function in another translation-unit for references to the function that are not inlined.

For an inline definition, the compiler must *not* create a global definition of the function. That means any reference to an inline definition that is not inlined must be a reference to a global function defined elsewhere. Put another way, the object file produced by compiling this translation unit will *not* contain a global symbol for the inline definition. And any reference to the function that is not inlined will be to an extern (global) symbol provided by some other object file or library at link time.

For an extern inline function declared by a file scope declaration with the extern storage-class-specifier (that is, the function definition and/or prototype), the compiler must provide a global definition of the function in the resulting object file. The compiler can choose to inline any references to that function seen in the translation unit where the function definition has been provided, or the compiler can choose to call the global function.

The behavior of any program that relies on whether or not a function call is actually inlined, is undefined.

Note also an inline function with external linkage may not declare or reference a static variable anywhere in the translation-unit.

Definition of translation-unit: A source file and all of its includes, recursively.

Like it does for static functions, the Sun C compiler decides if it is profitable to inline a reference to an inline definition or an extern inline function at `-x03` and above. When not profitable to inline, or at an optimization of less than `-x03`, a reference to the global function will be generated. Likewise, a reference to the address of the function is always a reference to the global function.

The rules for C++ differ: a function which is inline anywhere must be inline everywhere and must be defined identically in all the translation units that use it.

`gcc` C Rules

The GNU C rules differ and are described in the GNU C manual, which can be found at `http://gcc.gnu.org/`.

- A function defined with inline and without either extern or static keywords will always cause a global function to be emitted. `gcc` may inline references to the function within the object code for the translation-unit in which it appears. And references to the function from other translation-units in the program will be satisfied by linking with this object code.
- For a function defined with "extern inline" a global function is never emitted. References to the function are either inlined or are made to a global function that must be defined somewhere else.
- A function defined with "static inline" acts the same as C99.

Sun C Compiler `gcc` Compatibility for Inline Functions

To obtain behavior from the Sun C compiler that is compatible with `gcc`'s implementation of extern inline functions for most programs, use the `-features=no%extinl` flag. When this flag is specified, the Sun C compiler will treat the function as if it was declared as a static inline function.

The one place this is not compatible will be when the address of the function is taken. With `gcc`, this will be an address of a global function, and with the Sun compiler the local static definition address will be used.

`http://blogs.sun.com/dew/entry/c99_inline_function`

Catching Security Vulnerabilities in C Code

Douglas Walls, February 21, 2006

Check out what Sun Studio C compiler has provided for detect coding practices that could lead to security vulnerabilities. Specifically, Sun added security vulnerability checking to lint (`http://docs.sun.com/source/819-3688/lint.html`), the C program checker.

Below is an overview of the flag to specify on the `lint` command to obtain security vulnerability checking. And here is a testimonial (`http://blogs.sun.com/roller/page/rotondo/20050614`) about how it is used in the Solaris sources.

```
lint -errsecurity=core
```

This flag checks for source code constructs that are almost always either unsafe or difficult to verify. Checks at this level include:

- Use of variable format strings with the `printf`() and `scanf`() family of functions
- Use of unbounded string (%s) formats in `scanf`() functions
- Use of functions with no safe usage: `gets`(), `cftime`(), `ascftime`(), `creat`()
- Incorrect use of `open`() with `O_CREAT`

Consider source code that produces warnings at this level to be a bug. The source code in question should be changed. In all cases, straightforward safer alternatives are available.

```
lint -errsecurity=standard
```

Includes all checks from the core level plus constructs that may be safe but have better alternatives available. This level is recommended when checking newly written code. Additional checks at this level include:

- Use of string copy functions other than `strlcpy`()

- Use of weak random number functions
- Use of unsafe functions to generate temporary files
- Use of `fopen`() to create files
- Use of functions that invoke the shell

Replace source code that produces warnings at this level with new or significantly modified code. Balance addressing these warnings in legacy code against the risks of destabilizing the application.

```
lint -errsecurity=extended
```

Contains the most complete set of checks, including everything from the Core and Standard levels. In addition, a number of warnings are generated about constructs that may be unsafe in some situations. The checks at this level are useful as an aid in reviewing code but need not be used as a standard with which acceptable source code must comply. Additional checks at this level include:

- Calls to `getc`() or `fgetc`() inside a loop
- Use of functions prone to pathname race conditions
- Use of the `exec`() family of functions
- Race conditions between `stat`() and other functions

Review source code that produces warnings at this level to determine if the potential security issue is present.

`http://blogs.sun.com/dew/entry/catching_security_vulnerabilities_in_c`

◆ ◆ ◆ CHAPTER 3

Hardware-Specific Topics

This chapter covers some hardware-specific topics. It contains information about the memory models introduced in the EMT64 instruction set extension for x86 processors, as well as detailed coverage of how SPARC processors handle accesses to misaligned memory addresses.

AMD64 Memory Models

Alfred Huang, January 22, 2006

I recall the first production quality compiler I worked on was for an 80286 in the mid 80s. Memory model such as "small," "medium" and "large" with common extended keywords "__far" and "__near" were popular with that 16-bit segmented memory architecture. Thereafter, 80386 "linearized" the address space with its 32-bit pointers and the model-related terms simply became obsolete and disappeared.

It is interesting to see that AMD64 ABI reintroduced the memory models to the x86 world, but this time with a 64-bit architecture. So why is there memory model in the x64 architecture? I would guess it is for performance's sake. The x64 architecture basically has only one instruction that truly loads a 64-bit address to register, namely, `movabsq`. All other memory-related instructions contain only 4 bytes displacement which are then extended to 64-bit. If an address greater than 32-bits in the memory space is to be accessed, the compiler needs to load that "far" address with a `movabsq` instruction and then reference it indirectly, which is not as efficient as a single, direct access. Hence, the specified models of "small" and "kernel" in the x64 ABI assume certain address range limitations, so as to allow efficient memory access. "Medium" and "large" allows more flexibility with the address ranges at the expense of less efficient memory access.

To save time, let's talk about "small" and "medium" models only. Because "small" is the default of most x64 compilers, it is often mistakenly believed to be equivalent to the de facto model in 32-bit x86, if there were one. I'm afraid the small model of x64 is actually smaller than the de facto 32-bit x86. We have encountered many people complaining that their applications ran in 32-bit x86, but ran into the linker's "address does not fit" error when ported to the default x64.

According to the AMD64 ABI, the small model allows a data address space of [-2^{31}, 2^{31}-1], with the linker limiting allocation of symbols between [0, 2^{31}-2^{24}-1]. What that means is

Effective address (that is, offset+symbol at runtime) is limited to
[-2^{31}, 2^{31}-1]
If the "symbol" in offset+symbol is limited by the linker to
[0, 2^{31}-2^{24}-1]
The compiler can safely generate "offset" in the offset+symbol equation to
[-2^{31}, 2^{24}]

Note that the EA limitation means the upper 33 bits should be all 0s or all 1s to be valid. The ABI stated that the linker must issue an error otherwise. So, only 31 bits are truly eligible for memory address computation in the x64 small model, versus 32 bits in 32-bit x86, roughly half the space.

So what should be done when you have a linker "address does not fit" error? If changing the source code is not an option, one option in Sun Studio 11 is to try the `-xmodel=medium` option.

In the recent x64 ABI, the medium model is actually quite efficient. Not all static data are accessed with the "far" load 64-bit address followed by indirect reference. The medium model is now defined with extra data sections. Normally, there are the ".data", ".bss," etc data sections. Under the medium model, data objects larger than 65535 bytes are allocated in the corresponding ".ldata"and ".lbss" data sections. Data in the "normal" data sections are referenced efficiently with direct access, whereas data in the ".l" data sections are referenced with indirect access. This means a performance hit may be minimized while the data access range can be increased.

http://blogs.sun.com/alblog/entry/amd64_memory_models

-Kpic Under Small-Model Versus Medium-Model Code

Alfred Huang, January 31, 2006

Continuing on with my previous discussion of medium model. Some people pointed out their previously "address does not fit" application actually would link and run using `-Kpic` without using `-xmodel=medium`.

Yes, Position Independent Code may buy you farther addressing capability, but it has a limit based on code size and number of global statics and may not be as efficient as the medium-model code.

So what is the difference between Position Independent Code under small-model and medium-model code?

PIC code goes through the Global Offset Table (GOT), which the linker usually creates beneath the text section. It contains the actual 64-bit addresses of the static objects. Access to an object under the PIC mode consists of a 4-byte displacement reference from the referencing point in

the text to the corresponding entry in the GOT The 64-bit address in the GOT is then picked up and referenced indirectly. With the full 64-bit address referenced indirectly, the limit of 2G addressability (as discussed in my previous blog entry) is overcome. In a sense, it is pretty similar to that of medium model, where a full 64-bit address is explicitly loaded using the `movabsq` instruction and then referenced indirectly.

The difference is that PIC code requires the distance between the referencing point and the actual GOT entry to be within the 2G limit. A reference from the lower address of a very large text section to its corresponding GOT entry may become out of reach, leading to another "address does not fit" error.

Moreover, under PIC mode, all global statics will be accessed through GOT, leading to a degradation in performance, whereas under medium model, only objects larger than 65535 bytes will reside in the special ".lXXXX" sections which require 64-bit access.

Hence, for applications with normal code size and number of global static, using `-Kpic` will get around the problem if some degradation in performance can be tolerated.

`http://blogs.sun.com/alblog/entry/kpic_under_small_model_versus`

A Look Into AMD64 Aggregate Argument Passing

Alfred Huang, February 21, 2006

Recently there have been questions on argument passing for AMD64. As part of the calling convention, argument passing and returned values are described in detail in the AMD64 ABI. The portion on passing scalar arguments is clear and straightforward, but the description for passing aggregates is pretty algorithmic and rather obscure. Maybe I can help by explaining it with examples.

Generally speaking, all objects that can be accommodated in registers will be passed in registers until the designated registers run out and the memory stack is then used. Regardless of the actual object size, all arguments are passed in a multiple of 8 bytes.

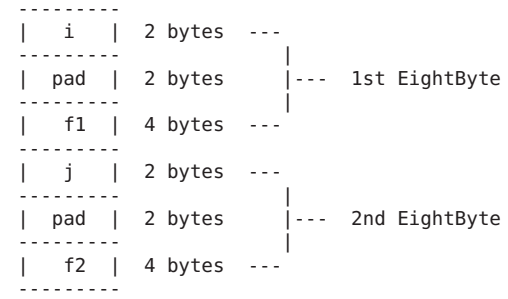
To start the topic on passing aggregates, let me review the argument passing of the most common scalar types, namely integers and floating point types. The first six integer types (class INTEGER) are passed in `%rdi`, `%rsi`, `%rcx`, `%rdx`, `%r8`, `%r9`, then on the memory stack. Likewise, the first 8 float and double types (class SSE) are passed in `%xmm0` to `%xmm7`, then on the memory stack.

Aggregate arguments larger than 16 bytes (2 EightBytes) are always passed on stack. It is the aggregate argument smaller than or equal to 16 bytes that is the most interesting. First of all, you have to figure out the fields of the aggregate belonging to the 1st and 2nd EightBytes. This can be achieved with the knowledge of the possible padding used in between the fields of a struct.

Example 1:

```
struct S { short i;
           float f1;
           short j;
           float f2;
         } s;
```

Since the alignment of `f1` and `f2` is 4, there is a padding of 2 bytes between `i` and `f1`, and between `j` and `f2`. So we have:

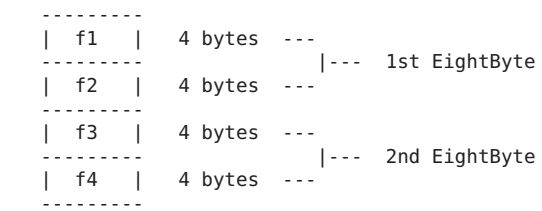


Now the rule in the AMD64 ABI says to consider two adjacent fields in an EightByte recursively in a merge step. I will not repeat the rules here, but one of the rules is that if one class is INTEGER, the result class is INTEGER. In this case, since `i` is of class INTEGER and `f1` is of class SSE, the result is INTEGER. Hence the first EightByte has class INTEGER and the second EightByte also has class INTEGER. If object `s` is passed as the first argument, it will then be passed in `%rdi` and `%rsi`, in which `i` and `f1` are contained in `%rdi`, while `j` and `f2` are contained in `%rsi`.

Example 2:

```
struct S { float f[4] } s;
```

Since `s` is 128 bits in size, which exactly fits an `xmm` register, would `s` be passed in a single `xmm` register? The answer is no, as shown here:



Since `f1` and `f2` are both of class SSE, the result class for the first EightByte is SSE, and likewise for the second EightByte. Hence if object `s` is passed as the first argument, it will then be passed in `%xmm0` and `%xmm1`, where `%xmm0` contains the value of `f1` and `f2`, while `%xmm1` contains the value of `f3` and `f4`.

Example 3:

Should the entire aggregate reside in one single class of register when being passed? Again the answer is no. Consider the following case:

```
struct S { int i;
           float f1;
           float f2;
           float f3;
         } s;

   ---------
   |  i    |   4 bytes  ---
   ---------               |---  1st EightByte
   |  f1   |   4 bytes  ---
   ---------
   |  f2   |   4 bytes  ---
   ---------               |---  2nd EightByte
   |  f3   |   4 bytes  ---
   ---------
```

Note that `i` is of class INTEGER and `f1` is of class SSE. As one of the merge rules says, if one class is INTEGER, the result is INTEGER, so the first EightByte is of class INTEGER, while the second EightByte is of class SSE. Hence, if object `s` is passed as the first argument, its first 8 bytes containing `i` and `f1` are passed in `%rdi`, whereas the remaining 8 bytes containing `f2` and `f3` are passed in `%xmm0`.

Hope these little examples provide some insights into the interpretation of the aggregate argument passing rule in the AMD64 ABI.

http://blogs.sun.com/alblog/entry/amd64_aggregate_argument_passing

On Misaligned Memory Accesses

Darryl Gove, June 1, 2006

The UltraSPARC processor does not handle misaligned memory operations (loads and stores) in hardware. An application can be compiled to either crash on a misaligned memory accesses, or trap to software to correct the alignment. A further option is that an application can be compiled to assume that data is always misaligned, and use multiple loads or stores for each memory operation.

The behavior is controlled through the `-xmemalign` flag as follows:

- `-xmemalign=1s` is equivalent to the old flag `-misalign`, which assumes that everything is misaligned, and generates multiple loads or stores to access memory. This option is appropriate if *most* memory operations are misaligned (this is rarely the case).
- `-xmemalign=8i` This has been the default for SPARC v8 applications since Sun Studio 9. The compiler assumes that all memory operations have the correct alignment, but uses a trap handler to correct the situations where this is not true. There is an overhead from using a trap handler to correct alignment problems.

- `-xmemalign=8s` equivalent to the old flag `-dalign`. This is the default for SPARC v9 applications. This is included in `-fast`. It means that the compiler should assume correct alignment, but the application will crash if this is not the case.

To port an application which may or may not have alignment issues, the appropriate flag is the `-xmemalign=8i` flag which is enabled by default for 32–bit code. The code will work, but may run slower due to having correct alignment issues in software. So the obvious question is how to detect whether there is a problem with misaligned memory accesses.

The first approach might be to compile with `-xmemalign=8s` and see if the application runs. This would be a slow and rather painful way of testing this. Fortunately there are other options.

The debugger has the facility (available as a command line) to check for misaligned memory accesses. The tool to use is:

```
bcheck <app> <params>
```

The following test code has a misalignment problem:

```
#include <stdio.h>

void main ()
{
  char a[10];
  double *d;
  d=(double*)&a[1];
  *d=5.0;
  printf("%f",d);
}
```

This code can be compiled and run under `bcheck`:

```
$ cc -g -O -xmemalign=8i miss.c
$ bcheck -all a.out
...

signal SEGV (no mapping at the fault address) in main (optimized) at line 6 in file "miss.c"
    6     *d=5.0;
dbx: read of 4 bytes at address 20 failed -- Error 0
```

The code was compiled to correct misalignment problems, but under `bcheck` it failed because of a misaligned memory access. This is due to differences in the way that misalignment is handled in 32-bit and 64-bit modes. In 32-bit mode the kernel handles the alignment, in 64-bit the kernel passes control back to user-code to handle the alignment. So `bcheck` cannot capture the misalignment issue in 32-bit mode. The following example shows that it can be done in 64-bit mode:

```
$ cc -g -O -xtarget=generic64 -xmemalign=8i miss.c
$ bcheck -all a.out
...
  errors are being redirected to file 'a.out.errs'
```

```
...

$ more a.out.errs
<rtc> Misaligned write (maw):
Attempting to write 4 bytes at address 0xffffffff7fffeec5
    which is 197 bytes above the current stack pointer
=>[1] main() (optimized), at 0x10000261c (line ~6) in "miss.c"

<rtc> Read from unallocated (rua):
Attempting to read 4 bytes at address 0x100106e78
    which is 17480 bytes into the heap; no blocks allocated
=>[1] __do_misaligned_ldst_instr(0xffffffff7fffed40, 0xffffffff7fffee00, 0x100106e78,
      0xffffffff7fffe601, 0x10, 0x1), at 0x100000d74
  [2] __misalign_trap_handler(0x100102a18, 0xffffffff7fffe6d1, 0x2, 0x0, 0x11e52c, 0xd0), at 0x1000020a0
  [3] __rtc_dispatch(0x1, 0x100102000, 0x100102, 0x100000, 0x100002000, 0x100002), at 0xffffffff7352e910
  [4] main() (optimized), at 0x1000025f8 (line ~2) in "miss.c"
```

The report indicates a Misaligned write (maw), so it has successfully detected the misalignment problem. The report also runs the code to completion, so all the problems are captured.

Of course not all programs can be ported to 64 bits just to check for the location of misaligned memory accesses. In fact, if the misaligned memory accesses are not causing a performance problem, there's no real reason to hunt them down and remove them. The way to check this is to profile the application and see whether there's a lot of time required that might be due to misaligned memory accesses.

The following test program spends a bit more time on misaligned memory accesses, which will make the problem more apparent when the program is profiled:

```
#include <stdio.h>

static int i,j;
double *d;
void main ()
{
  char a[10];
  d=(double*)&a[1];
  j=1000000;
  for (i=0;i<j; i++)
  *d=5.0;
  printf("%f",d);
}
```

First of all, profile the program built as a 32-bit executable:

```
$ cc -xO1 -g miss.c
$ collect a.out
$ er_print -metrics e.user:e.system -dis main test.2.er
   Excl.     Excl.
   User CPU  Sys. CPU
    sec.      sec.
...
                             9.   *d=5.0;
                            10.   printf("%f",d);
...
```

```
   0.        0.                [ 9]    10ccc:  std         %f32, [%o2]
## 0.370     1.081             [ 8]    10cd0:  add         %l5, 696, %l6
   0.        0.                [ 9]    10cd4:  add         %g2, 64, %g3
```

In 32-bit mode, there is no userland handler for misaligned loads. The alignment is corrected in the kernel, so system time is an indicator that there could be problems with alignment. The time spent correcting the misaligned store is shown as system time on the following instruction.

The same test can be performed in 64-bit mode:

```
$ cc -g -O -xtarget=generic64 -xmemalign=8i miss.c
$ collect a.out
$ er_print -metrics e.user:e.system -func test.4.er
Functions sorted by metric: Exclusive User CPU Time

Excl.     Excl.      Name
User CPU  Sys. CPU
 sec.      sec.
0.570     0.         <total>

0.460     0.         __misalign_trap_handler
0.050     0.         __do_misaligned_ldst_instr
0.050     0.         __fp_read_pdreg
0.010     0.         main
0.        0.         _start
```

In the 64-bit case, much time is spent in the trap handler that corrects misalignment. This is a good indication that misaligned memory accesses are a problem, but does not indicate where the problem is.

```
er_print -dis main test.4.er
   Excl.     Incl.
   User CPU  User CPU
    sec.      sec.
...
   0.        0.                [ 8] 100002738:  or          %l6, 258, %l7
## 0.        0.560             [ 9] 10000273c:  std         %f32, [%o7]
   0.        0.                [ 8] 100002740:  or          %g2, 258, %g3
...
```

To determine the place where the misaligned memory accesses are occurring, it is necessary to look at the call stack for the trap handler. This quickly points to the main routine being the location. Inspecting the disassembly shows that there is Inclusive (but not exclusive) user time attributed to the store instruction, which demonstrates that is the instruction that is accessing misaligned data.

http://blogs.sun.com/d/entry/on_misaligned_memory_accesses

The Meaning of `-xmemalign`

Darryl Gove, February 1, 2008

I made some comments on a thread on the forums about memory alignment (`http://forum.java.sun.com/thread.jspa?threadID=5256806`) on SPARC and the `-xmemalign` flag. I've talked about memory alignment (`http://blogs.sun.com/d/entry/on_misaligned_memory_accesses`) before, but this time the discussion was more about how the flag works. In brief:

- The flag has two parts: `-xmemalign=[1|2|4|8][i|s]`
- The number specifies the alignment that the compiler should assume when compiling an object file. So if the compiler is not certain that the current variable is correctly aligned (say it's accessed through a pointer) then the compiler will assume the alignment given by the flag. Take a single-precision floating-point value that takes four bytes. Under `-xmemalign=1[i|s]` the compiler will assume that it is unaligned, so it will issue four single-byte loads to load the value. If the alignment is specified as `-xmemalign=2[i|s]` the compiler will assume two-byte alignment, so it will issue two loads to get the four-byte value.
- The suffix `[i|s]` tells the compiler how to behave if there is a misaligned access. For 32-bit codes the default is `i`, which fixes the misaligned access and continues. For 64-bit codes the default is `s`, which causes the app to die with a SIGBUS error. This is the part of the flag that has to be specified at link time because it causes different code to be linked into the binary depending on the desired behavior.

`http://blogs.sun.com/d/entry/the_meaning_of_xmemalign`

Identifying Misaligned Loads in 32-Bit Code Using DTrace

Darryl Gove, June 12, 2007

A previous blog entry talks about handling and detecting misaligned memory accesses (`http://blogs.sun.com/d/entry/on_misaligned_memory_accesses`). For 64-bit code, this is easy to achieve using the Performance Analyzer. For 32-bit code, the analysis is a bit more tricky. Fortunately it is possible to do the 32-bit analysis with `dtrace`

Consider the following program, which has a misaligned memory access. The default mode of the compiler (since Sun Studio 9) will compile the binary to trap to fix the misalignment and continue.

```
% more align.c
void main()
{
  volatile char a[10];
  int i;
```

```
  for (i=0; i<100000000; i++) {(*(int*)(&a[1]))++;}
}
```

The following DTrace script will instrument the misaligned data access trap handler and report all the pids that trigger this.

```
% more tr.d
fbt::do_unaligned:entry
{
  @p[pid]=count();
}
```

It can be run with

```
% sudo dtrace -s tr.d
dtrace: script 'tr.d' matched 1 probe
^C


    14873           260932
```

The script returns the pid (14873) which is having misalignment issues. This information is useful, in that it is trivial to recompile the binary with a different setting for `-xmemalign` and avoid the behavior. But it would be very useful to know where the traps are occurring in the binary - perhaps most of the traps only happen in one place, and that place can be fixed in the source.

```
% more tr.d
fbt::do_unaligned:entry
{
  @[ustack()]=count();
}
```

This script produces output that identifies the locations in the binary where the traps are being generated. For the simple test code there are two locations: the load and the store.

```
sudo dtrace -s tr.d
dtrace: script 'tr.d' matched 1 probe
^C
              align`main+0x10
              align`_start+0x108
            130466

              align`main+0x18
              align`_start+0x108
            130466
```

The disassembly for the loop is as follows:

```
main()
        10b80:  9d e3 bf 90  save       %sp, -112, %sp
...
        10b90:  d0 07 bf f7  ld         [%fp - 9], %o0  <<<<<< misaligned
```

```
10b94:  90 02 20 01  inc        %o0
10b98:  d0 27 bf f7  st         %o0, [%fp - 9]  <<<<<< misaligned
10b9c:  ba 07 60 01  inc        %i5
10ba0:  80 a7 40 09  cmp        %i5, %o1
10ba4:  06 bf ff fb  bl         main+0x10       ! 0x10b90
10ba8:  01 00 00 00  nop
```

http://blogs.sun.com/d/entry/identifying_misaligned_loads_in_32

Calculating Processor Utilization From the UltraSPARC T1 and UltraSPARC T2 Performance Counters

Darryl Gove, September 2007

Summary

Using the performance counters for the UltraSPARC T1 and UltraSPARC T2 processors to estimate core load and find potential areas for performance improvement.

Introduction

The UltraSPARC T1 and UltraSPARC T2 processors are designed for high throughput, and as such they replicate a very simple core multiple times so that the processor can handle many threads. The UltraSPARC T1 processor has eight cores, and each core can issue one instruction per cycle and supports four threads, making a total of 32 virtual processors. The UltraSPARC T2 processor has eight cores, and each core can issue two instructions per cycle and can support eight threads, making a total of 64 virtual processors. The peak performance of the processor can be calculated by multiplying the frequency by the number of cores and the number of instructions that can be issued for each core. Assuming that both processors run at 1.4GHz, then the UltraSPARC T1 processor can sustain 1.4*8 = 11.2 billion instructions per second, and the UltraSPARC T2 processor can sustain 22.4 billion instructions per second.

Because multiple threads are sharing a single core, the question of whether the core is fully loaded or not becomes interesting. For example, suppose that the core is fully loaded. That means each thread should be getting its fair share of the available instruction slots. So each thread should be able to issue an instruction every four cycles. In this instance it becomes interesting to ask whether running fewer threads on the core would improve the latency of the application. Alternatively, if not all the threads on a particular core are busy, it is interesting to ask whether the core has sufficient instruction issue capacity to handle additional threads.

This article examines the issue of utilization of UltraSPARC T1 and UltraSPARC T2 cores and attempts to determine whether performance would benefit from fewer or more virtual processors being assigned work.

Instruction Utilization

It is relatively easy to determine the instruction issue rate on a system-wide basis using the `cpustat` command (with superuser privileges). The following code example shows using `cpustat` to collect instruction count data once every second for 10 seconds.

```
$ cpustat -c Instr_cnt,sys 1 10


time cpu event pic1
1.009 22 tick 10298
1.009 23 tick 10744
1.009 24 tick 55105
1.009 11 tick 11154
1.009 5 tick 483468
1.009 26 tick 10731
1.009 21 tick 79061
1.009 16 tick 83529
1.009 18 tick 22184
1.009 2 tick 41845
....
```

The `cpustat` output has one line per CPU. It is relatively easy to post-process this output to provide formatted output showing the utilization of each core as a percentage of the maximum possible issue rate for the core. (The utility `psrinfo -v` reports the processor type and clock speed information necessary for this calculation.) In fact the downloadable tool corestat (`http://cooltools.sunsource.net/corestat/`) already does this calculation.

Information on the instruction count is useful in determining whether the core is fully saturated, and as such whether there is spare capacity for adding additional threads, or whether it might be possible to improve latency by reducing the number of active threads.

Floating-Point Computation

The UltraSPARC T1 processor has a single floating-point unit, and the UltraSPARC T2 processor has one floating-point unit per core. Consequently a similar question can be asked about the utilization of the floating-point unit. Both processors have a counter to record floating-point instructions. The result of the `cpustat` command, showing both the floating-point instruction count and the total instruction count, is shown in the following code example.

```
$ cpustat -c FP_instr_cnt,sys,Instr_cnt,sys 1 10

time cpu event pic0 pic1
1.011 26 tick 0 10642
1.011 6 tick 4 31433
1.011 12 tick 50 467295
1.011 21 tick 12 33915
1.011 14 tick 0 10304
1.011 23 tick 0 10496
...
```

Again, it is possible to calculate utilization for the floating-point units of both processor types.

Stall Budget Utilization

There is a difference between traditional processors and the UltraSPARC T1 and UltraSPARC T2 processors in the way that they respond to optimization. On a traditional processor, optimization of the code typically involves identifying the cause of processor stalls and working to eliminate these stalls. For example, you can add prefetch instructions to reduce cache misses and the corresponding memory stall time.

However, on the UltraSPARC T1 and UltraSPARC T2 processors these stall cycles are used by the other threads that are sharing the core. So the stall cycles can no longer be looked at as places for optimization. It is still interesting to know about the stall cycles, but removing stall cycles from a single thread will not necessarily lead to a direct improvement in throughput, although it may help the latency of the thread.

To explain this situation, consider the fact that under an ideal load, each thread gets to issue an instruction every four cycles. This means that for every instruction, the thread has three cycles during which it cannot issue an instruction. If the thread has more than three cycles of stall for every instruction, then it ends up getting less than its fair share of the instruction budget. If the instruction has fewer than three cycles of stall, it does not follow that it will be able to issue instructions any more frequently, because the other three threads might still have instructions to be issued.

Another way of looking at this is to imagine that each thread has an instruction budget that corresponds to the number of instructions that it should expect to issue per second, under fair load. Each thread also has a stall budget which corresponds to the number of cycles per second that the thread can be stalled before the stall events will start to reduce the number of instructions executed. This stall budget is three times the instruction budget.

Various events, such as cache misses, will cause stalls, and consequently use up this budget of stall cycles. Therefore it is of interest to know the large contributors to stall. If the counter counts events rather than cycles, it is necessary to multiply the count of events by the estimated cost per event. These costs are processor specific.

Estimating Stall Budget Usage for the UltraSPARC T1 Processor

Unfortunately, most of the performance counters on the UltraSPARC T1 processor count events rather than cycles. The exception is the store queue counter, which counts the number of cycles in which the store queue is full. However, because of the simplicity of the pipeline, it is possible to estimate the number of cycles lost to the various stalls by multiplying the number of events by an estimate of the cost of the event, as shown in the following table.

TABLE 3–1 Estimated Number of Cycles Lost to Stalls

Counter	Comment	Cost in Cycles
`SB_full`	Cycles when store buffer is full	1
`FP_instr_cnt`	Floating-point instruction count	30
`IC_miss`	Instruction cache miss	20
`DC_miss`	Data cache miss	20
`ITLB_miss`	Instruction TLB miss	100
`DTLB_miss`	Data TLB miss	100
`L2_imiss`	Instruction fetches that miss L2 cache	100
`L2_dmiss_ld`	Loads that miss L2 cache	100
`Instr_cnt`	Instruction count	1

A simple script can be written that runs an application multiple times and calculates the number of cycles contributed to processor stall by the various events. The results of running this script on an application that has a data set that is resident in the on-chip second level cache is shown in the following table.

TABLE 3–2 Sample Results of Calculating UltraSPARC T1 Processor Stall Cycles

Comment	Cost in Cycles	Raw Count	Scaled Count	Estimated Time at 1.2GHz	Percentage of Total Runtime
Cycles when store buffer is full	1	3,626,323	3,626,323	0	1%
Floating-point instruction count	30	31	930	0	0
Instruction cache miss	20	99,418	1,988,360	0	0
Data cache miss	20	16,829,760	336,595,200	0.28	53%
Instruction TLB miss	100	139	13,900	0	0
Data TLB miss	100	25,669	2,566,900	0	0
Instruction fetches that miss L2 cache	100	3,661	36,6100	0	0
Loads that miss L2 cache	100	771,285	77,128,500	0.06	12%
Instruction count	1	87,048,958	87,048,958	0.07	14%
Cycles	1	636,000,000	636,000,000	0.53	100%

Unsurprisingly, the majority (50%) of the stall time comes from loads that miss the on-chip cache but are resident in the second level cache. A small number of loads also miss the second level cache and contribute significant time because of the additional cost of fetching data from memory. If this were real code, then focusing on improving the cache utilization and footprint would be the way to improve overall performance.

However, looking at instruction count, it is apparent that the application is using 14% of the instruction budget. An ideal application would use 25%, because there are four threads per core, and each core can issue one instruction per cycle. So although reducing the number of cache misses would improve performance, the maximum performance gain would be from 14% utilization of the instruction budget to 25% utilization, nearly doubling performance, assuming that all the threads on the core are active.

Placing this in context, on a traditional processor, all the memory stall time could potentially be converted into performance, but on a CMT processor, there is an upper bound imposed by the sharing of cycles between the multiple threads.

Estimating Stall Budget Usage for the UltraSPARC T2 Processor

The UltraSPARC T2 processor has eight threads sharing a core that is capable of issuing two instructions per cycle. This gives the same budget of instructions: each thread should be able to issue an instruction every four cycles, and has three cycles where it is unable to issue an instruction.

There is a different set of performance counters on the UltraSPARC T2 processor, with new multipliers. There is no counter of cycles spent in store queue stalls, and the floating-point instruction counter has a changed name. The following table shows the performance counters and their multipliers.

TABLE 3–3 Performance Counters for the UltraSPARC T2 Processor

Counter	Comment	Cost in Cycles
`Instr_FGU_arithmetic`	Floating-point instruction count	8
`IC_miss`	Instruction cache miss	20
`DC_miss`	Data cache miss	20
`ITLB_miss`	Instruction TLB miss	100
`DTLB_miss`	Data TLB miss	100
`L2_imiss`	Instruction fetches that miss L2 cache	100
`L2_dmiss_ld`	Loads that miss L2 cache	100

TABLE 3–3 Performance Counters for the UltraSPARC T2 Processor *(Continued)*

Counter	Comment	Cost in Cycles
`Instr_cnt`	Instruction count	1

Results from running the same code on an UltraSPARC T2 processor are shown in the following table.

TABLE 3–4 Sample Results of Calculating UltraSPARC T2 Processor Stall Cycles

Comment	Cost in Cycles	Raw Count	Scaled Count	Estimated Time at 1.4GHz	Percentage of Total Runtime
Floating-point instruction count	30	37	296	0	0
Instruction cache miss	20	39,749	794,980	0	0
Data cache miss	20	16,803,200	336,064,000	0.24	62%
Instruction TLB miss	100	29	2,900	0	0
Data TLB miss	100	6,408	640,800	0	0
Instruction fetches that miss L2 cache	100	421	42,100	0	0
Loads that miss L2 cache	100	2,062	206,200	0	0
Instruction count	1	86,756,212	86,756,212	0.06	16%
Cycles	1	5,460,000,000	5,460,000,000	0.39	100%

Again, most of the time is spent in load operations of data that is resident in the second level cache. The application is getting about 16% of the total cycles, which is still less that the theoretical peak of 25% of the total cycles. So reducing the cache misses could improve performance further. One issue that needs to be taken into consideration is that the memory pipe is shared between eight threads, so the peak performance of the application depends on there being one load for every two instructions. Otherwise, the application could be limited to issuing one instruction every eight cycles.

Ramifications for Optimizing for CMT Processors

The previous discussion shows that applications running on CMT processors have a great tolerance for cycles spent in memory stalls. On more traditional processors, memory stall times can be minimized, which leads directly to performance gains. On a CMT processor, reduction in stall times leads to performance gains only up to the point at which the process consumes 25% of the instruction issue budget. Reductions in stall events beyond that are unlikely to lead to significant performance gains, so efforts to further reduce instruction stalls are wasted work.

For CMT processors, there are three ways of improving performance, shown here in order from the most effective to the least effective:

- **Use more threads.** Each additional thread gets a new instruction issue budget. Two threads can potentially do twice the work of a single thread.
- **Reduce instruction count.** For UltraSPARC T1 and UltraSPARC T2 processors, each thread gets to issue a single instruction at a time, so the instruction count corresponds directly to the length of time it takes to complete the task. A traditional processor might be able to issue multiple instructions from a single thread in the same cycle, so some of the instructions issued could be obtained for free.
- **Reduce stall time.** This might not directly improve performance because stall time on one thread is an opportunity for another thread to do work. When the core is issuing its peak instruction rate there are no possible performance gains from reducing cycles spent on stall events. Of course, reducing stall time on one of the threads might enable that thread to get its fair share of the instruction budget, and it might be possible to reduce the latency of one of the threads, but it will not have an impact on the throughput of the system.

http://developers.sun.com/solaris/articles/t1t2_perf_counter.html

When to Use `membars`

Darryl Gove, May 7, 2008

`membar` instructions are SPARC assembly language instructions that enforce memory ordering. They tell the processor to ensure that memory operations are completed before it continues execution. However, the basic rule is that the instructions are usually only necessary in "unusual" circumstances - which fortunately will mean that most people don't encounter them.

The UltraSPARC Architecture manual (http://opensparc-t2.sunsource.net/specs/UA2007-current-draft-HP-EXT.pdf) documents the situation very well in section 9.5. It gives these rules which cover the default behavior:

- Each load instruction behaves as if it were followed by a `MEMBAR #LoadLoad` and `#LoadStore`.
- Each store instruction behaves as if it were followed by a `MEMBAR #StoreStore`.
- Each atomic load-store behaves as if it were followed by a `MEMBAR #LoadLoad`, `#LoadStore`, and `#StoreStore`.

There's a table in section 9.5.3 which covers when `membars` are necessary. Basically, `membar` s are necessary for ordering of block loads and stores, and for ordering non-cacheable loads and stores. There is an interesting note where it indicates that a `membar` is necessary to order a store followed by a load to a different addresses; if the address is the same, the load will get the correct data. This at first glance seems odd – why worry about whether the store is complete if the load

is of independent data? However, I can imagine this being useful in situations where the same physical memory is mapped using different virtual address ranges - not something that happens often, but it could happen in the kernel.

As a footnote, the equivalent x86 instruction is the `mfence`. There's a good discussion of memory ordering in section 7.2 of the Intel Systems Programming Guide (`http://download.intel.com/design/processor/manuals/253668.pdf`).

There's some more discussion of this topic on Dave Dice's blog (`http://blogs.sun.com/dave/entry/java_memory_model_concerns_on`).

`http://blogs.sun.com/d/entry/when_to_use_membars`

Flush Register Windows

Darryl Gove, March 7, 2008

The SPARC architecture has an interesting feature called Register Windows. The idea is that the processor should contain multiple sets of registers on chip. When a new routine is called, the processor can give a fresh set of registers to the new routine, preserving the value of the old registers. When the new routine completes and control returns to the calling routine, the register values for the old routine are also restored. The idea is for the chip not to have to save and load the values held in registers whenever a routine is called. This reduces memory traffic and should improve performance.

The trouble with register windows, is that each chip can only hold a finite number of them. Once all the register windows are full, the processor has to spill a complete set of registers to memory. This is in contrast with the situation where the program is responsible for spilling and filling registers - the program only need spill a single register if that is all that the routine requires.

Most SPARC processors have about seven sets of register windows, so if the program remains in a call stack depth of about seven, there is no register spill/fill cost associated with calls of other routines. Beyond this stack depth, there is a cost for the spills and fills of the register windows.

The SPARC architecture book (`http://opensparc-t2.sunsource.net/specs/UA2007-current-draft-HP-EXT.pdf`) contains a more detailed description of register windows in section 5.2.2.

Most of the time software is completely unaware of this architectural decision. In fact user code should never have to be aware of it. There are two situations where software does need to know about register windows. These really only impact virtual machines or operating systems:

- **Context switches.** In a context switch, the processor changes to executing another software thread, so all the state from that thread needs to be saved for the thread to later resume execution. Note that `setjmp` and `longjmp` which are sometimes used as part of code to implement context switching already have the appropriate flushes in them.

- **Garbage collection.** Garbage collection involves inspecting the state of the objects held in memory and determining whether each object is live or dead. Live objects are identified by having other live objects point to them. So all the registers need to be stored in memory so that they can be inspected to check whether they point to any objects that should be considered live.

The SPARC V9 instruction `flushw` will cause the processor to store all the register windows in a thread to memory. For SPARC V8, the same effect is attained through `trap x03`. Either way, the cost can be quite high since the processor needs to store up to about 7 sets of register windows to memory. Each set is sixteen 8-byte registers, which results in potentially a lot of memory traffic and cycles.

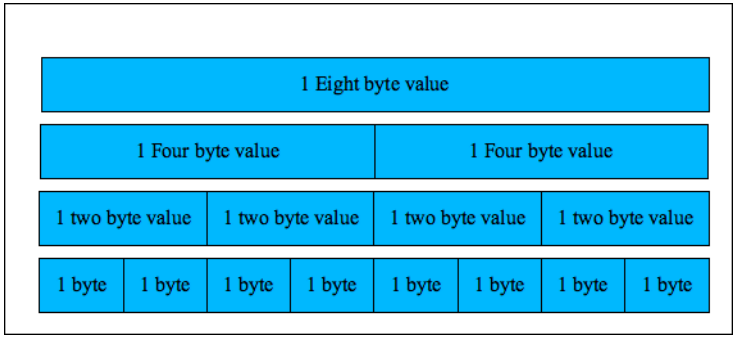
`http://blogs.sun.com/d/entry/flush_register_windows`

Sun Studio: Using VIS Instructions to Speed Up Key Routines

Darryl Gove, Geetha Vallabhaneni, January 5, 2006

Introduction

The VIS instruction set includes a number of instructions that can be used to handle several items of data at the same time. These are called SIMD (Single Instruction Multiple Data) instructions. The VIS instructions work on data held in floating-point registers. The floating-point registers are 8-bytes in size, and the VIS instructions can operate on them as two 4–byte ints, four 2–byte shorts, or eight 1-byte chars, as shown in the following figure.



The advantage of using VIS instructions is that an operation can be applied to different items of data in parallel, meaning that it takes the same time to compute eight 1–byte results as it does to calculate one 8-byte result. In theory this means that code that uses VIS instructions can be many times faster than code without them.

Further information on the VIS instruction set, including manuals and libraries, can be found at `http://www.sun.com/processors/vis/`.

VIS Performance

There can be a performance gain by using VIS instructions. However, determining how much of a performance gain is not straightforward. The following factors come into play:

- The VIS instructions work on multiple data at one time, so doing a VIS add operation on eight chars can be faster than doing eight separate add operations.
- The VIS instructions use values held in floating-point registers, and the values need to be loaded and stored using floating-point loads and stores. This has a benefit on the UltraSPARC-III family of processors since floating-point values can be prefetched into the on-chip prefetch cache. Doing this can avoid processor stalls where the integer data needs to be fetched from memory.
- The VIS instructions typically take longer than the equivalent integer operation. A VIS instruction will often have a four-cycle latency (VIS instructions are handled by the floating-point unit, so the latency for VIS instructions is the same as for floating-point instructions), whereas the equivalent integer operation might take one cycle.
- Unfortunately not all integer operations are available through VIS instructions, so it may be necessary to move data back to the integer registers for processing. This can only be done through memory, so it is quite a slow operation.

Compiling With VIS

Using VIS requires a target architecture of at least `v8plusa` or `v9a`. This can be achieved by compiling using the `-xarch=v8plusa` or `-xarch=v9a` compiler flag.

There are two ways to get the compiler to generate VIS instructions:

- There are macros for the VIS instructions available in `<vis.h>`. The compiler will recognize these if the `-xvis` compiler flag is specified.
- It is also possible to put VIS instructions into inline templates. Details on writing and using inline templates can be found in "Using Inline Templates to Improve Application Performance" on page 74.

Because VIS instructions are not directly generated by the compiler, it may happen that the generated code is suboptimal. (The VIS instructions will typically be late-inlined as discussed in the article on inline templates.) Therefore, if the performance is not as fast as expected, it is always worth checking the resulting assembly code to see if it looks reasonable.

Example Routine Coded Without VIS Instructions

The example we will use is a simple search-type routine that works on integer (4-byte) data rather than characters. The routine scans an array for a particular value, and then reports the number of integers scanned.

```
int search(int value, int* array)
{
  int len=0;
  while (*array++!=value) {len++;}
  return len;
}
```

We also define a test harness so that the performance of the existing code can be measured.

```
#include <stdio.h>
#include <sys/time.h>

int array[10*1024*1024];
static hrtime_t last_time=0;

void seconds()
{
  hrtime_t time;
  time=gethrtime();
  if (last_time!=0)
  {
    printf( "Elapsed seconds = %5.3f\n",
      ((double)(time-last_time)*1.0e-9) );
  }
  last_time=time;
}

void init()
{
  int i;
  for(i=0; i<10*1024*1024; i++) {array[i]=i;}
}

void main()
{
  int loop;
  init();
  seconds();
  for (loop=0; loop <10*1024*1023; loop+=100*1024)
  {
    if (search(loop,array)!=loop)
    {
      printf("Miscompare\n");
      break;
    }
  }
  seconds();
}
```

The timing loop is more complex than might appear necessary. However, the loop has the following characteristics:

- The timing loop also validates the results. This ensures that when the code is optimized, the new versions can also be checked for correctness.
- The timing code tries out a variety of array lengths to ensure that the optimization has to give good performance over a wide range of lengths. Of course, the method of timing has the disadvantage that poor performance at one length can be masked by good performance at another length.

There are some weaknesses in the test harness:

- The code does not attempt to warm the caches into the same state, so it is possible that the earlier test codes might be penalized because the caches are cold.
- The test code does not check different alignments. Performance might be different depending on whether the array starts on a particular 4–byte offset in the cacheline.

Building and Running the Example Code

First we'll run the example code.

```
% cc test.c
% a.out

Elapsed seconds = 14.481
```

The code was compiled without optimization, so performance will be poor. It is interesting to look at the hot loop in this light.

```
10cb4: inc      %i5
10cb8: ld       [%fp + 72], %o0
10cbc: ld       [%o0], %i4
10cc0: ld       [%fp + 72], %o0
10cc4: inc      4, %o0
10cc8: st       %o0, [%fp + 72]
10ccc: ld       [%fp + 68], %o0
10cd0: cmp      %i4, %o0
10cd4: bne      search+0x34         ! 0x10cb4
10cd8: st       %i4, [%fp - 24]
```

It is readily apparent that this is very poor code. The loop index variable (held in `%i4`) and the array offset variable (`%o0`) are stored and reloaded every iteration. The way to evaluate this is to count the number of loads and stores. There are four loads and two stores per trip around the loop. Comparing this with the source code, it seems that there should only be a single load per iteration. The obvious way of improving the situation is to recompile with some amount of optimization.

```
% cc -O test.c
% a.out

Elapsed seconds = 8.614
```

So the performance improves by nearly a factor of two. The disassembly for the hot loop looks like the following:

```
10ca4:  ld         [%g5], %o4
10ca8:  cmp        %o4, %o5
10cac:  be,pn      %icc,search+0x44        ! 0x10ccc
10cb0:  inc        4, %g5
10cb4:  ld         [%g5], %o2
10cb8:  inc        %o0
10cbc:  inc        4, %g5
10cc0:  cmp        %o2, %o5
10cc4:  bne,a,pt   %icc,search+0x1c        ! 0x10ca4
10cc8:  inc        %o0
```

In this case the loop has been unrolled twice (there are two iterations performed before the predicted taken branch at the end of the loop) but not pipelined (the two iterations are not interleaved together). The most significant gain is that the index variable is now held in a register and does not end up being stored and reloaded every iteration. In this case there are two loads in the block of code, but the block of code is for two iterations, so the code has the optimal number of loads.

However, the loop still does not contain prefetch instructions. Adding prefetch instructions will enable the processor to start fetching data in advance of the data being needed. This will mean that the data will often be ready when the processor needs it, and hence the processor will spend less time waiting for the data to be returned from memory.

Manually Adding Prefetch

Using the header file `<sun_prefetch.h>`, it is possible to manually add prefetch statements into the source code for the program.

```
#include <sun_prefetch.h>

....

int prefetch_search(int value, int* array)
{
  int len=0;
  while (*array++!=value)
  {
    len++;
    sparc_prefetch_read_many(array+16);
  }
  return len;
}
```

Running this code with the manual prefetch statements gets the following results.

```
% cc -O test.c
% a.out

Elapsed seconds = 5.606
```

The prefetch statement says to prefetch for 16 ints from the current location within array. This is 16*4 bytes = 64 bytes (each int takes four bytes of memory), or one cacheline ahead. Prefetch can be made more effective by having more time for the prefetch to complete before the load is issued. To demonstrate this, the offset can be changed from +16 to +64, which means to prefetch for 64*4 bytes ahead, or four cachelines. The following result is obtained.

```
% cc -O test.c
% a.out

Elapsed seconds = 3.566
```

So from using optimization and manually inserting prefetch it is possible to get a nearly five–fold gain in performance for this bit of code.

Including VIS Instructions in the Source Code

One way of using VIS instructions is to include them in the source code of the application. This requires the use of the `<vis.h>` header file and the flag `-xvis` in order that the compiler can recognize them. VIS instructions also require an architecture of at least `v8plusa`.

The code can be modified to use VIS instructions. Since the comparison is of four-byte integers, two can be loaded and then compared with the target value at the same time. The macro `vis_fcmpeq32` generates the VIS instruction which performs the compare. The following code performs the search using VIS instructions:

```
#include <vis.h>

...

int srcvis_search(int value, int *array) {
  union {double d; int i1,i2;} tmpR;
  double* tmpI;
  int ret;
  int ind = 0;
  tmpR.i1=value;
  tmpR.i2=value;
  if(!((unsigned long)(&array[0]) & 4)
      || array[ind++] != value)
  {
    tmpI = (double*) &array[ind];
    for(; !(ret=vis_fcmpeq32(tmpR.d, *tmpI++)); ind+=2)
    {
       sparc_prefetch_read_many(tmpI+32);
    }
    if (!(ret &2)) { ind++;}
  }
  return ind;
}
```

The compile line for this code is:

```
% cc -O -xvis -xarch=v8plusa test.c
% a.out

Elapsed seconds = 2.639
```

The VIS code is faster than the previous integer code. There are two main reasons for this. There is some performance gain from being able to compare two integer values at once, but the instructions to do so are longer latency, so there is not much to be gained from this. (Of course, if the code was working with eight bytes, or four shorts, then the VIS instructions would lead to greater performance gains.) The other contributor to performance is that the floating-point load instructions can load data from the on-chip prefetch cache, which reduces the time spent waiting for data from the off-chip caches.

It is worth discussing the code, which looks significantly more complex than the previous versions of the same code. The reasons for the complexity are as follows:

- Since VIS works using the floating-point registers, the value that is being searched for needs to be duplicated into both the upper and lower halves of the floating-point register. This is achieved using the union statement. This value will be used to compare with the value loaded from memory.
- The loads are loading eight byte values, so the loads need to be eight-byte aligned. To get the correct alignment, the IF statement checks for misalignment. If it is misaligned, it handles the first value in the array before passing control on to the FOR loop.
- The return value from the VIS compare instruction is a bit pattern which indicates which of the two values being compared was different. Bit 1 indicates that the upper value was different. Bit 0 indicates that the lower value was different. If the upper value was not different, then it is necessary to add one to the return value to show that the miscompare took place with the lower value. (Note that the lower bit being set does not necessarily mean that the miscompare took place with the lower value. The upper value could also have miscompared.)

Using VIS and Inline Templates

In order to obtain the best possible performance from VIS, it is necessary to use inline templates to schedule the code. The following code is basically doing the same algorithm as the C source code, but the code layout is slightly tweaked to improve performance.

```
/* Routine vis_search(int value, unsigned int * array); */
/* %o0 = value*/
/* %o1 = address of array*/

.inline vis_search,8
  st %o0,[%sp+0x48]    /* store search value */
  clr %o3              /* counter = 0 */
  and %o1,4,%o2        /* check for not 8-byte aligned*/
  cmp %o2,4
  bne 1f
```

```
  ld  [%o1],%o2        /* load misaligned int */

  cmp %o2,%o0
  be 2f                /* found */
  add %o1,4,%o1        /* dealt with misaligned int */

  add %o3,1,%o3        /* compared first character */
1:
  ld [%sp+0x48],%f0    /* load upper word */
  fmovs %f0,%f1        /* copy to lower word */
  ldd [%o1],%f2        /* load 2 ints */

3:
  add %o1,8,%o1        /* move pointer to next pair */
  prefetch [%o1+256],0 /* Prefetch four lines ahead */
  fcmpne32 %f0,%f2,%o0 /* compare ints */
  ldda [%o1]%asi,%f2   /* non-faulting load 2 ints */
  cmp %o0,3            /* check result of compare */
  be,a 3b              /* branch if not found */
  add %o3,2,%o3        /* increment count by two; annulled if found */

  andcc %o0,2,%o0      /* check bit 2 of return value from compare */
  bnz,a 2f
  add %o3,1,%o3        /* add one if the first half matched */

2:
  or %o3,%g0,%o0       /* Copy counter to output */
.end
```

The VIS code starts by checking and correcting for non-eight-byte aligned arrays. It then duplicates the integer value into both halves of the floating-point value. The inner loop is then very similar to the VIS example coded in C. The example below shows compiling and running this code There is a slight performance gain over the VIS instructions used at source code level. The gain can be attributed to using non-faulting (speculative) load instructions to fetch the next value before the compare of the current value has completed.

The non-faulting load instruction is equivalent to a normal load, except in the case where the load would access unmapped memory (for example, off the end of an array). In this circumstance a normal load would cause a runtime error because the memory is not mapped. However, in the same situation, a non-faulting load would not cause an error and would just return a zero value. The advantage of using this instruction is that the load can now be moved before the test of whether the end of the array has been reached, safe in the knowledge that no fault will occur if the load does happen to pass the end of the array.

```
% cc -O vis_search.il test.c
% a.out

Elapsed seconds = 2.416
```

It is possible to further improve the performance of the hand-coded VIS routine. For example, it would be possible to unroll and pipeline the inner loop such that two or more iterations were computed in parallel. Whilst doing that optimization would improve performance, it would also make the code less clear, so it was not shown here.

Concluding Remarks

This article has demonstrated a number of useful techniques for improving performance. To recap:

- Whilst the compiler generally does a good job of inserting prefetch into code, there is always the possibility of manually inserting prefetch statements into the code to cover the cases where the compiler does not.
- It is possible to write source code that includes VIS instructions in a way that leads to performance gains.
- If necessary, further performance can be extracted from codes by manually recoding the hot points using inline templates.

Full Source Code

Here is a link to the full source code test.c (`http://developers.sun.com/solaris/articles/src/test.c`) and to the vis_search.il (`http://developers.sun.com/solaris/articles/src/vis_search.il`) inline code.

Compiling and running the program should produce results similar to that shown below.

```
% cc -xvis -O -xarch=v8plusa test.c vis_search.il

% a.out

Elapsed seconds = 8.628
Elapsed seconds = 5.606
Elapsed seconds = 2.645
Elapsed seconds = 2.410
```

`http://developers.sun.com/solaris/articles/vis.html`

Using the UltraSPARC Hardware Cryptographic Accelerators

Lawrence Spracklen, May 29, 2008

Summary

A brief synopsis of how to leverage the UltraSPARC hardware cryptographic accelerators from your application.

Introduction

Sun's UltraSPARC T1, T2 and T2 Plus processors support high-performance hardware cryptographic accelerators on chip. These accelerators can significantly reduce the normally significant overheads associated with cryptography and secure operation.

On the UltraSPARC T1, T2 and T2 Plus processors, there is a cryptographic accelerator per each core, such that an 8-core processor provides 8 accelerators. The algorithms supported by these accelerators vary with processor and are illustrated in the following table:

TABLE 3–5 Algorithms Supported

Algorithm	UltraSPARC T1	UltraSPARC T2/ UltraSPARC T2 Plus
Public-key algorithms		
RSA, DSA, DH	•	•
ECC, ECDSA, ECDHA		•
Symmetric algorithms		
RC4		•
DES, 3DES		•
AES-{128,192,256}		•
Cryptographic hashes		
MD5		•
SHA-1		•
SHA-224/256		•

The public-key operations are performed by the accelerator's modular arithmetic unit, while symmetric cipher and cryptographic hash operations are performed by the accelerator's cipher and hash unit (CHU). The UltraSPARC T1 accelerators are composed of just a Modular Arithmetic Unit (MAU), while the UltraSPARC T2/T2 Plus accelerators have both MAU and CHU, both of which can operate in parallel. The accelerators operate at the core frequency (in parallel with the core) and are capable of delivering cryptographic performance that is typically an order of magnitude better than can be achieved on traditional processors in software, as is illustrated in the following table.

TABLE 3–6 Cryptographic Performance

Algorithm	UltraSPARC T1	UltraSPARC T2/ UltraSPARC T2 Plus
RSA-1024	20,000 sign operations/sec/chip (8-core)	37,000 sign operations/sec/chip (8-core)
AES-128-CBC		44Gb/s/chip (8-core)
SHA-1		32Gb/s/chip (8-core)

This article describes how to code your application such that it can leverage these hardware accelerators. Many important applications will already leverage the UltraSPARC hardware accelerators, either directly out of the box or with minimal configuration. These include the Sun Studio webserver, the Apache webserver, KSSL and IPsec to name but a few. More details of how to configure these applications are provided in a Sun cryptographic blueprint [1].

Using the UltraSPARC Hardware Cryptographic Accelerators

Access to the cryptographic accelerators is controlled by the Solaris Cryptographic Framework. For non-privileged applications, access is via the userland cryptographic framework (UCF), while for kernel modules (such as KSSL or IPsec) access is via the kernel cryptographic framework (KCF). This article focuses on the userland cryptographic framework.

The userland cryptographic framework exposes a PKCS#11 [2] compliant API to non-priv userland applications. Applications can interact directly with the UCF via the PKCS#11 interface, or indirectly via:

- Java Cryptography Extension (JCE)
- OpenSSL
- Network Security Services (NSS)

The remainder of this article focuses on how to interact with the UCF directly and indirectly via JCE, OpenSSL and NSS.

Direct Interaction With UCF

For PKCS#11 compliant applications, `libpkcs11.so` is the gateway to the UCF, and it's just a simple matter of linking against this library (located in `/usr/lib`). Given the fairly widespread use of the PKCS11 interface, especially with respect to traditional off-chip cryptographic accelerators (such as the Sun SCA6000 card), many applications already leverage PKCS#11. If an application doesn't already use the PKCS#11 interface, it is pretty straightforward to modify the application, with documents showing example implementations readily available [3].

Offload via OpenSSL

If the application uses OpenSSL for its cryptographic requirements (and many do), access to the accelerators can be achieved by using a version of OpenSSL that has been modified to support the PKCS#11 engine. A patched version of OpenSSL is supplied with Solaris 10 and is located in `/usr/sfw/lib`, allowing application compilation as follows:

```
cc -fast -I /usr/sfw/include -L /usr/sfw/lib -lcrypto aes_test.c -o aes_test.out
```

For operations that are to be offloaded, it is necessary to restrict use to the EVP_ functions and explicitly indicate the use of the PKCS11 engine. Something like the following works for bulk ciphers (the process for RSA is similar):

```
ENGINE *e;
ENGINE_LOAD_builtin_engines();
e = ENGINE_by_id("pkcs11");
ENGINE_set_default_Ciphers(e);
EVP_CIPHER_CTX_init(&ctx);
EVP_EncryptInit(&ctx, EVP_des_cbc (), key, iv);
EVP_EncryptUpdate(.....);
```

PKCS#11 engine patches are available from OpenSSL.org for a number of different versions of OpenSSL, if the version of OpenSSL that ships with the Solaris OS isn't suitable [4].

Offload via JCE

For applications that utilize the Java Cryptography Extension (JCE), the application should simply be configured to utilize the `SunPKCS11-Solaris` provider. Accordingly, in order for applications to use the hardware accelerators automatically, it is just necessary to ensure that `sun.security.pkcs11.SunPKCS11` is configured as the first provider in the `$JAVA_HOME/jre/lib/security/java.security` file.

The `SunPKCS11-Solaris` provider can also be explicitly selected as follows:

```
String provider = "SunPKCS11-Solaris";
Cipher aescipher = Cipher.getInstance("AES/ECB/NoPadding", provider);
```

It should be noted that the `SunPKCS11-Solaris` provider currently only offloads a subset of the chaining modes supported by the hardware, so make sure that the chaining mode and padding mode are supported [5]. The modes supported by the hardware accelerators are as follows:

TABLE 3–7 Supported Chaining Modes

Cipher	Supported Chaining Modes
AES	ECB, CBC, CTR
DES/3DES	ECB, CBC,CFB64

Offloading via NSS

In order for NSS to use the hardware cryptographic accelerators, the Solaris Cryptographic Framework should be added as a provider for NSS. This is achieved by modifying the appropriate NSS security databases. As an example, the following illustrates how Firefox can offload RSA operations to the hardware:

```
/usr/sfw/bin/modutil -dbdir /home/sprack/.mozilla/firefox/r5s548iw.default/ \
  -add "Solaris Crypto Framework" -libfile /usr/lib/libpkcs11.so -mechanisms RSA
/usr/sfw/bin/modutil -dbdir /home/sprack/.mozilla/firefox/r5s548iw.default/ \
  -enable "Solaris Crypto Framework"
```

The use of the mechanism option indicates that the Solaris Cryptographic Framework should be the default provider for RSA operations [6].

Observability

When operations are submitted to the cryptographic framework, the cryptographic framework will, as appropriate, route processing for these operations to the Niagara cryptographic provider `ncp`(7D) device driver for public-key operations, and the Niagara-2 cryptographic provider `n2cp`(7D) device driver for symmetric cipher and cryptographic hash operations. These device drivers then perform the actual offload to the hardware accelerators and return the results to the framework. The interaction between these drivers and the cryptographic frame is controlled via `cryptoadm`.

`kstat` can be used to provide insight into the cryptographic operations that `ncp` and `n2cp` are handling, as follows:

```
kstat -m ncp | less
kstat -m n2cp | less
```

`cputrack` can be utilized to determine the activity of the hardware accelerators directly. (Use `cputrack -h` to determine which counters to track.)

Concluding Thoughts

Cryptographic processing overheads are finding their way into an ever wider array of applications as security becomes ever more important. By providing on-chip hardware cryptographic accelerators, the UltraSPARC processors can vastly reduce these overheads, and in many situations enable respectable performance even when operating securely.

The Solaris Cryptographic Framework provides a simple way by which applications can leverage the benefits of the UltraSPARC hardware accelerators, while continuing to ensure application portability

References

[1] Using the cryptographic accelerators in the UltraSPARC T1 and T2 processors (http://www.sun.com/blueprints/0306/819-5782.pdf)

[2] PKCS #11: Cryptographic Token Interface Standard (ftp://ftp.rsasecurity.com/pub/pkcs/pkcs-11/v2-20/pkcs-11v2-20.pdf)

[3] Solaris Cryptographic Framework (http://learningsolaris.com/docs/crypt_framework_official_march_2005.pdf)

[4] Miscellaneous OpenSSL Contributions (http://www.openssl.org/contrib/)

[5] Sun PKCS#11 Provider's Supported Algorithms (http://java.sun.com/javase/6/docs/technotes/guides/security/p11guide.html#ALG)

[6] Configuring Solaris Cryptographic Framework and Sun Java System Web Server 7 on Systems With UltraSPARC T1 Processors (http://www.sun.com/bigadmin/features/articles/web_server_t1.html)

http://blogs.sun.com/sprack/entry/using_the_ultrasparc_hardware_cryptographic

Atomic SPARC: Using the SPARC Atomic Instructions

Richard Marejka, March 2008

Summary

The SPARC architecture is a RISC processor that originally appeared in systems from Sun Microsystems in 1986 as the Sun-4/260 (http://www.operating-system.org/betriebssystem/_english/w-plattform.htm). Since then, the processor has undergone many refinements to meet the changing needs of its customers.

Today's SPARC processors are designed for both high performance and energy efficiency. This is achieved by incorporating multiple cores on a processor die and multiple processors within a system. These systems yield excellent results when applied to multiprocess and multithreaded jobs. The degree of parallelism and efficiency could not be achieved without atomic instructions. These instructions provide the basis for the synchronization required to achieve the high degree of parallelism associated with the Solaris Operating System (Solaris OS) and SPARC.

This article briefly introduces the SPARC memory model and atomic instructions, then implements a number of IBM AIX interfaces for use on the Solaris OS. The article assumes that the reader is familiar with assembler language programming. This collection of examples serves

to demonstrate the use of the SPARC atomic instructions and memory models, and it provides a small library that can be useful for the programmer who wants to port IBM AIX-based source code to the Solaris OS / SPARC platform.

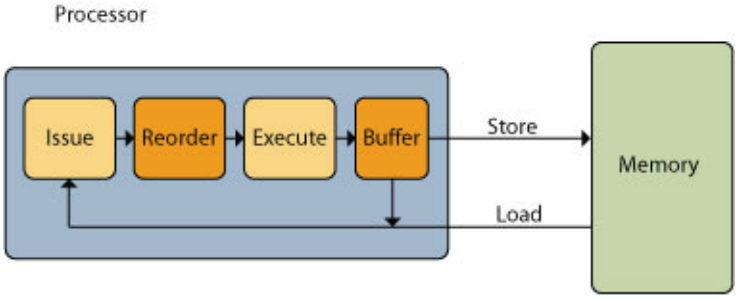
Memory Model

The SPARC Version 9 (SPARC v9) specification defines three memory models, from least restrictive to most restrictive:

- **Relaxed Memory Order (RMO).** There are no ordering constraints on memory references outside of those required for processor self-consistency. Where ordering is required, the developer must explicitly design and code for it by using the `membar` instruction.
- **Partial Store Order (PSO).** This has all of the requirements of RMO, plus loads are ordered with respect to earlier loads, and atomic load-stores are ordered with respect to loads. A `membar` instruction is required to order stores and atomic load-stores with respect to each other.
- **Total Store Order (TSO).** This has all the requirements of PSO, plus stores are ordered with respect to earlier stores, and atomic load-stores are ordered with respect to loads and stores.

The SPARC architecture provides multiple memory models for two reasons: so that implementations can schedule memory operations for higher performance, and so that programmers can create synchronization primitives using shared memory. The less-restrictive memory models, RMO and PSO, afford more opportunities for application performance improvements by the processor.

An idealized SPARC v9 processor has the form shown in the following figure.



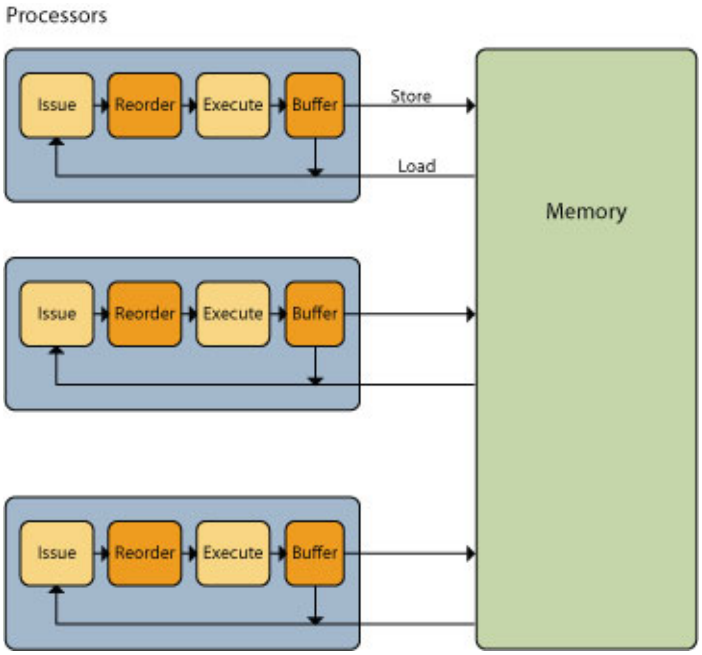
The processor's issue unit reads instructions from memory and issues them in program order. *Program order* is the order determined by the control flow of the application under the assumption that each instruction executes independently and sequentially.

The reorder unit gathers those issued instructions for dispatch to the execute unit. The reorder unit allows an implementation to rearrange instructions to perform some instructions in parallel for greater efficiency. The reordering is constrained to maintain program order.

The execute unit carries out the instruction and writes results to a buffer unit.

The buffer unit schedules write operations to memory. The presence of the buffer unit frees the execute unit from delays incurred by writing to memory. The buffer unit can also respond to load requests for a memory address when it holds the contents of the address from a previous store. This introduces a potential for inconsistency -- a write request to memory can be present in the buffer unit, reloaded by the issue unit from the buffer unit, modified by the execute unit again and the write request enqueued yet again before a write to memory by the buffer unit. Although this can in theory produce an inconsistency between processes in a single-processor system, this inconsistency does not in fact occur, because the actions of a process-context switch include a buffer unit flush to memory.

In a multiprocessor system, an inconsistency can occur as a result of the presence of the buffer unit. The inconsistency will arise when memory shared between processes is modified by their respective processors but unwritten to memory -- that is, when the modified values are present in the buffer unit of more than one processor. The following figure illustrates a multiprocessor system.



`membar` Instruction

The SPARC v9 architecture includes the `membar` instruction. The term *membar* is a contraction of "memory barrier." The `membar` instruction has two variants:

- *Ordering* gives the programmer control over the order of loads and stores issued by a processor.
- *Sequencing* gives the programmer control over the order and completion of memory operations.

The ordering `membar` instruction imposes an order on the instruction stream of one processor. Loads and stores that appear before the `membar` instruction are ordered with respect to those appearing after the `membar`. The atomic instructions are ordered as if they are both a load and a store, because they perform both operations. The instruction contains four bit-encoded ordering relationships:

- `#LoadLoad`
- `#StoreLoad`
- `#LoadStore`
- `#StoreStore`

The semantics of each `#XY` relationship are as follows: "All X operations that appear before the `membar` in program order complete before any Y operations that follow after the `membar` in program order." More complex ordering requirements can be created by combining relationships using the bit-wise `or` operator.

The SPARC Version 8 (SPARC v8) `stbar` instruction is a subset of `membar`, equivalent to a `membar` with an ordering relationship of `#StoreStore`.

Depending on the memory model under which the system is operating, the programmer must explicitly insert memory-ordering instructions to guarantee program correctness. For example, the SPARC assembler code to release a lock using the store unsigned byte (`stub`) instruction is as follows:

```
unlock_ldstub:
    nop                                 ! TSO or
    membar    #StoreStore                 ! PSO or
    membar    #StoreStore | #LoadStore    ! RMO
    stub    %g0,[%o0]
    retl
    nop
```

Remember that a lock protects some variable that will be modified after the lock is acquired and before the lock is released. The modification of the variable implies a store to memory by the program. In PSO mode, the `#StoreStore` will cause the store to the variable to be ordered with respect to the store to the lock. If the order was not imposed, the lock could appear to be free before the variable had been updated in memory.

Those who follow the defensive school of programming will develop code assuming the least restrictive model, RMO, because this will execute correctly when using the other two SPARC memory models. The defensive school would also gather all synchronization primitives into a common system library.

Atomic Instructions

SPARC machine instructions are normally executed to completion without interruption. This includes the memory access instructions of load and store. In multiprocessor-multicore systems, two or more processes executing an instruction using the same memory address are guaranteed to occur in a serial but undefined order. This guarantees memory consistency, but the order of operations is undefined, as are the memory contents after the operations complete.

Atomic instructions act like both a load and a store, extending the "without interruption" requirement to include both operations. These instructions allow the creation of multithreaded and cooperating multiprocess applications that take advantage of the concurrency offered by today's high-performance systems.

SPARC Version 9 (v9) has three atomic instructions:

- `ldstub`
- `swap`
- `cas`

Load-Store Unsigned Byte: `ldstub`

The load-store unsigned byte (`ldstub`) instruction was the original atomic primitive used in the Solaris OS to implement mutual exclusion locks (`mutex`) and other thread synchronization primitives. The instruction writes `0xff` into a byte and returns the old value in a register. To acquire a lock, a caller used `ldstub` and then inspected the value returned. If the returned value was zero, the lock had then been acquired and the lock byte now contained `0xff`. If the return value was `0xff`, the lock was held by a previous caller. The caller would then execute some adaptive algorithm to wait until the lock became available, that is, its value became or returned to zero.

Here is the algorithm for the instruction, presented in a C-like pseudocode:

```
int8_t
ldstub( int8_t *lock_byte ) {
    int8_t   old_value;

    atomic {
        old_value  = *lock_byte;
        *lock_byte = 0xff;
    }
```

```
    return( old_value );
}
```

The `ldstub` is the classic test-and-set instruction. The shortcoming of the instruction is that it has a consensus number (`http://www.cs.tau.ac.il/%7Eshanir/multiprocessor-synch-2003/universal/notes/universal.pdf`) of two and hence cannot resolve more than two contending processes in a wait-free fashion.

Swap Register With Memory: `swap`

The swap register with memory (`swap`) instruction exchanges the contents of a word in memory with a register. According to the SPARC v9 manual, the `swap` instruction is deprecated, and programmers should use the `casxa` instruction in its place.

The algorithm is as follows:

```
int32_t
swap( int32_t *word, int32_t new_value ) {
    int32_t   old_value;

    atomic {
        old_value = *word;
        *word     = new_value;
    }

    return( old_value );
}
```

Similar to the `ldstub` instruction, `swap` has a consensus number of two.

Compare and Swap: `cas`

The SPARC v9 manual introduced the newest atomic instruction: compare and swap (`cas`) at section 8.4.6.1. The instruction uses a memory address and two registers. It compares a word in memory with a register. If these are equal, the instruction swaps the contents of the word in memory with a second register.

The instruction has an infinite consensus number: It can resolve an infinite number of contending processes in a wait-free fashion. You can use `cas` for so-called lock-free operations (linked-list management is the classic example). The term *lock-free* is somewhat of a misnomer. The process still uses locking, but the locking takes place in hardware instead of the more commonly coded mutual-exclusion lock.

Here is the algorithm for `cas` :

```
int64_t
cas64( int64_t *word, int64_t test_value, int64_t new_value ) {
    int64_t   old_value;
```

```
    atomic {
        old_value    = *word;

        if ( *word == test_value )
            *word= new_value;
    }

    return( old_value );
}
```

To determine whether a swap took place, compare the return value in the second register with the test value used in the first register. Here is what this looks like in this article's pseudocode:

```
if ( cas64( &cas_word, testV, newV ) == testV )
    /* swap took place */
```

You can use a simple performance enhancement when you write this in SPARC assembler.

Solaris OS Interfaces

The Solaris OS has many interfaces for use in concurrent and multiprocessor systems. These Solaris thread interfaces (`http://docs.sun.com/app/docs/doc/801-6659`) date from at least Solaris 2.4 (March 1993). The POSIX thread interfaces (`http://docs.sun.com/app/docs/doc/802-1949/802-1949?a=browse`) were added in Solaris 2.5 (June 1995). These are often referred to as the Pthread interfaces and originally appeared in the 3T reference manual section. In Solaris 10, both Solaris and POSIX thread interfaces are documented in the *Multithreaded Programming Guide* (`http://docs.sun.com/app/docs/doc/816-5137`).

The Pthread interfaces have been extended over the years to remain POSIX compliant and to add functionality that is not part of the POSIX standard. The thread library has been rewritten at least once. This rewrite harmonized the Solaris OS and POSIX interfaces, integrated the interfaces into `libc`, and improved performance.

Most vendors have adapted the POSIX thread interfaces, so porting applications between vendor platforms is largely a matter of recompiling for any thread interfaces. However, a simple recompile will not solve one class of problems. If the application used synchronization primitives that were vendor specific, then porting the application requires more than a recompile. In some cases, this may be trivial. For example, a simple interface mapping can be made from Solaris OS to POSIX thread interfaces with almost perfect fidelity. But once again, some interfaces will not be trivial to implement.

IBM AIX Interfaces

The IBM AIX (`http://www.ibm.com/aix`) platform has offered a few vendor-specific synchronization primitives since version 3.2 was released in 1992. The paper "Turning the AIX Operating System Into an MP-Capable OS" by Jacques Talbot provides a good background on

the PowerPC processor and synchronization under the AIX operating system. The paper is available from the USENIX 1995 Technical Conference Proceedings (`http://www.sagecertification.org/publications/library/proceedings/neworl/`) under the Potpourri II section.

The synchronization primitives in question are the following:

```
#include   <sys/atomic_op.h>

int         fetch_and_add( atomic_p word_addr, int value );
uint        fetch_and_or( atomic_p word_addr, int mask );
uint        fetch_and_and( atomic_p word_addr, int mask );
void        _clear_lock( atomic_p word_addr, int value );
boolean_t   _check_lock( atomic_p word_addr, int old_val, int new_val );
boolean_t   compare_and_swap( atomic_p word_addr, int *old_val_addr, int new_val );
boolean_t   test_and_set( atomic_p word_addr, int mask );
```

Knowing the SPARC atomic instructions and a little SPARC assembler, you can implement the seven AIX synchronization primitives for the Solaris OS. This should result in a compatibility library for porting AIX applications to the Solaris platform. The library will be implemented for Solaris / SPARC v9 platforms running in 32-bit mode. The programmer community will have to provide SPARC 64-bit, AMD, and Intel implementations.

Each of the interfaces in this section will be presented in a C-like pseudocode and implemented in SPARC assembler. All of the interfaces are leaf routines and as such can take advantage of the leaf procedure optimizations as described in sections D.5 and H.1.2 of the SPARC Architecture Manual v8 (`http://www.sparc.org/standards/V8.pdf`) (PDF).

The `fetch_and_add`, `fetch_and_or`, and `fetch_and_and` Interfaces

The `fetch_and_OP` interfaces all follow the same algorithm, substituting the specific operation where required. The SPARC platform does not operate directly on memory. All operations are register-based with the exception of load, store, and the atomic instructions. This requires the `fetch_and_OP` interfaces to employ a `cas` instruction within a loop. Once the expected result is computed within the loop, the `cas` instruction is used to attempt the update. If the update fails, the loop continues.

Here is the algorithm:

```
int
fetch_and_OP( atomic_p word_addr, int value ) {
    int   result;

    atomic {
        result       = *word_addr;
        *word_addr OP= value;
```

```
    }
    return( result );
}
```

In SPARC assembler, this becomes the following:

```
    fetch_and_OP:
          ld        [%o0],%g1         ! load the current value
    loop: OP        %g1,%o1,%o2       ! compute the desired result
          cas       [%o0],%g1,%o2     ! try to CAS it into place
          cmp       %g1,%o2
          bne,a,pn  %icc,loop         ! CAS failed, try again
          mov       %o2,%g1           ! save current value for next iteration
          retl
          mov       %o2,%o0           ! return the old value
```

All that remains to complete the three interfaces is the substitution of `and`, `or`, or `add` for the `OP` placeholder instruction.

Note that two conditional branch features are used: annul and predictive. The annul "`,a`" will skip the `mov` instruction in the delay slot if the branch is not taken. The predictive "`,pn`" is a hint to the processor that the conditional branch will likely not be taken, that is, the processor should assume low contention.

Also, a performance enhancement is made possible by the return value of the `cas` instruction. Rather than employing a load to read the word from memory, the value returned by the `cas` instruction by way of the second register is used.

The `_clear_lock` and `_check_lock` Interfaces

These two interfaces update a lock word in an atomic manner. The `_clear_lock` interface simply sets the value of the lock to a given value -- essentially a runtime initialization. The `_check_lock` interface conditionally updates the lock word with a new value. The interface `_check_lock` is a compare-and-swap type of operation.

The algorithms are as follows:

```
void
_clear_lock( atomic_p word_addr, int value ) {
    *word_addr    = value;
    return;
}

boolean_t
_check_lock( atomic_p *word_addr, int old_val, int new_val ) {
    int    result;

    atomic {
        if ( *word_addr == old_val )
            *word_addr  = new_val;
            result      = FALSE;
```

```
        } else
            result      = TRUE;
    }
    return( result );
}
```

Because each of these interfaces is intended to be used for synchronization, a memory barrier will be required. Here is the SPARC assembler for them:

```
    _clear_lock:
            membar  #StoreStore|#LoadStore  ! memory barrier (RMO)
            st      %o1,[%o0]               ! store the word
            retl
            nop

    _check_lock:
            cas     [%o0],%o1,%o2           ! try the CAS
            cmp     %o1,%o2
            mov     0,%o0                   ! assume it succeeded - return FALSE/0
            movne   %icc,1,%o0              ! may have failed - return TRUE/1
            membar  #LoadLoad|#LoadStore    ! memory barrier (RMO)
            retl
            nop
```

Of note in the `_check_lock` interface is the use of a conditional move instruction, specifically `movne`. The conditional move uses the integer condition codes, by the use of `%icc`, to move 1 to the return register only if the `cas` instruction failed as detected by the `cmp` instruction.

The `compare_and_swap` Interface

The `compare_and_swap` interface directly maps to the SPARC v9 `cas` instruction with two implementation requirements: The return value is a `boolean_t`, and the old value is returned by way of the second argument if the swap did not take place.

```
boolean_t
compare_and_swap( atomic_p word_addr, int *old_val_addr, int new_val ) {
    int         oldV    = *old_val_addr;
    boolean_t   result;

    atomic {
        if ( *word_addr == oldV ) {
            *word_addr    = new_val;
            result        = TRUE;
        } else {
            *old_val_addr = word_addr;
            result        = FALSE;
        }
    }
    return( result );
}
```

The SPARC assembler for the interface is a straightforward mapping to the `cas` instruction with the appropriate argument and return value management.

```
    compare_and_swap:
          ld    [%o1],%g1       ! set the old value
          cas   [%o0],%g1,%o2   ! try the CAS
          cmp   %g1,%o2
          be,a  true
          mov   1,%o0           ! return TRUE/1
          mov   0,%o0           ! return FALSE/0
          st    %o2,[%o1]       ! store existing value in memory
    true: retl
          nop
```

The only item of note is the return handling, which is written using a conditional branch with the annul bit. If the swap took place, the branch will be taken and the `mov 1,%o0` will be executed in the delay slot. If the branch is not taken, the annul bit will cause the first move instruction to be skipped, and execution will continue with the `mov 0,%o0` instruction.

The `test_and_set` Interface

This interface is a bitwise test-and-set operation. A bitwise `and` of the `mask` and contents of the word in memory are the test part of the operation. If no bits in `mask` are set in the word, then `mask` is added to the word using a bit-wise `or` operation. This is the set part of the operation.

```
boolean_t
test_and_set( atomic_p word_addr, int mask ) {
    boolean_t   result;

    atomic {
        if ( *word_addr & mask )
            result   = FALSE;
        else {
            *word_addr  |= mask;
            result       = TRUE;
        }
    }
    return( result );
}
```

The implementation will use a loop, but not because the interface is required to succeed before returning; that is not a requirement. The loop is required because of a race-condition between the `if ( ... )` and the `*word_addr |= mask` steps. To illustrate, here is the pseudocode rewritten without the `atomic` section:

```
boolean_t
test_and_set( atomic_p word_addr, int mask ) {
    boolean_t   result;

    loop {
        int oldV   = *word_addr;

        if ( oldV & mask ) {        /* test step */
            result = FALSE;
            break;
```

```
        } else {
            int   newV = oldV | mask;

            if ( cas32( word_addr, oldV, newV ) == oldV ) {
                result = TRUE;
                break;
            } else         /* *word_addr changed between test step and cas */
                continue;
        }
    }
    return( result );
}
```

The preceding form assumes the existence of a `cas32` function that includes an `atomic` section. This is a safe assumption because the function can be written as follows:

```
    cas32:
        cas    [%o0],%o1,%o2
        retl
        mov    %o2,%o0
```

And the function is atomic by definition.

This second form of `test_and_set` as shown earlier closely maps to SPARC assembler and is as follows:

```
    test_and_set:
           ld     [%o0],%g1       ! load the current value
    loop:  andcc  %g1,%o1,%g0     ! test mask against value
           bnz,a  done
           mov    0,%o0           ! return FALSE/0
           or     %g1,%o1,%o2     ! compute the desired result
           cas    [%o0],%g1,%o2   ! try the CAS
           cmp    %g1,%o2
           be,a   done
           mov    1,%o0           ! return TRUE/1
           ba     loop            ! try again
           mov    %o2,%g1         ! save current value for next iteration
    done:  retl
           nop
```

Conclusion

This article has provided an overview of the SPARC v9 processor memory model and atomic instructions as they pertain to multiprocessor systems and shared memory applications. It also provided a Solaris OS implementation of several IBM AIX interfaces that you can use to aid in porting AIX-based applications to the Solaris OS.

For More Information

- *The SPARC Architecture Manual Version 9* (http://www.amazon.com/SPARC-Architecture-Manual-Version-9/dp/0130992275/ref=sr_1_1?ie=UTF8[amp]s=books[amp]qid=1204746544[amp]sr=1-1)
- *The SPARC Architecture Manual Version 8* (http://www.sparc.org/standards/V8.pdf) (PDF); see section D.5 Leaf Procedure Optimization, pp. 198-200
- *SPARC Assembly Language Reference Manual* (http://dlc.sun.com/pdf/816-1681/816-1681.pdf) (PDF)
- *System V Application Binary Interface, SPARC Processor Supplement*, UNIX Software Operations (http://www.amazon.com/System-Application-Interface-Processor-Supplement/dp/0131046969/ref=sr_1_2?ie=UTF8[amp]s=books[amp]qid=1204746670[amp]sr=1-2) (Englewood Cliffs, N.J.: Prentice Hall, 1990; ISBN 0-13-877630-X)
- IBM Systems Information Center. (http://publib.boulder.ibm.com/infocenter/systems/index.jsp)
- Turning the AIX Operating System Into an MP-Capable OS (http://www.sagecertification.org/publications/library/proceedings/neworl/talbot.html), by Jacques Talbot
- Operating System Documentation Project: Platforms (http://www.operating-system.org/betriebssystem/_english/w-plattform.htm).
- Consensus number: Computability of Fault-Tolerant Distributed Protocols (http://www.di.ens.fr/%7Egoubault/link004.html)
- Consensus number: The Universality of Consensus (http://www.cs.tau.ac.il/%7Eshanir/multiprocessor-synch-2003/universal/notes/universal.pdf) (PDF)
- *Java Concurrency in Practice* (http://www.informit.com/store/product.aspx?isbn=0321349601), by Brian Goetz and others; see section 15.2.1 Compare and Swap (http://safari.oreilly.com/0321349601/ch15lev1sec2)

http://developers.sun.com/solaris/articles/atomic_sparc/

Reserving Temporary Memory By Using `alloca`

Darryl Gove, May 22, 2008

There are occasions where it's useful to allocate a small temporary working area for the duration of a call to a routine - for example to hold an array. One way of doing this is to use `malloc` and `free`:

```
void f(int a)
{
  int* array=(int*)malloc(sizeof(int)*a);
```

```
  ...
  free(array);
}
```

Obviously the use of `malloc` and `free` does incur some overhead, and which is undesirable, particularly if this is a performance critical routine.

An alternative approach is to use `alloca`(3C) which allocates memory on the stack. Being on the stack, the memory is "freed" when the routine exits. Well, the memory isn't allocated, so it's not really freed, but accesses to the memory once the routine exits are not valid - you'll be accessing data either beyond the stack, or in the stack frame of another routine. Neither situation is likely to be good.

The equivalent code is:

```
#include <alloca.h>


void f(int a)
{
  int* array=(int*)alloca(sizeof(int)*a);
  ...
}
```

For routines which do require temporary storage, this can be a much faster way of allocating it.

http://blogs.sun.com/d/entry/reserving_temporary_memory_using_alloca

`alloca` Internals

Darryl Gove, May 22, 2008

Temporary memory allocated using `alloca`(3C) is allocated on the stack. Stacks start at the top of memory and grow downwards (whereas `heap` grows upwards, low memory to high memory). There are two pointers that help the processor determine where the stack is, the frame pointer and the stack pointer. The usual situation is something like this:

```
Frame pointer -> top of stack
                 variables
Stack pointer -> bottom of stack
```

When a new routine is called and allocates local variables, the resulting stack looks like:

```
              -> top of stack
                 variables
Frame pointer -> old bottom of stack
                 variables
Stack pointer -> bottom of stack
```

The old stack pointer becomes the new frame pointer, a new amount of space is reserved on the stack for the local variables, and then the stack pointer points to the new bottom of the stack.

When alloca is called, the application reserves more memory on the stack. This is basically a case of moving the stack pointer downwards, like this:

```
              -> top of stack
                 variables
Frame pointer -> old bottom of stack
                 variables
                 [alloca'd memory]
Stack pointer -> bottom of stack
```

So rather than the overhead of calling new and free, the memory gets allocated with the cost of very few instructions - often just two. One instruction to calculate the base address of the allocated memory, and one instruction to move the stack pointer.

http://blogs.sun.com/d/entry/alloca_internals

alloca on SPARC

Darryl Gove, May 22, 2008

On SPARC there's a slight complication. The load and store instructions have an offset range of register -4096 to register +4096. To use a larger offset than that it is necessary to put the offset into a register and use that to calculate the address.

If the size of the local variables are less than 4KB, then a load or store instruction can use the frame pointer together with an offset in order to access the memory on the stack. If the stack is greater than 4KB, then it's possible to use the frame pointer to access memory in the upper 4KB range, and the stack pointer to access memory in the lower 4KB. Rather like this diagram shows:

```
frame pointer -> top of stack
                 ^
                 | Upper 4KB can be accessed
                 v using offset+ frame pointer
                 ^
                 | Lower 4KB can be accessed
                 v using offset+ stack pointer
stack pointer -> bottom of stack
```

The complication is when temporary memory is allocated on the stack using alloca, and the size of the local variables exceed 4KB. In this case it's not possible to just shift the stack pointer downwards, since that may cause variables that were previously accessed through the stack pointer to become out of the 4KB offset range, or change the offset from the stack pointer where variables are stored (by an amount which may only be known at runtime). Either of these situations would not be good.

Instead of just shifting the stack pointer, a slightly more complex operation has to be carried out. The memory gets allocated in the middle of the range, and the lower memory gets shifted (or copied) downwards. The end result is something like this:

```
frame pointer -> top of stack
                 ^
                 | Upper 4KB can be accessed
                 v using offset+ frame pointer
                 [Alloca'd memory]
                 ^
                 | Lower 4KB can be accessed
                 v using offset+ stack pointer
stack pointer -> bottom of stack
```

The routine that does this manipulation of memory is called __builtin_alloca (http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/lib/libc/sparc/gen/alloca.s#42). You can see in the code that it moves the stack pointer, and then has a copy loop to move the contents of the stack.

Unfortunately, the need to copy the data means that it takes longer to allocate memory. So if the function `__builtin_alloca` appears in a profile, the first thing to do is to see whether it's possible to reduce the amount of local variables/stack space needed for the routine.

As a footnote, take a look at the equivalent code for the x86 version of __builtin_alloca (http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/lib/libc/i386/gen/alloca.s#33). The x86, being CISC, does not have the limit on the size of the offset that can be used. Hence the x86 code does not need the copy routine to move variables in the stack around.

http://blogs.sun.com/d/entry/alloca_on_sparc

Reading the Tick Counter

Darryl Gove, July 11, 2008

It's often necessary to time the duration of a piece of code. To do this, I often use `gethrtime()`, which is typically a quick call. Obviously, the faster the call, the more accurate I can expect my timing to be. Sometimes the call to `gethrtime` is too long (or perhaps too costly, because it's done so frequently). The next thing to try is to read the `%tick` register.

The `%tick` register is a 64-bit register that gets incremented on every cycle. Since it's stored on the processor, reading the value of the register is a low-cost operation. The only complexity is that being a 64-bit value, it needs special handling under 32-bit codes, where the a 64-bit return value from a function is passed in the `%o0` (upper bits) and `%o1` (lower bits) registers.

The inline template to read the `%tick` register in a 64-bit code is very simple

```
.inline tk,0
   rd %tick,%o0
.end
```

The 32-bit version requires two more instructions to get the return value into the appropriate registers:

```
.inline tk,0
   rd %tick,%o2
   srlx %o2,32,%o0   //upper bits into %o0
   sra %o2,0,%o1     //lower bits into %o1
.end
```

Here's an example code which uses the `%tick` register to get the current value plus an estimate of the cost of reading the `%tick` register:

```
#include <stdio.h>

long long tk();

void main()
{
  printf("value=%llu duration=%llu\n",tk(),-tk()+tk());
}
```

The compile line is:

```
$ cc -O tk.c tk.il
```

The output should be something like:

```
$ a.out
value=4974674895272956 duration=32
```

Indicating that 32 cycles (in this instance) elapsed between the two reads of the `%tick` register. Looking at the disassembly, there are certainly a number of cycles of overhead:

```
10bbc:  95 41 00 00  rd         %tick, %o2
10bc0:  91 32 b0 20  srlx       %o2, 32, %o0
10bc4:  93 3a a0 00  sra        %o2, 0, %o1
10bc8:  97 32 60 00  srl        %o1, 0, %o3
10bcc:  99 2a 30 20  sllx       %o0, 32, %o4
10bd0:  88 12 c0 0c  or         %o3, %o4, %g4
10bd4:  c8 73 a0 60  stx        %g4, [%sp + 96]
10bd8:  95 41 00 00  rd         %tick, %o2
```

This overhead can be reduced by treating the `%tick` register as a 32-bit read, and effectively ignoring the upper bits. For very short duration codes this is probably acceptable, but is unsuitable for longer running code blocks. With this (inelegant hack) the following code is generated, which usually returns a value of 8 cycles on the same platform:

```
10644:  91 41 00 00  rd         %tick, %o0
10648:  b0 07 62 7c  add        %i5, 636, %i0
1064c:  b8 10 00 08  mov        %o0, %i4
10650:  91 41 00 00  rd         %tick, %o0
```

http://blogs.sun.com/d/entry/reading_the_tick_counter

◆ ◆ ◆ CHAPTER 4

Solaris Performance Primer

This chapter describes some freely available Solaris performance monitoring tools from the Tools CD (`http://blogs.sun.com/partnertech/entry/update_solaris_performance_toolscd_3`) and the DTrace toolkit (`http://opensolaris.org/os/community/dtrace/dtracetoolkit/`).

This chapter will help the impatient ones who want to solve 90% of the day-to-day performance issues without long studies by using some key tools like `perfbar` or `netbar`. Being able to scale quickly is pivotal in the web age where the load can grow by an order of magnitude overnight. This chapter is organized as follows:

- "Explore Your System" on page 140 – What are the features and limits of your system? How to analyze your Solaris installation, patches, and so on. Quantify the abilities of your hardware: How many processors, disks, memory, etc. do you have?
- "System Utilization" on page 148 – What is your system actually doing? Checking out your processes, top users, resource consumption. Understanding your system utilization.
- "Process Introspection" on page 158 – What is your process doing? Tools for process introspection. Learn which files are being used, which libraries are being loaded, which call stacks you're currently using, etc.
- "Process Monitoring With `prstat`" on page 169 – Monitoring the processes and threads in projects and zones with `prstat`.
- "Understanding I/O" on page 179 – Understand your IO: Who is writing how to where? How is the disk subsystem doing?
- "Understanding the Network" on page 186 – Understanding network traffic: Which network interface is working how hard? Is your interface overloaded? And so on.
- "Tracing Running Applications" on page 191 – Tracing at large: How to get information about current system call. How to use DTrace to answer common questions like: Who is writing to which file right now?

These sections will hopefully answer the most common performance questions with standard tools available for the Solaris OS.

Other sites which have Solaris Performance related information are:

- The ones who are interested in a more complete (and professional) resource may want to consider Darryl Gove's book *Solaris Application Programming* (`http://vig.pearsoned.com/store/product/1,3498,store-6404_isbn-0138134553,00.html`). A great chapter detailing the system tools is available for free online (`http://www.sun.com/books/documents/solaris_app_programming_ch04.pdf`).
- Solaris Internals wiki (`http://www.solarisinternals.com/`): A comprehensive wiki with available tools and best practise documents.
- *Solaris Performance and Tools* (`http://developers.sun.com/solaris/articles/solaris_perftools.html`): a book by Richard McDougall, Jim Mauro and Brendan Gregg from July 2006.

  This chapter was created by Thomas Bastian, a performance tuning specialist who gained his knowledge through many performance projects with the key software partners in EMEA from Sun Microsystems

Explore Your System

Stefan Schneider and Thomas Bastian, April 25, 2008

Solaris System Inventory

This section deals with your system. Solaris systems may have one CPU or hundreds. Solaris systems may have a single disk or entire farms. Anyone who deals with performance has to know what the quantitative aspects of the system are. The commands listed here will answer these questions.

It's pivotal that you get an understanding of the components of the system you want to tune. Knowing the hardware components and the installed software allows you to understand the quantitative limits of the system.

Solaris offers a wealth of commands to identify the characteristics of the running system. This section discusses commands that help administrators and software developers understand and document accurately the hardware and software specifications.

`uname` – Printing Information About the Current System

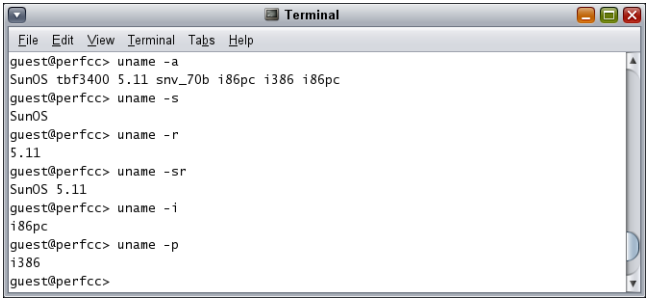
The `uname`(1) utility prints information about the current system on the standard output. The command outputs detailed information about the current system on operating system software revision level, processor architecture and platform attributes.

The table below lists selected options to `uname`:

TABLE 4–1 `uname` Selection Options

Option	Comments
`-a`	Prints basic information currently available from the system.
`-s`	Prints the name of the operating system.
`-r`	Prints the operating system release level.
`-i`	Prints the name of the platform.
`-p`	Prints the processor type or ISA [Instruction Set Architecture].

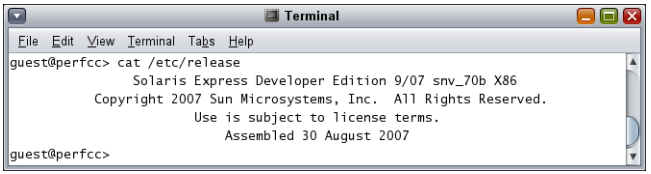
The following example shows sample output from `uname`.



`/etc/release` – Detailed Information About the Operating System

The file `/etc/release` contains detailed information about the operating system. The content provided allows engineering or support staff to unambiguously identify the Solaris release running on the current system.

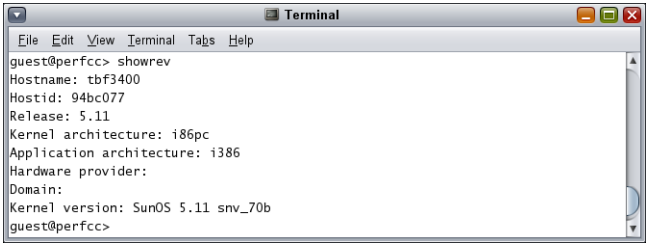
The following example shows sample output for the `/etc/release` file.



`showrev` – Show Machine, Software and Patch Revision

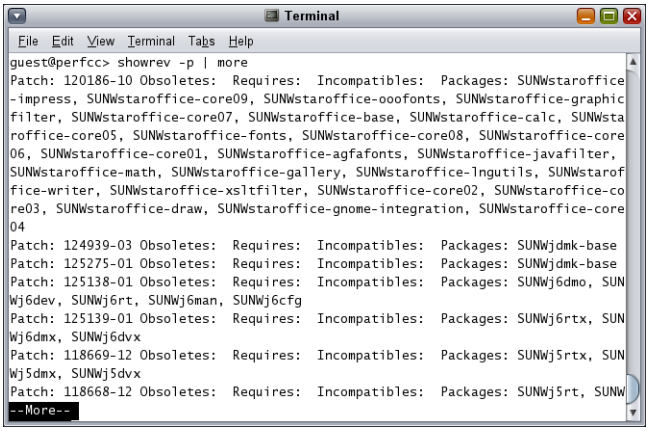
The showrev (`http://docs.sun.com/app/docs/doc/816-0211/6m6nc676p?a=view`) command shows machine, software revision and patch revision information. With no arguments, `showrev` shows the system revision information including hostname, hostid, release, kernel architecture, application architecture, hardware provider, domain and kernel version.

The following example shows the machine and software revision information output by `showrev`.



To list patches installed on the current system, use the `showrev` command with the `-p` argument.

The following example illustrates sample patch information output by `showrev -p`.



`isainfo` – Describe Instruction Set Architectures

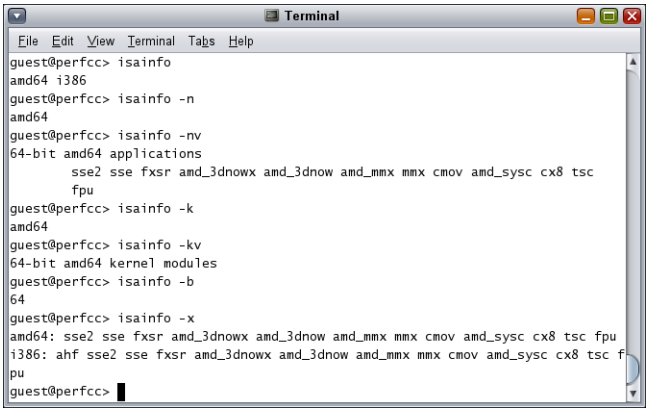
The `isainfo`(1) command describes instruction set architectures. The `isainfo` utility is used to identify various attributes of the instruction set architectures supported on the currently running system. It can answer whether 64-bit applications are supported, or if the running kernel uses 32-bit or 64-bit device drivers.

The table below lists selected options to `isainfo`:

TABLE 4–2 `isainfo` - Selected Options

Option	Comments
<none>	Prints the names of the native instruction sets for portable applications.
`-n`	Prints the name of the native instruction set used by portable applications.
`-k`	Prints the name of the instruction set(s) used by the operating system kernel components such as device drivers and STREAMS modules.
`-b`	Prints the number of bits in the address space of the native instruction set.

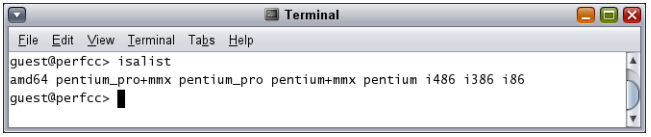
The following example shows sample instruction set architecture information output by the `isainfo` command.



`isalist` – Display Native Instruction Sets Executable on This Platform

The `isalist`(1) command displays the native instruction sets executable on this platform. The names are space-separated and are ordered in the sense of best performance. Earlier-named instruction sets may contain more instructions than later-named instruction sets. A program that is compiled for an earlier-named instruction sets will most likely run faster on this machine than the same program compiled for a later-named instruction set.

The following example shows sample native instruction set information output by the `isalist` command.



`psrinfo` – Display Information About Processors

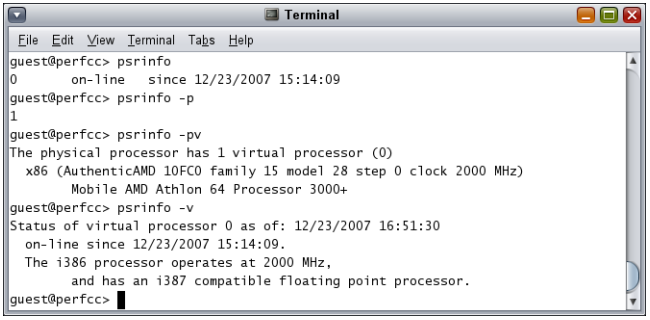
The `psrinfo`(1M) command displays information about processors. Each physical processor may support multiple virtual processors. Each virtual processor is an entity with its own interrupt ID, capable of executing independent threads.

The table below lists selected options to `psrinfo`.

TABLE 4–3 `psrinfo` - Selected Options

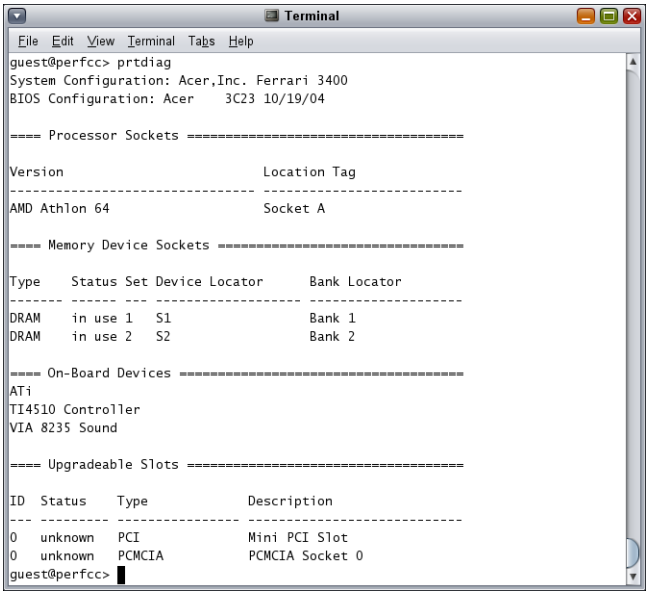
Option	Comments
<none>	Prints one line for each configured processor, displaying whether it is online, non-interruptible (designated by no-intr), spare, off-line, faulted or powered off, and when that status last changed.
`-p`	Prints the number of physical processors in a system.
`-v`	Verbose mode. Prints additional information about the specified processors, including: processor type, floating point unit type and clock speed. If any of this information cannot be determined, `psrinfo` displays unknown.

The following example shows sample processor information output by the `psrinfo` command.



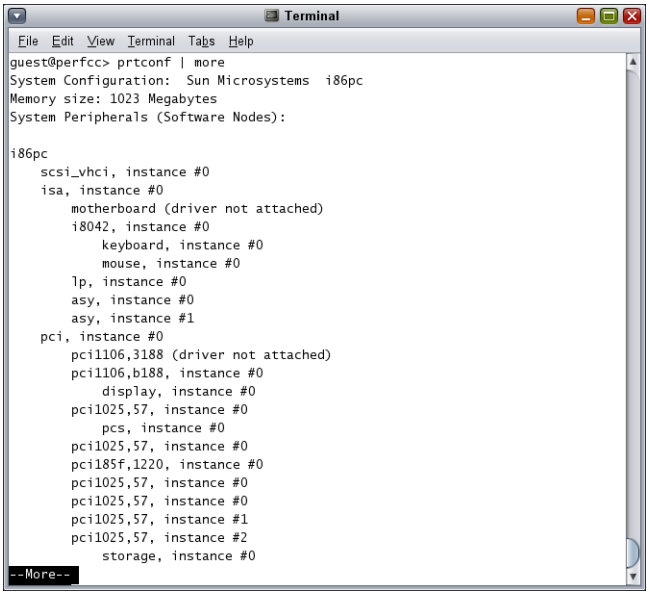
`prtdiag` – Display System Diagnostic Information

The `prtdiag`(1M) command displays system diagnostic information. On Solaris 10 for x86/x64 systems, the command is only available with Solaris 10 01/06 or higher.



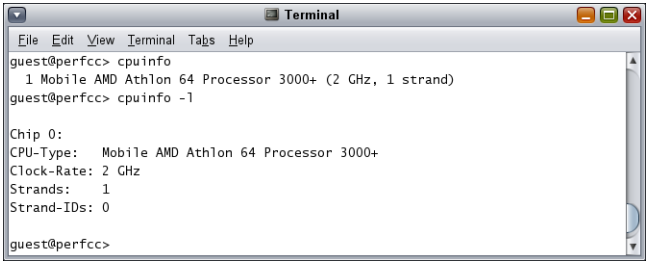
`prtconf` – Print System Configuration

The `prtconf`(1M) command prints system configuration information. The output includes the total amount of memory, and the configuration of system peripherals formatted as a device tree.



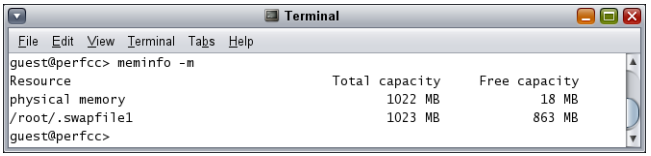
`cpuinfo` [Tools CD] – Display CPU Configuration

The `cpuinfo` utility prints detailed information about the CPU type and characteristics (number, type, clock and strands) of the running system.



`meminfo` [Tools CD] – Display Physical Memory, Swap Devices, Files

The `meminfo` tool displays configuration of physical memory and swap devices or files.



http://blogs.sun.com/partnertech/entry/solaris_performance_primer_explore_your

System Utilization

Stefan Schneider and Thomas Bastian, April 28, 2008

Understanding System Utilization

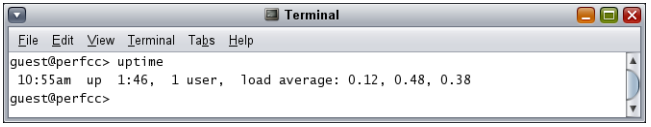
The following chapter introduces and demonstrates tools that help out in understanding overall system utilization.

`uptime` – Print Load Average

The easiest way to gain an overview on:

- How long a system has been running
- Current CPU load averages
- How many active users

is with the command `uptime`(1). The following example shows CPU load averages.



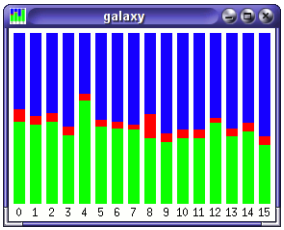
The numbers printed to the right of "load average" are the 1-, 5- and 15-minute load averages of the system. The load average numbers give a measure of the number of runnable threads and running threads. Therefore, the number has to be considered in relation to the number of active CPUs in a system. For example, a load average of three (3) on a single CPU system would indicate some CPU overloading, while the same load average on a thirty-two (32)-way system would indicate an unloaded system.

`perfbar` [Tools CD] – A Lightweight CPU Meter

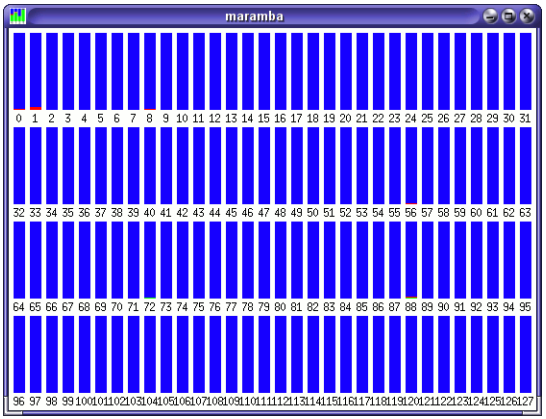
`perfbar` is a tool that displays a single bar graph that color codes system activity. The colors are as follows:

- Blue = System idle.
- Red = System time.
- Green = CPU time.
- Yellow = I/O activity (obsolete as of Solaris 10)

The following illustration shows sample output of `perfbar` for a system with 16 CPU cores.



Perfbar has been enhanced in Version 1.2 to provide better support for servers with many CPUs through a multiline visualization. The following example shows visualization of a Sun T5240 system with 128 strands (execution units).



`perfbar` can be called without specifying any command line arguments. `perfbar` provides a large number of options which can be viewed with the `-h` option:

```
$ perfbar -h

perfbar 1.2

maintained by Ralph Bogendoerfer
based on the original perfbar by:
Joe Eykholt, George Cameron, Jeff Bonwick, Bob Larson

Usage: perfbar [X-options] [tool-options]
   supported X-options:
      -display <display> or -disp <display>

      -geometry <geometry> or -geo <geometry>
      -background <background> or -bg <background>
      -foreground <foreground> or -fg <foreground>

      -font <font> or -fn <font>
      -title <title> or -t <title>
      -iconic or -icon
      -decoration or -deco
   supported tool-options:
      -h, -H, -? or -help: this help
      -v or -V: verbose
      -r or -rows: number of rows to display, default 1
      -bw or -barwidth: width of CPU bar, default 12
      -bh or -barheight: height of CPU bar, default 180
      -i or -idle: idle color, default blue
      -u or -user: user color, default green
      -s or -system: system color, default red
      -w or -wait: wait color, default yellow
      -int or -interval: interval for display updates (in ms),default 100
      -si or -statsint: interval for stats updates (in display intervals), default 1
      -avg or -smooth: number of values for average calculation, default 8
```

There are also a number of keystrokes understood by the tool:

- Q or q: Quit
- R or r: Resize - this changes the window to the default size according to the number of CPU bars, rows and the chosen bar width and height.
- Number keys 1 - 9: Display this number of rows.
- + and -: Increase or decrease number of rows displayed.

The tool is currently available as a beta in version 1.2. This latest version is not yet part of the Performance Tools CD 3.0. The engineers from the Sun Solution Center in Langen/German have made it available for free through:

- perfbar: Solaris SPARC (`http://blogs.sun.com/partnertech/resource/tools/perfbar.sparc`)
- perfbar: Solaris x86 (`http://blogs.sun.com/partnertech/resource/tools/perfbar.i386`)

`cpubar` [Tools CD] – A CPU Meter Showing Swap and Run Queue

`cpubar` displays one bar-graph for each processor with the processor speed(s) displayed on top. Each bar-graph is divided in four areas (top to bottom):

- Blue – CPU is available.
- Yellow – CPU is waiting for one or more I/O to complete (N/A on Solaris 10 and later).
- Red – CPU is running in kernel space.
- Green – CPU is running in user space.

As with `netbar` and `iobar`, a red and a dashed black & white marker shows the maximum and average used ratios respectively.

The bar-graphs labeled 'r', 'b' and 'w' display the run, blocked and wait queues. A non-empty wait queue is usually a symptom of a previous persistent RAM shortage. The total number of processes is displayed on top of these three bars.

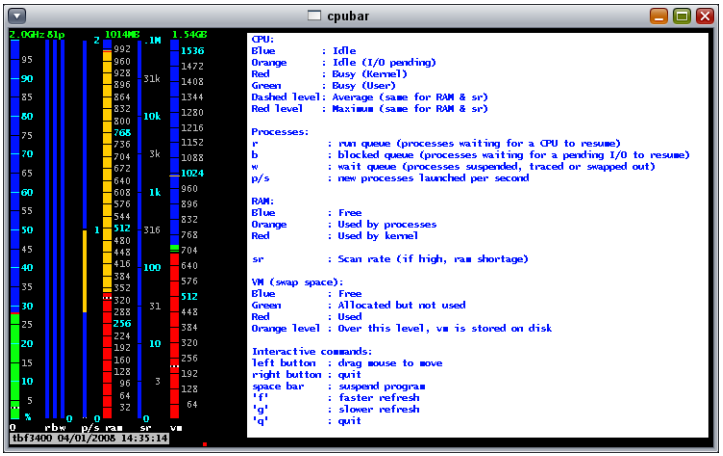
The bar-graph labeled 'p/s' displays the process creation rate per second.

The bar-graph labeled 'RAM' displays the RAM usage (red=kernel, yellow=user, blue=free). The total RAM is displayed on top.

The bar-graph ('sr') displays (using a logarithmic scale) the scan rate. (A high level of scans is usually a symptom of RAM shortage.)

The bar-graph labeled 'SWAP' displaysg the SWAP (a.k.a Virtual Memory) usage (red=used, yellow=reserved, blue=free). The total SWAP space is displayed on top.

The following illustration shows sample output of `cpubar`.

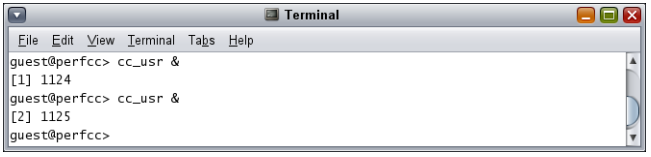


`vmstat` – System Glimpse

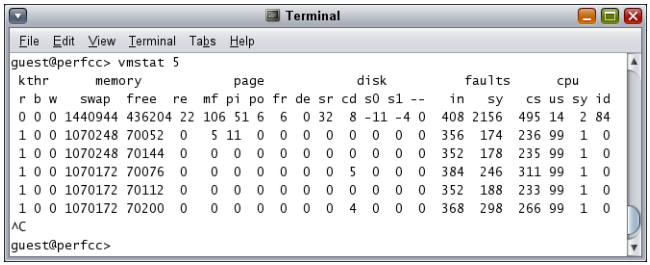
The `vmstat`(1M) tool provides a glimpse of the current system behavior in a one-line summary including both CPU utilization and saturation.

In its simplest form, the command `vmstat <interval>` (that is, `vmstat 5`) will report one line of statistics every <interval> seconds. The first line can be ignored as it is the summary since boot. All other lines report statistics of samples taken every <interval> seconds. The underlying statistics collection mechanism is based on `kstat` (see `kstat`(1)).

Let's run two copies of a CPU intensive application (`cc_usr`) and look at the output of `vmstat 5`. First start two (2) instances of the `cc_usr` program.



Now let's run `vmstat 5` and watch its output.



First observe the `cpu:id` column which represents the system idle time (here 0%). Then look at the `kthr:r` column which represents the total number of runnable threads on dispatcher queues (here 1).

From this simple experiment, one can conclude that the system idle time for the five-second samples was always 0, indicating 100% utilization. On the other hand, `kthr:r` was mostly one and sustained indicating a modest saturation for this single CPU system (remember we launched two (2) CPU intensive applications).

A couple of notes with regard to CPU utilization:

- 100% utilization may be fine for your system. Think about a high-performance computing job: the aim will be to maximize utilization of the CPU.
- Values of `kthr:r` greater than zero indicate some CPU saturation (that is, more jobs would like to run but cannot because no CPU was available). However, performance degradation should be gradual.
- Sampling interval is important. Don't choose too small or too large intervals.

`vmstat` reports some additional information that can be interesting, as listed in the following table.

TABLE 4–4 Columns of `vmstat` Output

Column	Comments
`in`	Number of interrupts per second.
`sys`	Number of system calls per second.
`cs`	Number of context switches per second (both voluntary and involuntary).
`us`	Percent user time: time the CPUs spent processing user-mode threads.
`sy`	Percent system time: time the CPUs spent processing system calls on behalf of user-mode threads, plus the time spent processing kernel threads.
`id`	Percent of time the CPUs are waiting for runnable threads.

`mpstat` – Report per-Processor or per-Processor Set Statistics

The `mpstat`(1M) command reports processor statistics in tabular form. Each row of the table represents the activity of one processor. The first table summarizes all activity since boot. Each subsequent table summarizes activity for the preceding interval. The output table includest the columns listed in the following table.

TABLE 4–5 Columns of `mpstat` Output

Column	Comments
`CPU`	Prints processor ID.
`minf`	Minor faults (per second).
`mjf`	Major faults (per second).
`xcal`	Inter-processor cross-calls (per second).
`intr`	Interrupts (per second).
`ithr`	Interrupts as threads (not counting clock interrupt) (per second).
`csw`	Context switches (per second).
`icsw`	Involuntary context switches (per second).
`migr`	Thread migrations (to another processor) (per second).
`smtx`	Spins on mutexes (lock not acquired on first try) (per second).
`srw`	Spins on readers/writer locks (lock not acquired on first try) (per second).
`syscl`	System calls (per second).
`usr`	Percent user time.
`sys`	Percent system time.
`wt`	Always 0.
`idl`	Percent idle time.

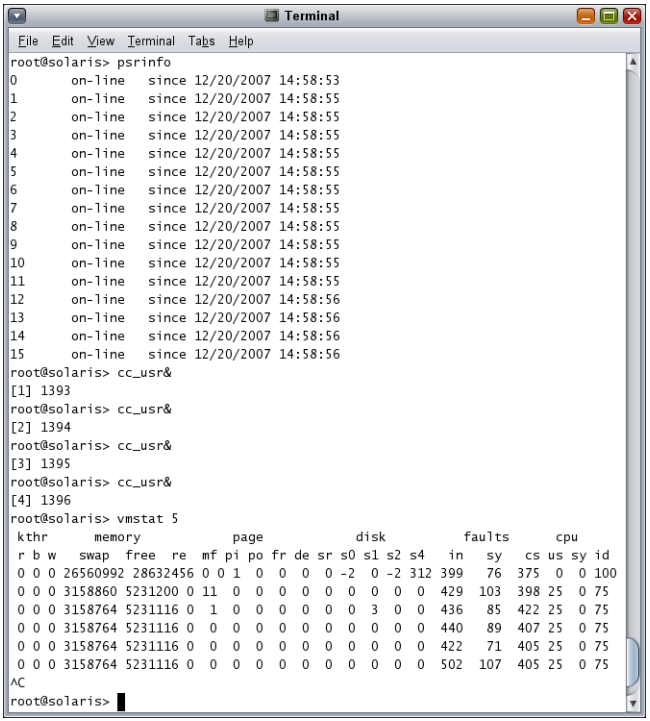
The reported statistics can be broken down into the following categories:

- Processor utilization: see columns `usr`, `sys` and `idl` for a measure of CPU utilization on each CPU.
- System call activity: see the `syscl` column for the number of system call per second on each CPU.

- Scheduler activity: see column `csw` and column `icsw`. As the ratio `icsw`/`csw` comes closer to one (1), threads get pre-empted because of higher priority threads or expiration of their time quantum. Also the column `migr` displays the number of times the OS scheduler moves ready-to-run threads to an idle processor. If possible, the OS tries to keep the threads on the last processor on which it ran. If that processor is busy, the thread migrates.
- Locking activity: column `smtx` indicates the number of mutex contention events in the kernel. Column `srw` indicates the number of reader-writer lock contention events in the kernel.

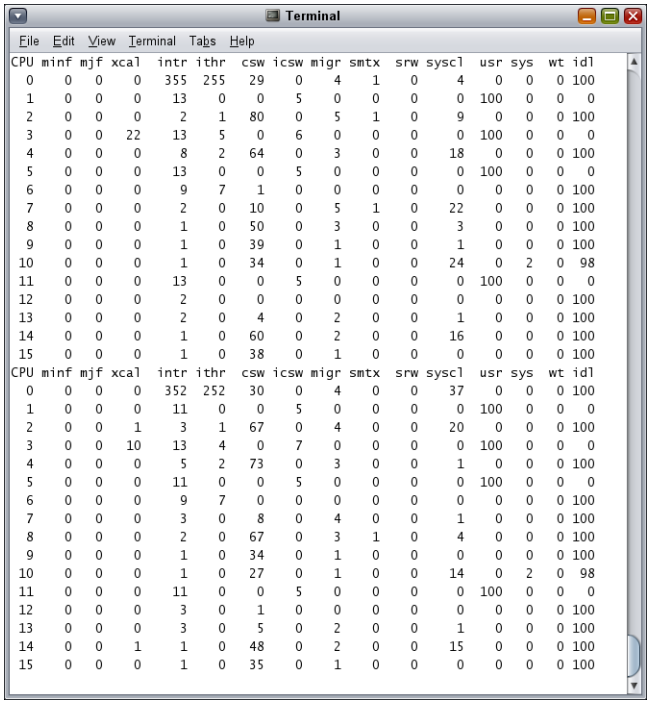
Now, consider the following sixteen-way (16) system used for test. This time four (4) instances of the `cc_usr` program were started and the output of `vmstat 5` and `mpstat 5` recorded.

Below, observe the output of processor information. Then the starting of the four (4) copies of the program and last the output of `vmstat 5`.



Rightly, `vmstat` reports a user time of 25% because one-fourth (¼) of the system is used (remember 4 programs started, 16 available CPUs, which is 4/16 or 25%).

Now let's look at the output of `mpstat 5`.



In the above output (two sets of statistics), one can clearly identify the four running instances of `cc_usr` on CPUs 1, 3, 5 and 11. All these CPUs are reported with 100% user time.

`vmstat` – Monitoring Paging Activity

The `vmstat` command can also be used to report on system paging activity with the `-p` option. Using this form of the command, one can quickly get a clear picture on whether the system is paging because of file I/O (OK) or paging because of physical memory shortage (BAD).

Use the command as follows: `vmstat -p <interval in seconds>`. The output format includes the information shown in the following table.

TABLE 4–6 `vmstat` Output Columns

Column	Description
`swap`	Available swap space in Kbytes.
`free`	Amount of free memory in Kbytes.
`re`	Page reclaims - number of page reclaims from the cache list (per second).

TABLE 4–6 `vmstat` Output Columns *(Continued)*

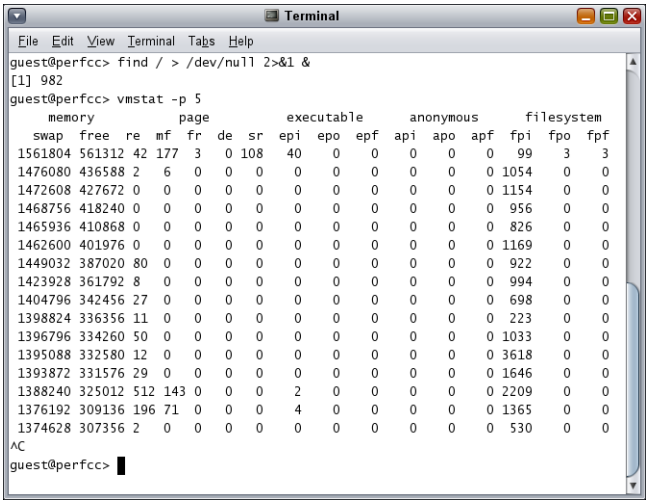
Column	Description
`mf`	Minor faults - number of pages attached to an address space (per second)
fr	Page frees in Kbytes per second.
`de`	Calculated anticipated short-term memory shortfall in Kbytes.
`sr`	Scan rate - number of pages scanned by the page scanner per second.
`epi`	Executable page-ins in Kbytes per second.
`epo`	Executable page-outs in Kbytes per second.
`epf`	Executable page-frees in Kbytes per second.
`api`	Anonymous page-ins in Kbytes per second.
`apo`	Anonymous page-outs in Kbytes per second.
`apf`	Anonymous page-frees in Kbytes per second.
`fpi`	File system page-ins in Kbytes per second.
`fpo`	File system page-outs in Kbytes per second.
`fpf`	File system page-frees in Kbytes per second.

As an example of `vmstat -p` output, let's try the following commands:

```
find / > /dev/null 2>&1
```

and then monitor paging activity with: `vmstat -p 5`

As can be seen from the output, the system is showing paging activity because of file system read I/O (column `fpi`).



http://blogs.sun.com/partnertech/entry/solaris_performance_primer_understanding_system

Process Introspection

Stefan Schneider and Thomas Bastian, April 29, 2008

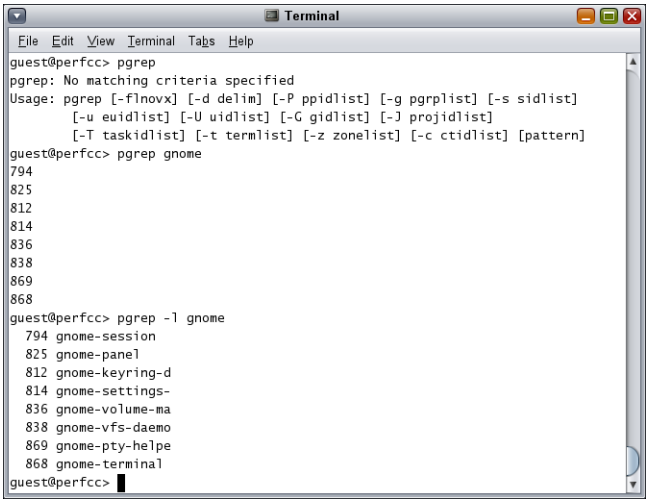
Process Introspection: What Is My Application Doing?

The next step in a performance analysis is to figure out what the application is doing. Configuring application is one thing. Checking if the application actually pulled all configuration information is another thing. The tools below tell you what your application is doing.

The Solaris OS provides a large collection of tools to list and control processes. For an overview and detailed description, please refer to the manual pages of `proc`(1). The following chapter introduces the most commonly used commands.

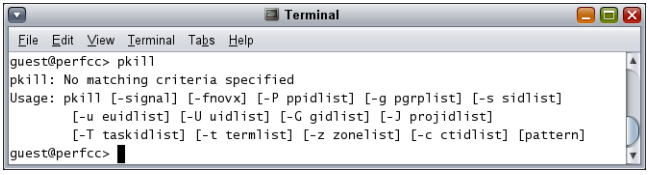
`pgrep` – Find Processes by Name and Other Attributes

The `pgrep`(1) command finds processes by name and other attributes. For that, the `pgrep` utility examines the active processes on the system and reports the process IDs of the processes whose attributes match the criteria specified on the command line. Each process ID is printed as a decimal value and is separated from the next ID by a delimiter string, which defaults to a newline.



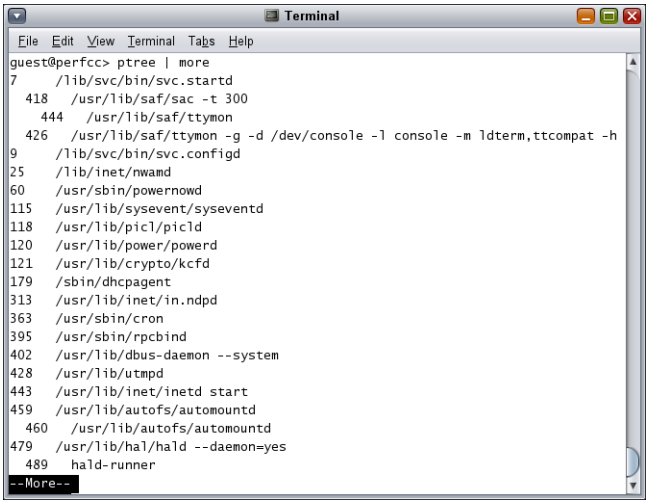
`pkill` – Signal Processes by Name and Other Attributes

The `pkill`(1) command signals processes by name and other attributes. `pkill` functions identically to `pgrep`, except that each matching process is signaled as if by `kill`(1) instead of having its process ID printed. A signal name or number may be specified as the first command line option to `pkill`.



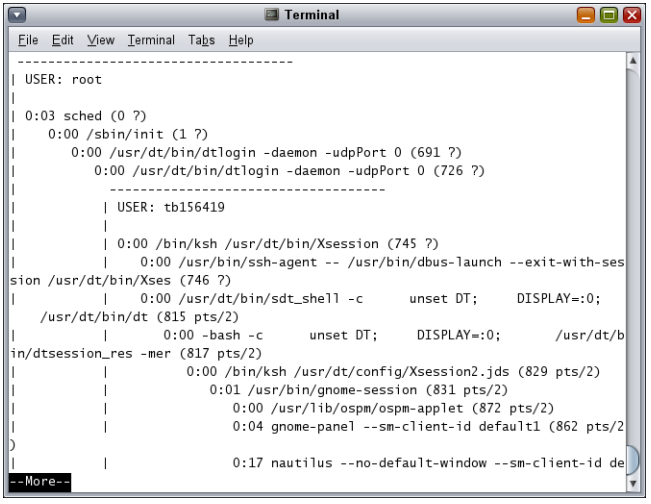
`ptree` – Print Process Trees

The `ptree`(1) command prints parent-child relationship of processes. For that, it prints the process trees containing the specified pids or users, with child processes indented from their respective parent processes. An argument of all digits is taken to be a process-ID, otherwise it is assumed to be a user login name. The default is all processes.



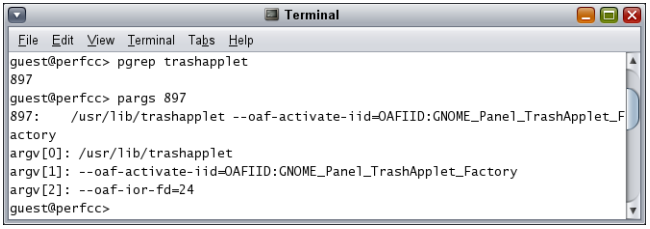
sta [Tools CD] – Print Process Trees

The sta tool provides similar output to ptree, as shown in the following illustration.



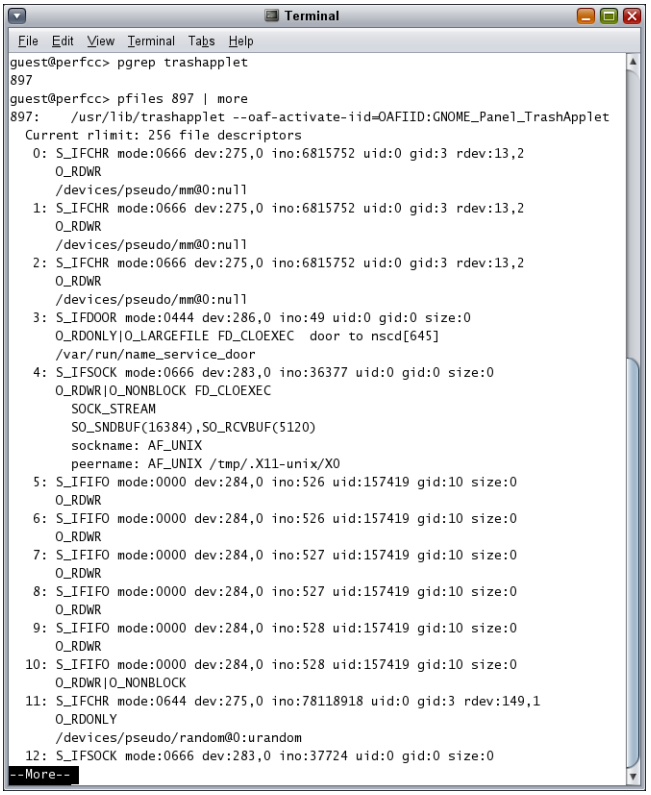
`pargs` – Print Process Arguments, Environment, or Auxiliary Vector

The `pargs`(1) utility examines a target process or process core file and prints arguments, environment variables and values, or the process auxiliary vector.



`pfiles` – Report on Open Files in Process

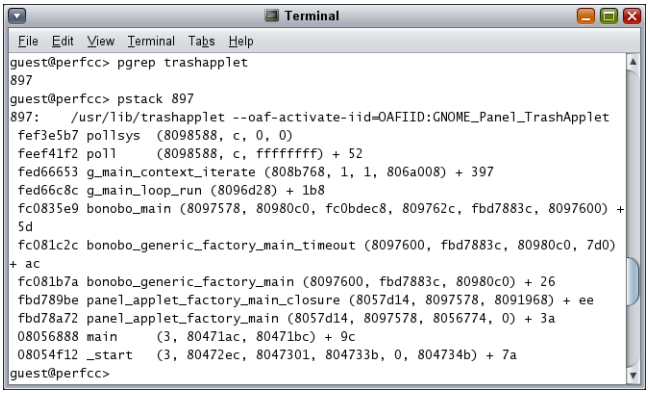
The `pfiles`(1) command reports `fstat`(2) and `fcntl`(2) information for all open files in each process. In addition, a path to the file is reported if the information is available from `/proc/pid/path`. This is not necessarily the same name used to open the file. See `proc`(4) for more information.



`pstack` – Print `lwp/process` Stack Trace

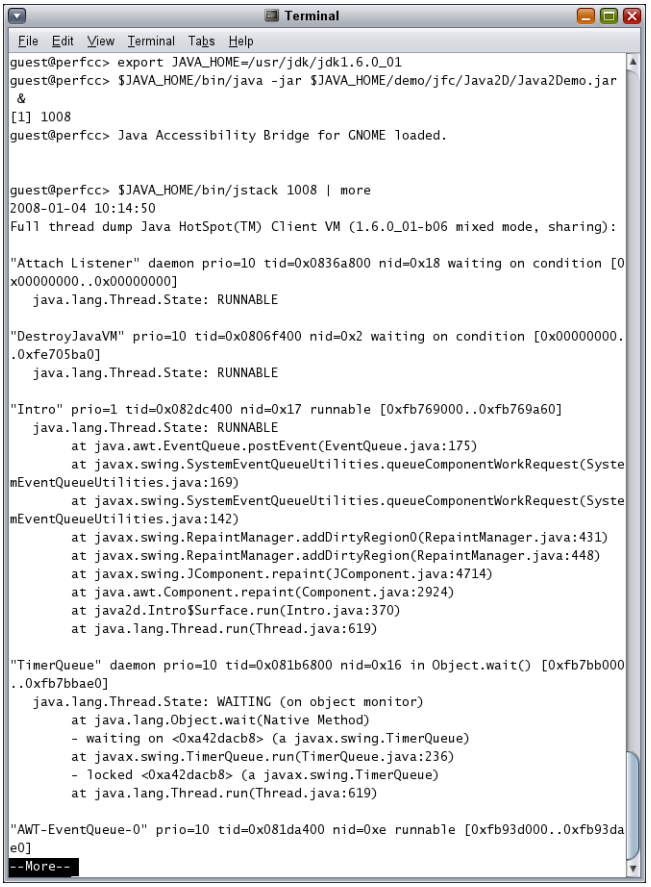
The `pstack`(1) command prints a hex+symbolic stack trace for each process or specified lwps in each process.

Note: Use `jstack` for Java processes



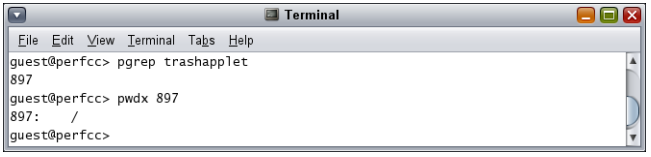
`jstack` – Print Java Thread Stack Trace

The `jstack` command (found in `$JAVA_HOME/bin`) prints Java stack traces of Java threads for a given Java process or core file or a remote debug server. For each Java frame, the full class name, method name, "bci" (byte code index) and line number, if available, are printed.



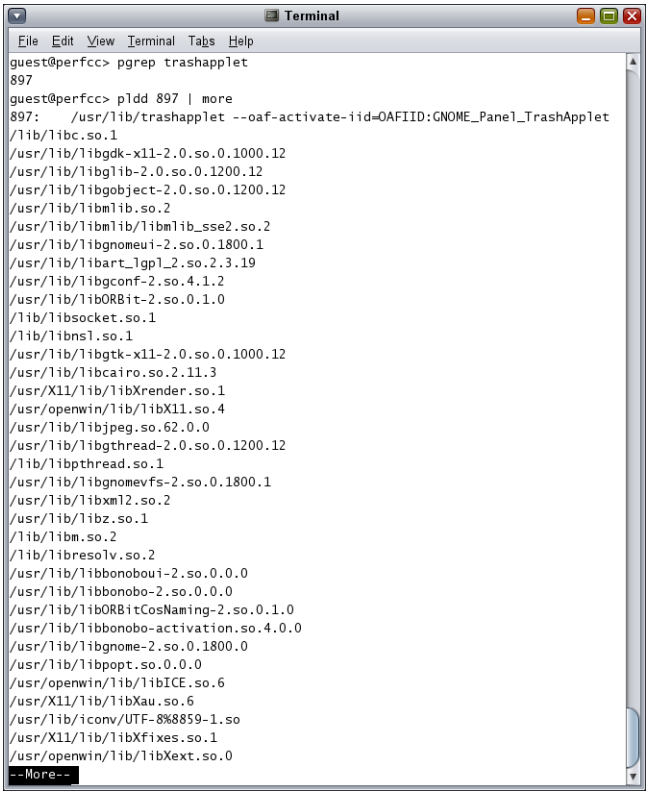
pwdx – Print Process Current Working Directory

The pwdx(1) utility prints the current working directory of each process.



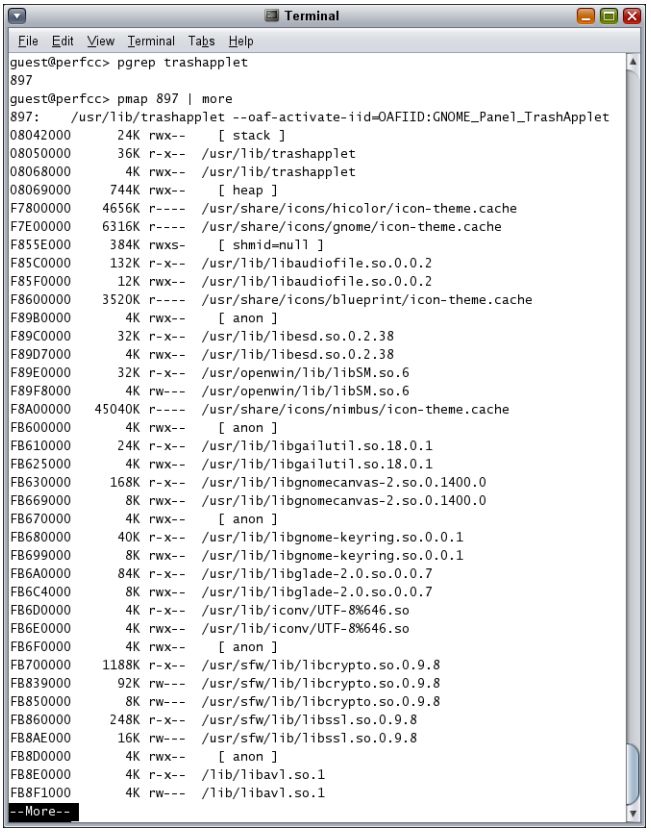
pldd – Print Process Dynamic Libraries

The pldd (1) command lists the dynamic libraries linked into each process, including shared objects explicitly attached using dlopen(3C). See also ldd(1).

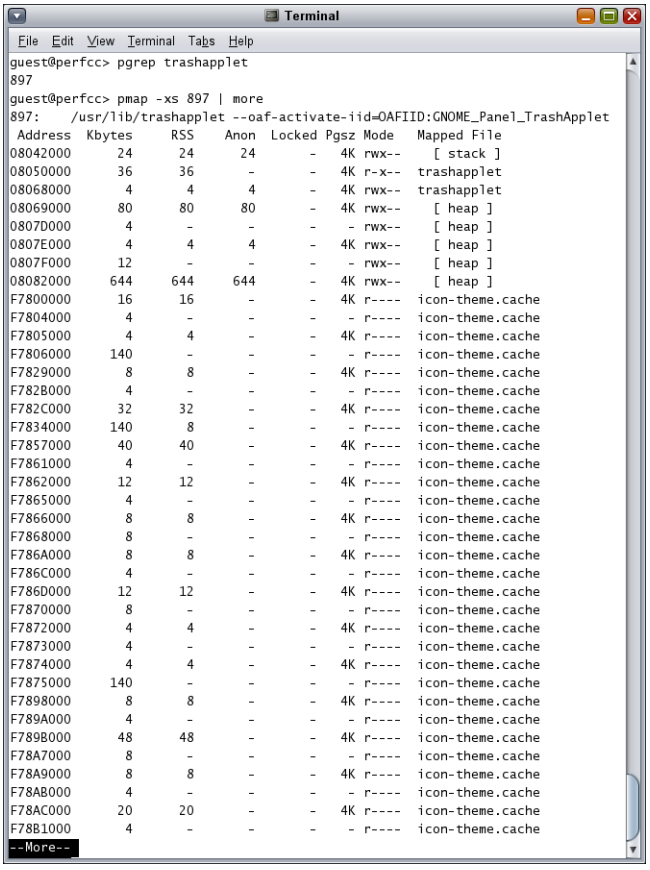


`pmap` – Display Information About the Address Space of a Process

The `pmap`(1) utility prints information about the address space of a process. By default, `pmap` displays all of the mappings in the virtual address order they are mapped into the process. The mapping size, flags and mapped object name are shown.

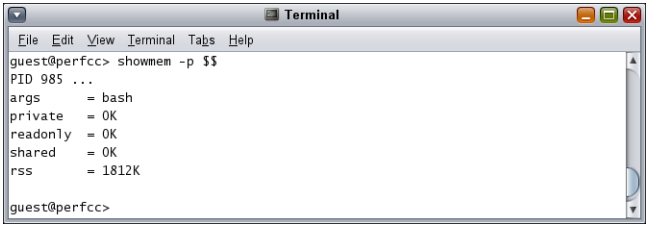


An extended output is available by adding the `-x` option (additional information about each mapping) and the `-s` option (additional page size information).



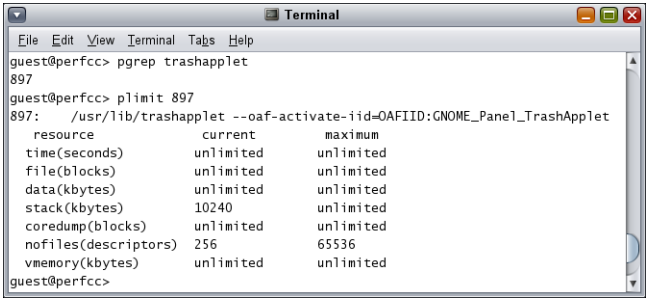
`showmem` [Tools CD] – Process Private and Shared Memory Usage

The `showmem` utility wraps around `pmap` and `ps` to determine how much private and shared memory a process is using.

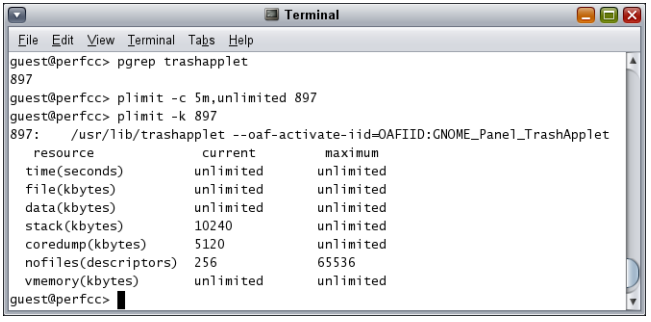


`plimit` – Get or Set the Resource Limits of Running Processes

In the first form, the `plimit`(1) utility prints the resource limits of running processes.



In the second form, the `plimit` utility sets the soft (current) limit and/or the hard (maximum) limit of the indicated resource(s) in the processes identified by the process-ID list, `pid`. As an example, let's limit the current (soft) core file size of the `trashapplet` process with PID 897 to five (5) MB, using the command: `plimit -c 5m,unlimited 897`.



http://blogs.sun.com/partnertech/entry/solaris_performance_prime_process_introspection

Process Monitoring With prstat

Stefan Schneider and Thomas Bastian, April 30, 2008

prstat: Process Monitoring in the System

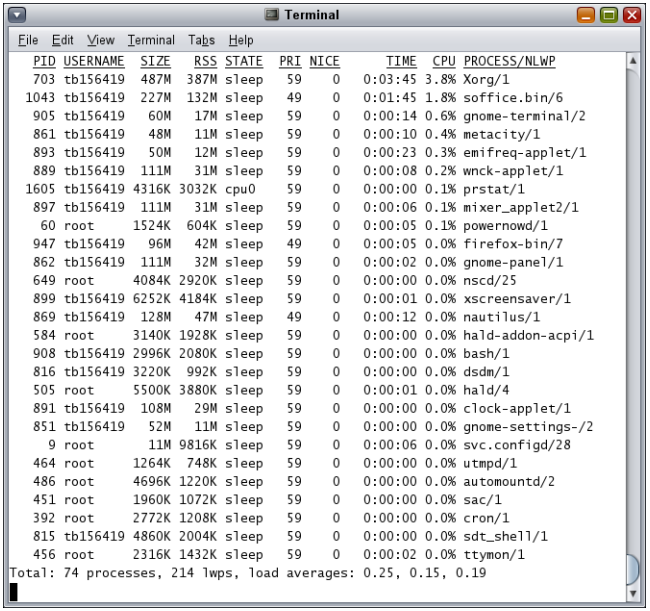
The following chapter takes a deeper look at the Solaris tool prstat(1), the all-around utility that helps understand system utilization.

prstat – The All-Around Utility

One of the most important and widely used utility found in Solaris is prstat(1), which gives fast answers to questions like:

- How much is my system utilized in terms of CPU and memory?
- Which processes (or users, zones, projects, tasks) are utilizing my system?
- How are processes/threads using my system (user bound, I/O bound)?

In its simplest form, the command prstat <interval> (that is, prstat 2) will examine all processes and report statistics sorted by CPU usage.



As can be seen in the screen capture, processes are ordered from top (highest) to bottom (lowest) according to their current CPU usage (in % - 100% means all system CPUs are fully utilized). For each process in the list, following information is provided:

- PID: the process ID of the process.
- USERNAME: the real user (login) name or real user ID.
- SIZE: the total virtual memory size of the process, including all mapped files and devices, in kilobytes (K), megabytes (M), or gigabytes (G).
- RSS: the resident set size of the process (RSS), in kilobytes (K), megabytes (M), or gigabytes (G).
- STATE: the state of the process (cpuN/sleep/wait/run/zombie/stop).
- PRI: the priority of the process. Larger numbers mean higher priority.
- NICE: nice value used in priority computation. Only processes in certain scheduling classes have a nice value.
- TIME: the cumulative execution time for the process.
- CPU: The percentage of recent CPU time used by the process. If the process is executing in a non-global zone and the pools facility is active, the percentage will be that of the processors in the processor set in use by the pool to which the zone is bound.
- PROCESS: the name of the process (name of executed file).
- NLWP: the number of lwps in the process.

The <interval> argument given to prstat is the sampling/refresh interval in seconds.

prstat – Sorting

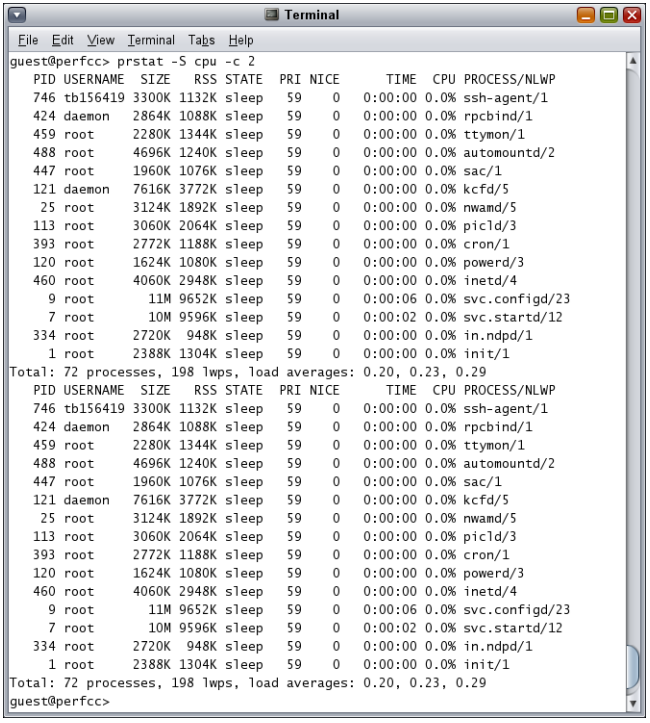
The prstat output can be sorted by another criteria than CPU usage. Use the option -s (descending) or -S (ascending) with the criteria of choice (for example, prstat -s time 2).

TABLE 4–7 prstat Sort Options

Criteria	Comments
cpu	Sort by process CPU usage. This is the default.
pri	Sort by process priority.
rss	Sort by resident set size.
size	Sort by size of process image.
time	Sort by process execution time.

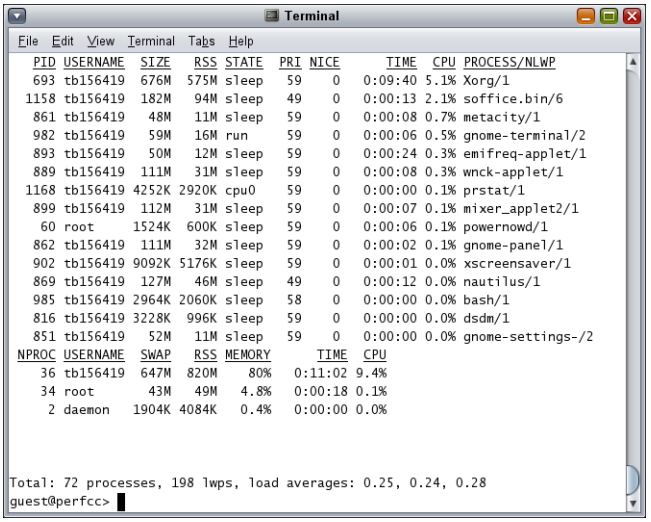
prstat – Continuous Mode

With the option -c to prstat, new reports are printed below previous ones, instead of overprinting them. This is especially useful when gathering information to a file (for example, prstat -c 2 > prstat.txt). The option -n <number of output lines> can be used to set the maximum length of a report. The following illustration shows a continuous report sorted by ascending order of CPU usage.



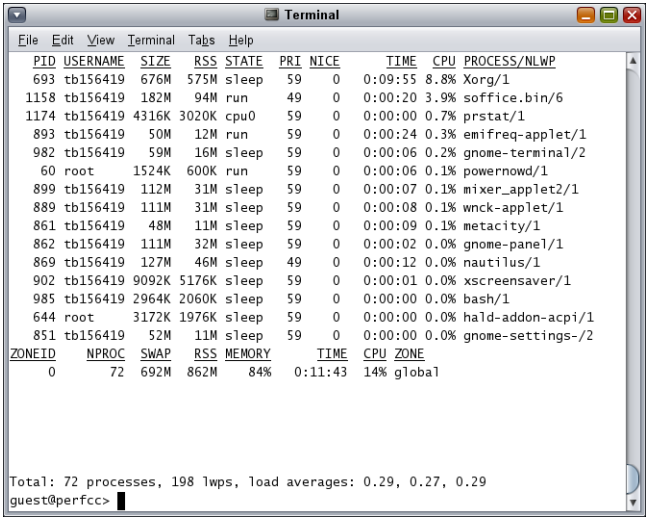
prstat – Report by Users

With the option -a or -t to prstat, additional reports about users are printed.



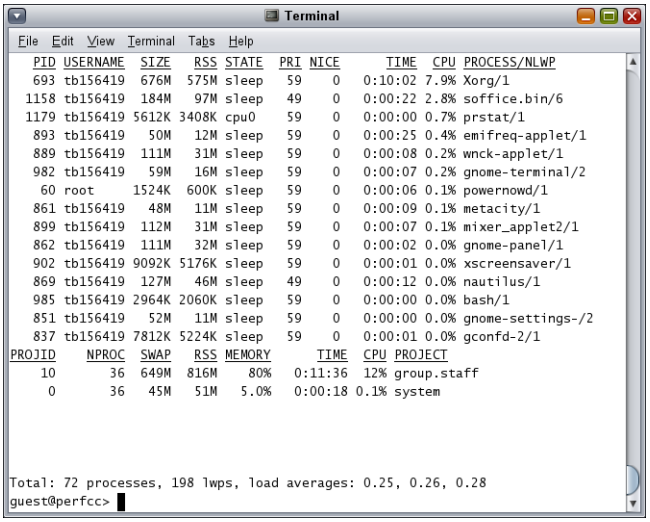
prstat – Report by Zones

With the option -Z to prstat, additional reports about zones are printed.



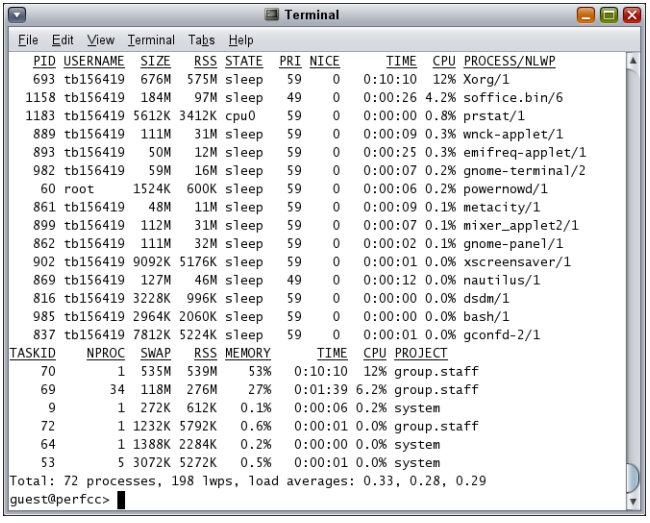
prstat – Report by Projects

With the option -J to prstat, additional reports about projects are printed. For more information, see projects(1).



`prstat` – Report by Tasks

With the option `-T` to `prstat`, additional reports about tasks are printed. For more information, see `newtask`(1).

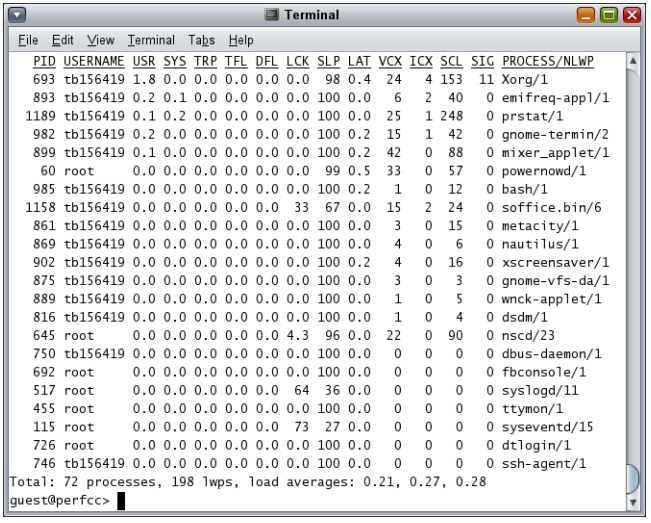


prstat – Microstate Accounting

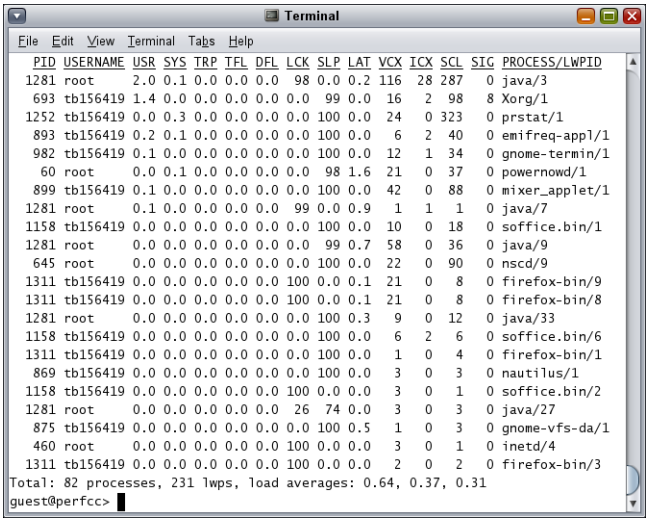
Unlike other operating systems that gather CPU statistics every clock tick or every fixed time interval (typically every hundredth of a second), Solaris 10 incorporates a technology called "microstate accounting" that uses high-resolution timestamps to measure CPU statistics for every event, thus producing extremely accurate statistics.

The microstate accounting system maintains accurate time counters for threads as well as CPUs. Thread-based microstate accounting tracks several meaningful states per thread in addition to user and system time, which include trap time, lock time, sleep time and latency time. `prstat` reports the per-process (option `-m`) or per-thread (option `-mL`) microstates.

The output for the command `prstat -mL 2` shown in the following illustration displays microstates for the running system. Looking at the top line with PID 693, one can see that the process Xorg spends 1.8% of its time in userland while sleeping (98%) the rest of its time.

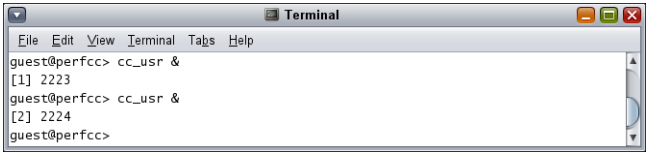


The output for the command `prstat -m 2` shown in the following illustration displays per-thread microstates for the running system. Looking at the line with PID 1311 (in the middle of the display), one can see the microstates for LWP #9 and LWP #8 of the process `firefox-bin`.

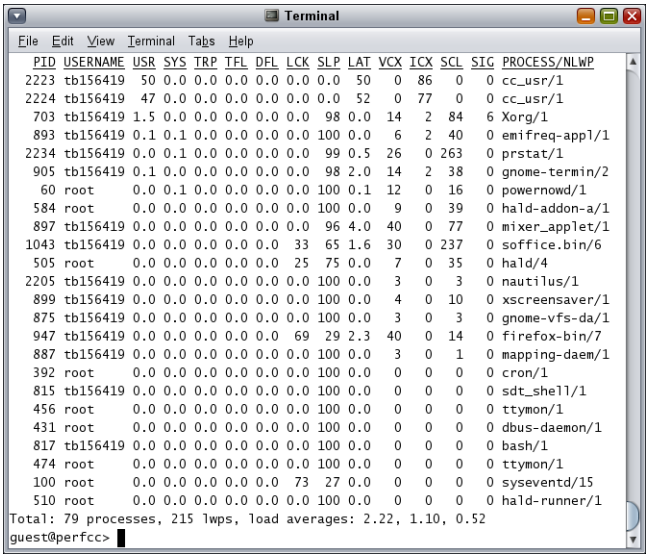


prstat Usage Scenario – CPU Latency

One important measure for CPU saturation is the latency (LAT column) output of prstat. This indicates the time a process spends waiting to get onto a CPU. Let's once again start two (2) copies of our CPU intensive application cc_usr.



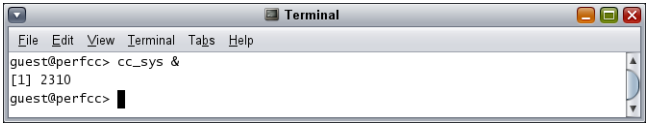
Now let's run prstat with microstate accounting reporting, by issuing prstat -m 2 and recording the output:



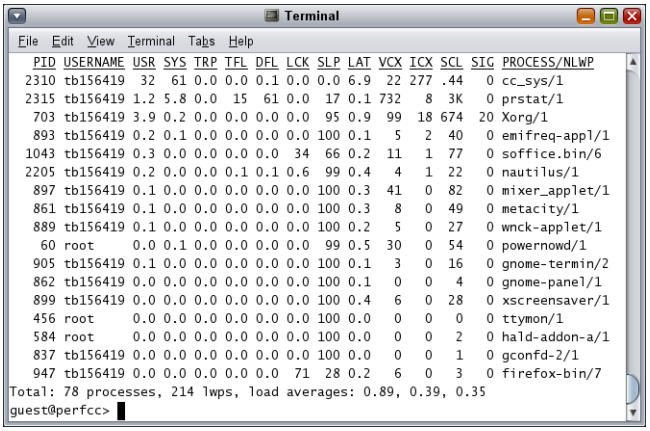
Please observe the top two (2) lines of the output with PID 2223 and PID 2224. One can clearly see that both processes exhibit a high percentage of their time (50% and 52% respectively) in LAT microstate (CPU latency). The remaining time is spent in computation as expected (USR microstate). Clearly in this example, both CPU bound applications are fighting for the one CPU of the test system, resulting in high waiting times (latency) to gain access to a CPU.

`prstat` Usage Scenario – High System Time

Let's run a system call intensive application (cc_sys (`http://blogs.sun.com/partnertech/resource/perfcodecamp/multiplatformbinaries.tar`)) and watch the output of `prstat`. First, start one instance of `cc_sys` :



Then watch the `prstat -m 2` output:



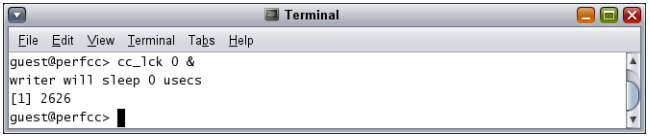
Note the top line of the output with PID 2310. One clearly identifies a high-system time usage (61%) for the process `cc_sys`.

`prstat` Usage Scenario – Excessive Locking

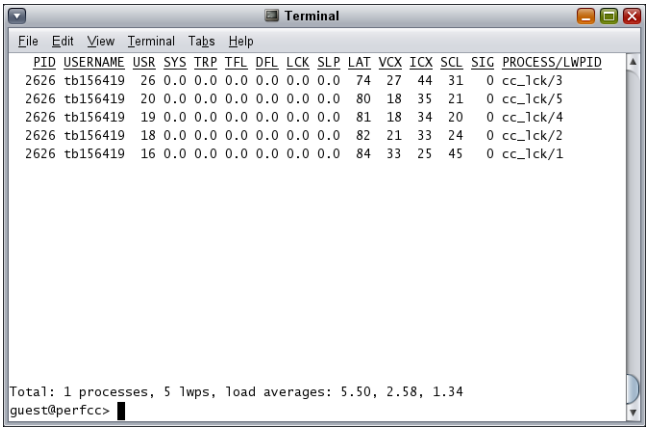
Frequently, poor scaling is observed for applications on multi-processor systems. One possible root cause is badly designed locking inside the application resulting in large time spent waiting for synchronization. The `prstat` column LCK reports on percentage of time spent waiting on user locks.

Let's look at an example with a sample program that implements a locking mechanism for a critical section using reader/writer locks. The programs has four (4) threads looking for access to the shared critical region as readers while one thread accesses the critical section in writer mode. To exhibit the problem, the writer has been slowed down on purpose, so that it spends some time holding the critical section (effectively barring access for the readers).

First start the program such as the writer spends zero (0) microseconds in the critical region (ideal case).

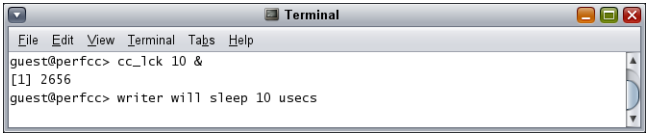


Now let's observe the per-thread microstates. Use `prstat -mL -p 2626 2` for this.

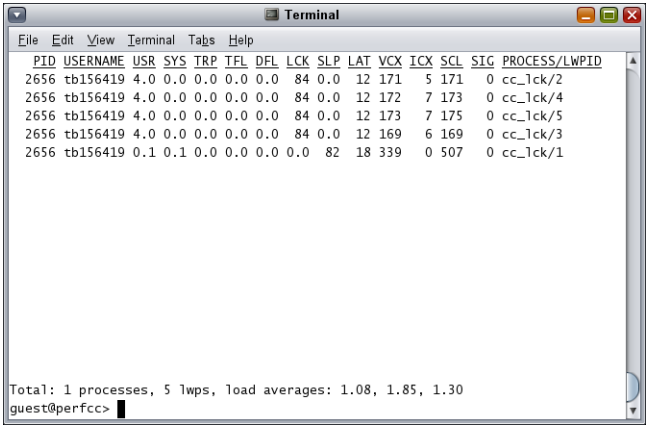


One can observe, that all threads (5) are fighting almost equally for computing resources. Since neither the reader nor the writer hold the critical section for long there is no wait time registered.

Now let's restart the whole test with a writer wait time of ten (10) microseconds.



Again, let's observe the microstates. Use `prstat -mL -p 2656 2` for this.



Now the picture looks different. The four (4) reader threads are spending 84% of their time waiting on the lock for the critical region. The writer (LWP #1) on the other hand, is spending most of its time sleeping (82%).

While in the case of this sample application the locking problems are obvious when looking at the source code, `prstat` microstate accounting capabilities can help pinpoint locking weaknesses in larger applications.

http://blogs.sun.com/partnertech/entry/solaris_performance_primer_process_monitoring

Understanding I/O

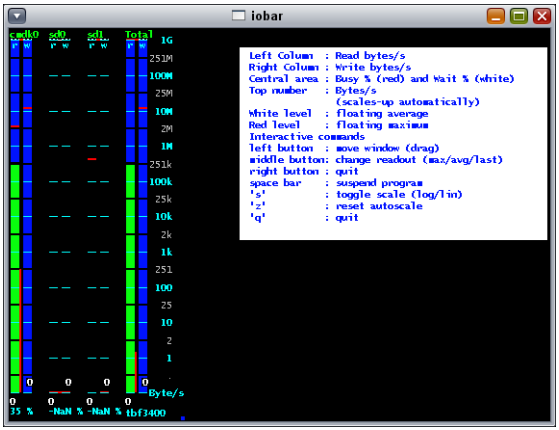
Stefan Schneider and Thomas Bastian, May 1, 2008

Understanding I/O (Input/Output)

This section takes a deeper look at understanding I/O behavior.

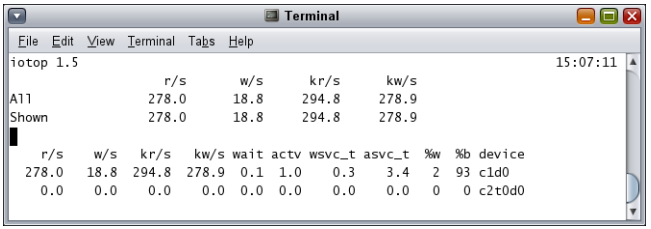
`iobar` [Tools CD] – Display I/O for Disk Devices Graphically

`iobar` displays two bar-graphs for each disk device. The read and write data rate are displayed in green in the left and right areas. The disk utilization is shown in the middle (red). At the bottom of the bars, input and output rates are displayed numerically. The value can be selected between last (green), average (white) and maximum (red) with the mouse middle button. The display mode can be toggled between logarithmic and linear with the left mouse button. In linear mode, scaling is automatic. All values are in bytes per second.



`iotop` [Tools CD] – Display `iostat -x` in a `top`-Like Fashion

`iotop` is a binary that collects I/O statistics for disks, tapes, NFS-mounts, partitions (slices), SVM meta-devices and disk-paths. The display of those statistics can be filtered by device class or using regular expressions. Also the sorting order can be modified.

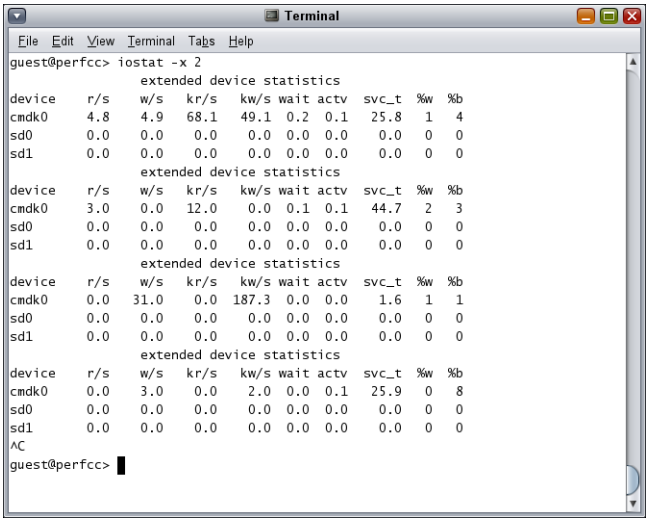


`iostat` – I/O Wizard

If you are looking at understanding I/O behavior on a running system, your first stop will be the command `iostat`(1M),which gives fast answers to questions like:

- How much I/O in terms of input/output operations/second (IOPS) and throughput (MB/second)?
- How busy are my I/O subsystems (latency and utilization)?

In its simplest form, the command `iostat -x <interval>` (for example, `iostat -x 2`) will examine all I/O channels and report statistics. For example, the following illustration shows the output for `iostat -x 2`:



As can be seen in the screen capture, the `iostat -x <interval>` command will report device statistics every <interval> seconds. Every device is reported on a separate line and includes the following information:

- `device`: device name
- `r/s`: device reads per second (read IOPS).
- `w/s`: device writes per second (write IOPS).
- `kr/s`: kilobytes read per second.
- `kw/s`: kilobytes written per second.
- `wait`: average number of transactions waiting for service (queue length)
- `actv`: average number of transactions actively being serviced (removed from the queue but not yet completed). This is the number of I/O operations accepted, but not yet serviced, by the device.
- `svc_t`: average response time of transactions, in milliseconds. The `svc_t` output reports the overall response time, rather than the service time of a device. The overall time includes the time that transactions are in queue and the time that transactions are being serviced.
- `%w`: percent of time there are transactions waiting for service (queue non-empty).
- `%b`: percent of time the disk is busy (transactions in progress).

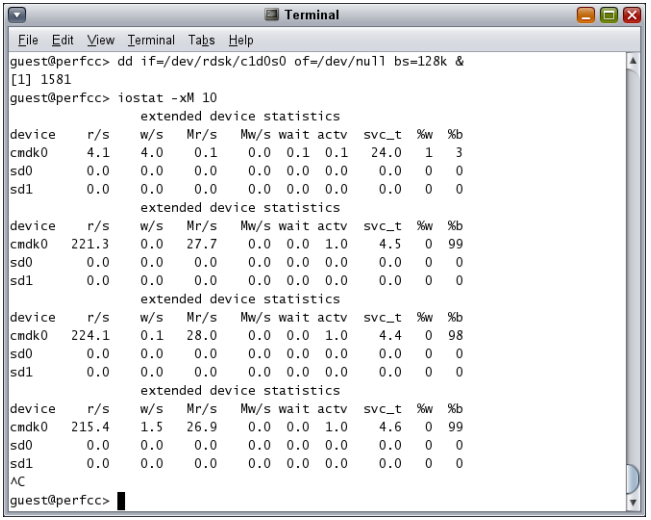
By adding the option `-M` to `iostat`, the report outputs megabytes instead of kilobytes.

`iostat` Usage Scenario – Sequential I/O

Let's study the output of `iostat` when doing sequential I/O on the system. For that, become super-user and in a terminal window, start the command:

```
dd if=/dev/rdsk/c1d0s0 of=/dev/null bs=128k &
```

Then start the `iostat` command with `iostat -xM 10` and watch the output. After a minute stop the `iostat` and `dd` processes.



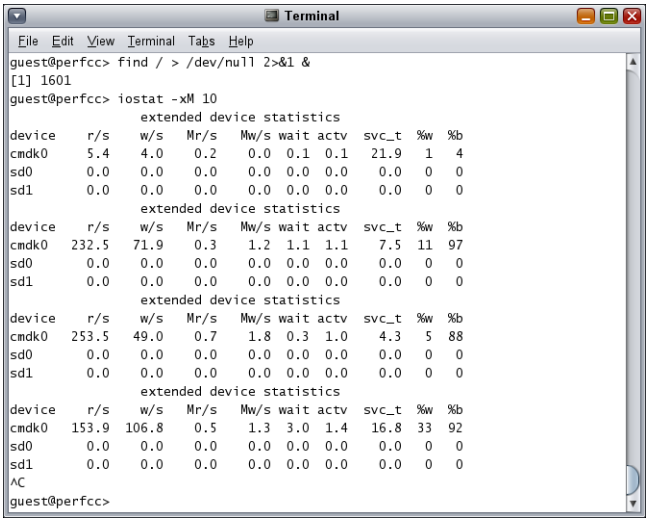
As can be seen in the screen capture, the disk in the test system can sustain a read throughput of just over 25 MB/second, with an average service time below 5 milliseconds.

`iostat` Usage Scenario – Random I/O

Let's study the output of `iostat` when doing random I/O on the system. For that, start the command:

```
find / >/dev/null 2>&1 &
```

Then start the `iostat` command with `iostat -xM 10` and watch the output. After a minute stop the `iostat` and `find` processes.



As can be seen in the screen capture, the same disk in the test system delivers less than 1 MB/second on random I/O.

Properly sizing an I/O subsystem is not a trivial exercise. One has to take into consideration factors like:

- Number of I/O operations per second (IOPS)
- Throughput in Megabytes per second (MB/s)
- Service times (in milliseconds)
- I/O pattern (sequential or random)
- Availability of caching

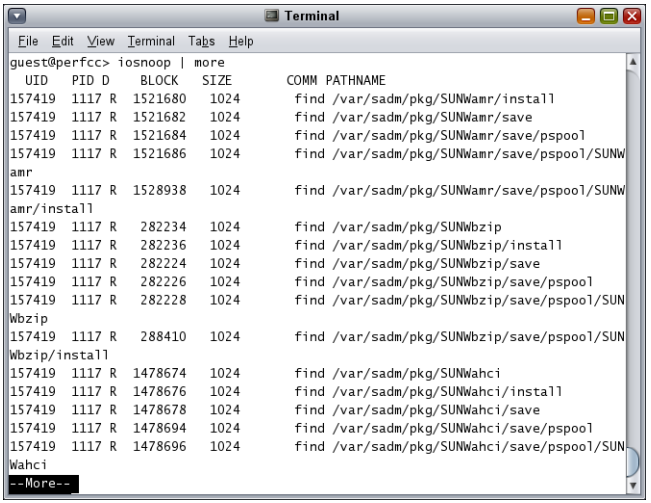
`iosnoop` [DTraceToolkit] – Print Disk I/O Events

iosnoop (`http://opensolaris.org/os/community/dtrace/dtracetoolkit/`) is a program that prints disk I/O events as they happen, with useful details such as UID, PID, filename, command, and so on. `iosnoop` is measuring disk events that have made it past system caches.

Let's study the output of `iosnoop` when doing random I/O on the system. For that, start the command:

```
find / >/dev/null 2>&1 &
```

Then start the `iosnoop` command and watch the output. After a minute stop the `iosnoop` and `find` processes.



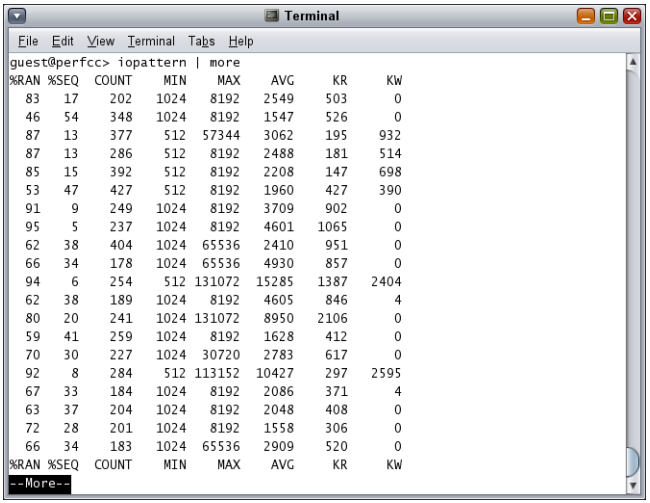
`iopattern` [DTraceToolkit] – Print Disk I/O Pattern

iopattern (`http://opensolaris.org/os/community/dtrace/dtracetoolkit/`) prints details on the I/O access pattern for disks, such as percentage of events that were of a random or sequential nature. By default, totals for all disks are printed.

Let's study the output of `iopattern` when doing random I/O on the system. For that, start the command:

```
find / >/dev/null 2>&1 &
```

Then start the `iopattern` command and watch the output. After a minute, stop the `iopattern` and `find` processes.



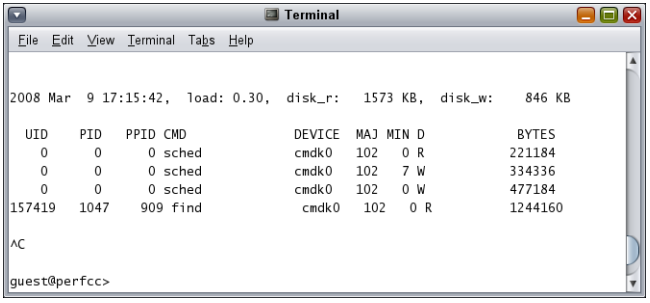
`iotop` [DTrace Toolkit] – Display Top Disk I/O Events by Process

`iotop` prints details on the top I/O events by process.

Let's study the output of `iotop` when doing random I/O on the system. For that, start the command:

```
find / >/dev/null 2>&1 &
```

Then start the `iotop` command and watch the output. After a minute, stop the `iotop` and `find` processes.



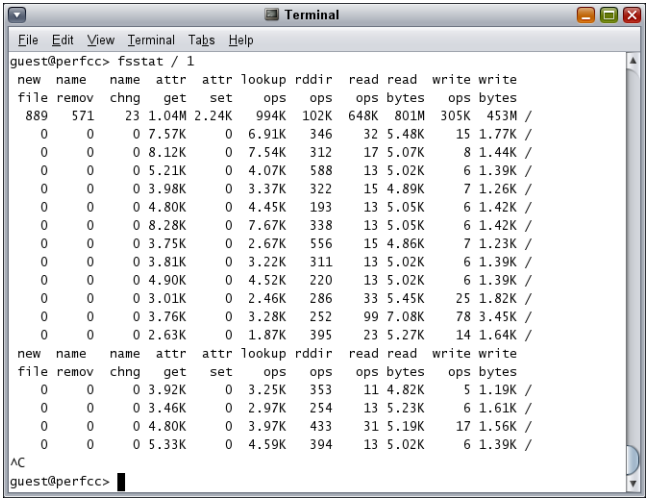
`fsstat` [Solaris 10] – Report File System Statistics

`fsstat`(1M) reports kernel file operation activity by the file system type or by the path name, which is converted to a mount point. Please see the man page `fsstat`(1M) for details on all options.

Let's study the output of `fsstat` when doing random I/O on the system. For that, start the command:

```
find / >/dev/null 2>&1 &
```

Then start the `fsstat / 1` command and watch the output. After a minute, stop the `fsstat` and `find` processes.



http://blogs.sun.com/partnertech/entry/solaris_performance_primer_understanding_io

Understanding the Network

Stefan Schneider and Thomas Bastian, May 2, 2008

Understanding Network Utilization

This section takes a deeper look at network utilization.

`netbar` [Tools CD] – Display Network Traffic Graphically

`netbar` displays two bar graphs for each network interface.

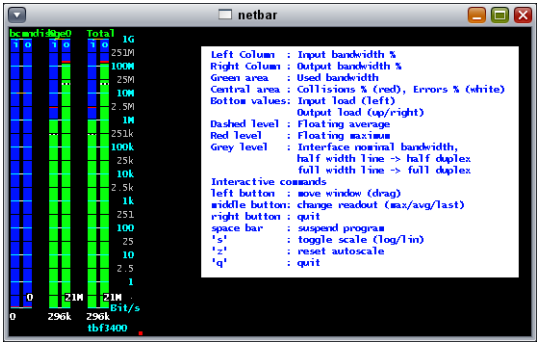
- The left one is the input bandwidth, the right one the output bandwidth.
- The green area shows the used bandwidth and the blue area shows the available one.

On each bar graph, a red marker shows the maximum bandwidth observed during the last period, and a dashed black & white marker shows the average bandwidth during the same period.

At the bottom of the bars, input and output rates are displayed numerically. The value can be selected between last (green), average (white) and maximum (red) with the mouse middle button. Between the bar graphs, a white line displays the error rate while a red line displays the collision rate.

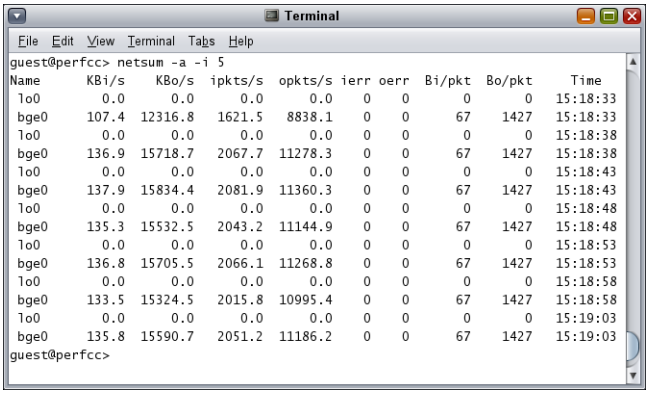
The display mode can be toggled between logarithmic and linear with the left mouse button. In linear mode, scaling is automatic.

A thin white line shows the reported maximum interface speed. If this line spans the whole two bars, the interface is in full-duplex mode, while if the line is limited to the half of the bars, the interface is in half-duplex mode. All values are in bits per second.



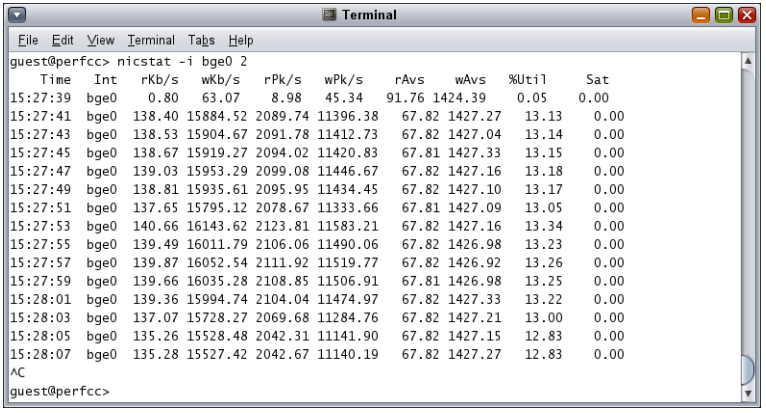
`netsum` [Tools CD] – Displays Network Traffic

`netsum` is a `netstat` like tool, however, its display output is in kilobytes per second, packets per second, errors, collisions, and multicast.



`nicstat` [Tools CD] – Print Statistics for Network Interfaces

`nicstat` prints statistics for the network interfaces such as kilobytes per second read and written, packets per second read and written, average packet size, estimated utilization in percent and interface saturation.



`netstat` – Network Wizard

If you are looking at understanding network behavior on a running system, your first stop may be the command `netstat`(1M), which gives fast answers to questions like:

- How many TCP/IP sockets are open on my system?
- Who communicates with whom? And with what parameters?

The `netstat` command has many options that will satisfy everyone's needs. Please refer to the `netstat`(1M) man page for details.

`netstat` Usage Scenario – List Open Sockets

Often, one will want to look at the list of network sockets on a system. The `netstat` command delivers this type of information for the protocol TCP with the following command:

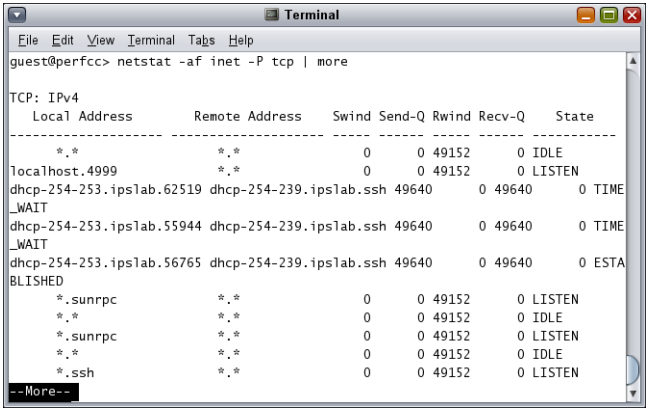
```
netstat -af inet -P tcp
```

If you are interested in the protocol UDP, replace `tcp` with `udp`, for example:

```
netstat -af inet -P udp
```

As an example, let's run the following command and capture the output:

```
netstat -af inet -P tcp
```



The command outputs one line per socket in the system. Included information is:

- *Local Address*: the local socket endpoint with interface and protocol port.
- *Remote Address*: the remote socket endpoint with interface and protocol port.
- *Swind*: sending window size in bytes.
- *Send-Q*: sending queue size in bytes.
- *Rwind*: receiving window size in bytes.
- *Recv-Q*: receiving queue size in bytes.
- *State*: protocol state (for example, LISTEN, IDLE, TIME_WAIT, and so on).

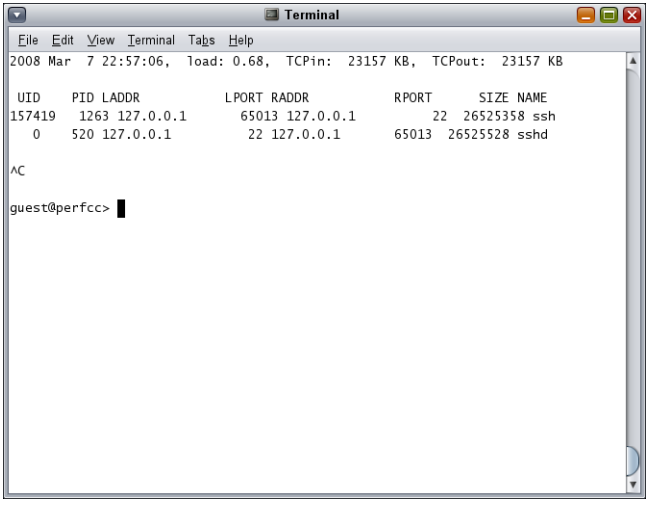
tcptop/tcptop_snv [DTrace Toolkit] – Network "Top"

tcptop (Solaris 10) and tcptop_snv (OpenSolaris) display top TCP network packets by process. To do so, the tool analyses TCP network packets and prints the responsible PID and UID, plus standard details such as IP address and port. The utility can help identify which processes are causing TCP traffic.

You can start the tool with the command tcptop on a Solaris 10 system or tcptop_snv on an OpenSolaris system. Let's study the output of tcptop_snv. For that, start tcptop_snv in one window and in another one, generate some network traffic with the command:

```
scp /kernel/genunix localhost:/tmp
```

The output should be similar to this example:



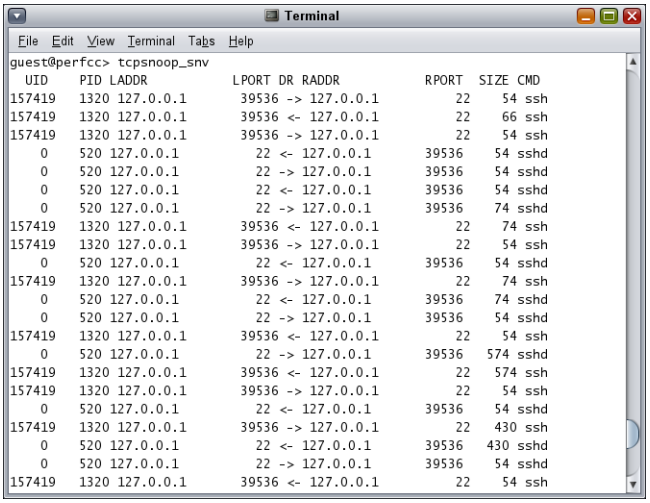
tcpsnoop/tcpsnoop_snv [DTrace Toolkit] – Network Snooping

tcpsnoop (Solaris 10) and tcpsnoop_snv (OpenSolaris) snoop TCP network packets by process. The tool operates in a similar way to tcptop and tcptop_snv, except that information is displayed continuously.

You can start the tool with the command tcpsnoop on a Solaris 10 system or tcpsnoop_snv on an OpenSolaris system. Let's study the output of tcpsnoop_snv. For that, start tcpsnoop_snv in one window and in another one, generate some network traffic with the command:

```
scp /kernel/genunix localhost:/tmp
```

The output should be similar to this example:



`nfsstat` – NFS statistics

`nfsstat`(1M) displays statistical information about the NFS and RPC (Remote Procedure Call) interfaces to the kernel. It can be used to view client and/or server side statistics broken down by NFS version (2, 3 or 4).

`http://blogs.sun.com/partnertech/entry/solaris_performance_primer_understanding_the`

Tracing Running Applications

Stefan Schneider and Thomas Bastian, May 5, 2008

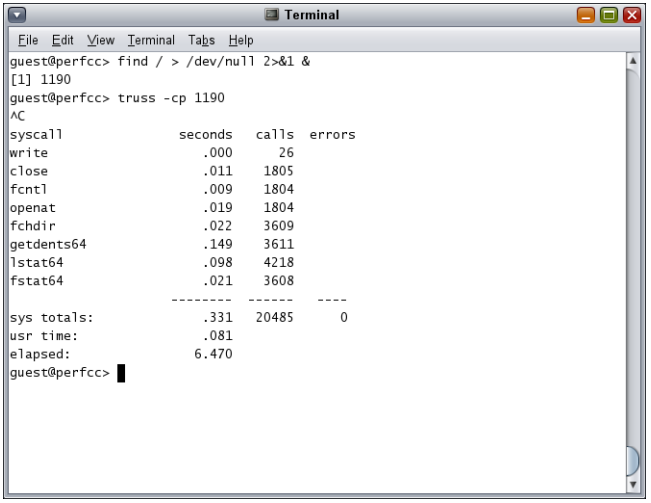
This section introduces further tools and techniques for tracing at large.

`truss` – First Stop Tool

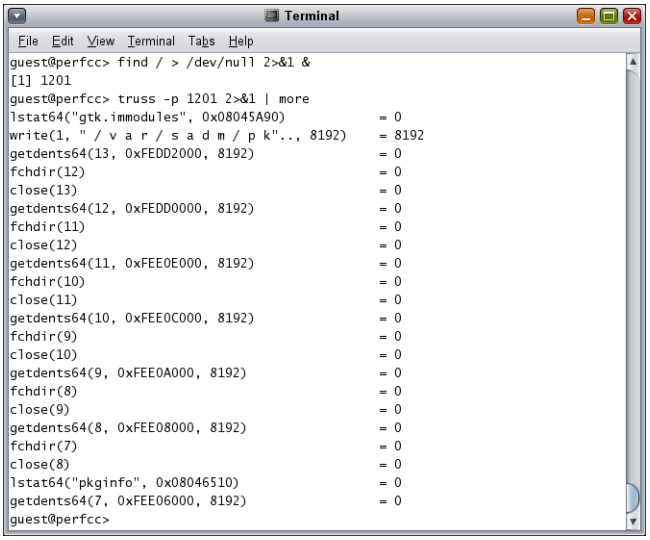
`truss`(1) is one of the most valuable tools in the Solaris OS for understanding various issues with applications. `truss` can help you understand which files are read and written, which system calls are called and much more. Although, `truss` is very useful, one has to understand that it is also quite intrusive on the applications traced and can therefore influence performance and timing.

A standard usage scenario for `truss` is to get a summary of system call activity of a process over a given window of time.

As can be seen in the following output, the `find` process issues large amounts of `fstat`(2), `lstat`(2), `getdents`(2) and `fchdir`(2) system calls. The `getdents`(2) system call consumes roughly 45% of the total system time (0.149 seconds of 0.331 seconds of total system time).



Another standard usage scenario of `truss` is to get a detailed view of the system calls issued by a given process.



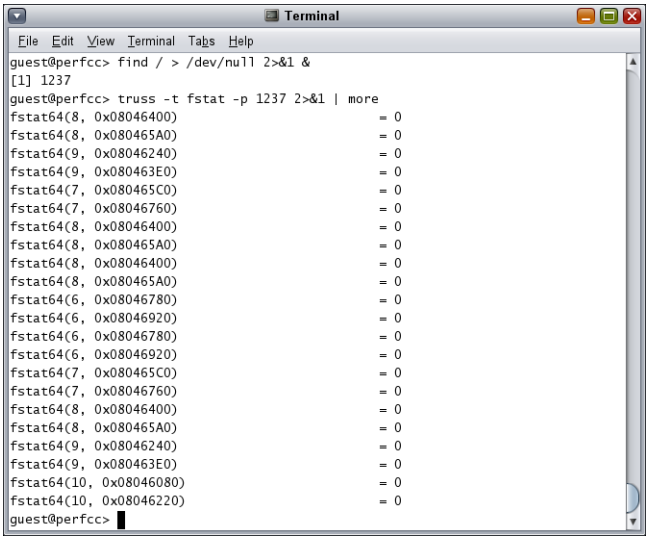
The output shows `truss` giving out details about the system calls issued and their parameters. Further details can be obtained with the `-v` option. For example:

```
truss -v all -p <pid>
```

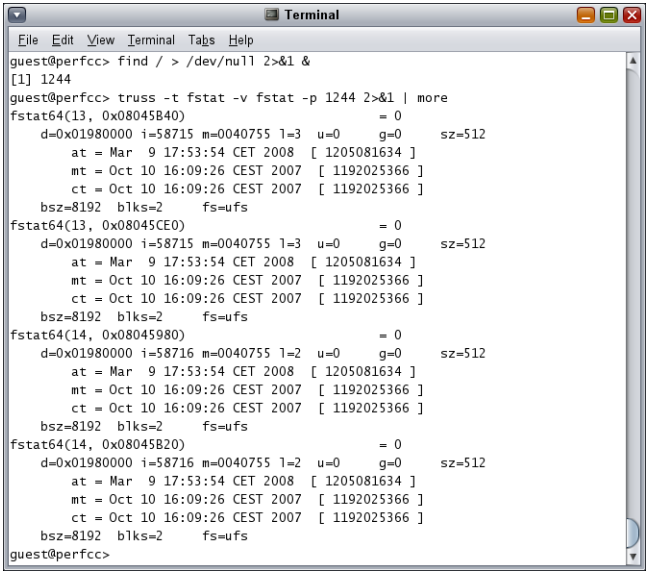
Yet another standard usage scenario is to restrict the output of `truss` to certain system calls.

```
truss -t fstat -p <pid>
```

would limit the output to `fstat`(2) system call activity.



Finally, combining the `-t` option with `-v`, one gets an output like this:



`plockstat` – Report User-Level Lock Statistics

The `plockstat`(1M) utility gathers and displays user-level locking statistics. By default, `plockstat` monitors all lock contention events, gathers frequency and timing data about those

events, and displays the data in decreasing frequency order, so that the most common events appear first. `plockstat` gathers data until the specified command completes or the process specified with the `-p` option completes. `plockstat` relies on DTrace to instrument a running process or a command it invokes to trace events of interest. *This imposes a small but measurable performance overhead on the processes being observed*. Users must have the `dtrace_proc` privilege and have permission to observe a particular process with `plockstat`.

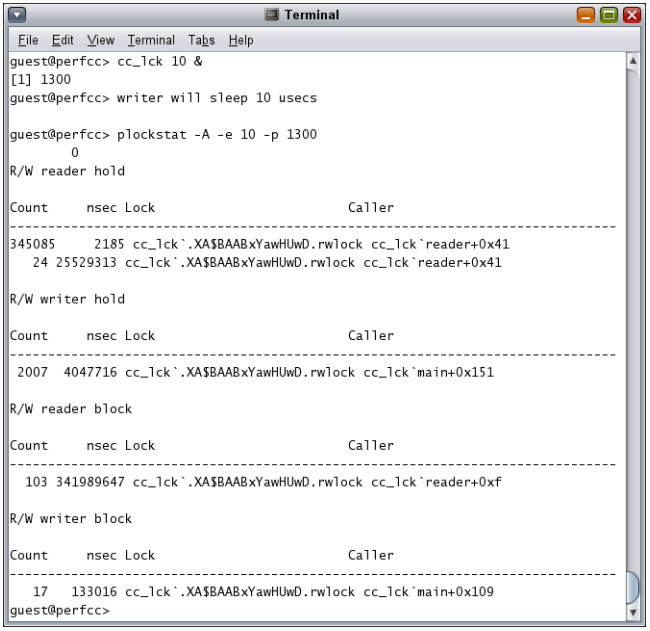
Let's study the output of `plockstat` by running our sample reader/writer locking program `cc_lck`. First start `cc_lck` with the writer blocking for ten microseconds:

```
cc_lck 10
```

Then run the `plockstat` tool for ten seconds:

```
plockstat -A -e 10 -p <pid>
```

The output should be similar to the following output. From the output, one can observe some contention on the reader/writer lock.



`pfilestat` [DTraceToolkit] – Trace Time Spent in I/O

`pfilestat` prints I/O statistics for each file descriptor within a process. In particular, the time break down during `read`() and `write`() events is measured. This tool helps in understanding the impact of I/O on the process.

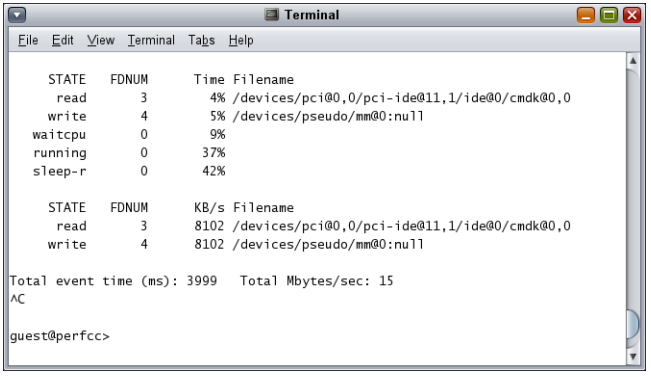
To study the output of `pfilestat`, let's start the following command as root:

```
dd if=/dev/rdsk/c1d0s0 of=/dev/null bs=1k &
```

Then in another window, let's start the `pfilestat` tool with the `pid` of the `dd` command as an argument:

```
pfilestat <pid of dd command>
```

The output should be similar to this example:



The `pfilestat` output breaks down the process time by the percentage spent for reading (`read`), writing (`write`), waiting for CPU (`waitcpu`), running on CPU (`running`), sleeping on read (`sleep-r`) and sleeping on write (`sleep-w`).

`cputrack/cpustat` – Monitor Process or System With CPU Performance Counters

The `cputrack`(1) utility allows CPU performance counters to be used to monitor the behavior of a process or family of processes running on the system. The `cpustat`(1M) utility allows CPU performance counters to be used to monitor the overall behavior of the CPUs in the system.

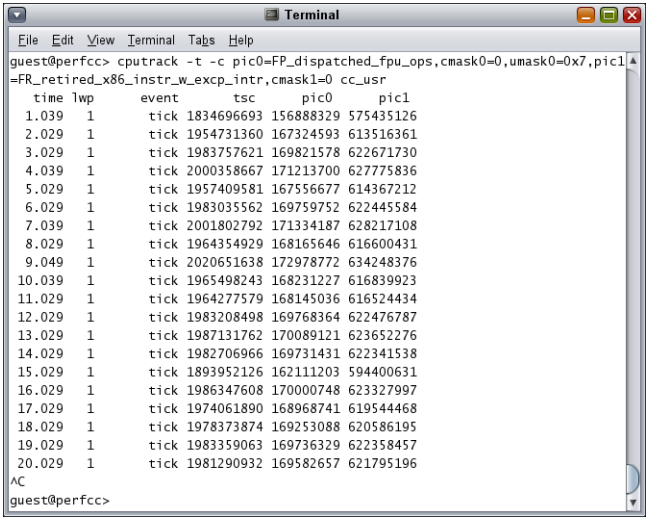
Using `cputrack/cpustat` requires intimate knowledge of the CPU and system under observation. Please consult the system/CPU documentation for details on the counters available. `cpustat` or `cputrack` with the `-h` option will list all available performance counters.

To observe the output of `cputrack`, let's run the tool with our sample program `cc_usr`.

Use the following system-specific command (all on one line):

```
cputrack -t -c pic0=FP_dispatched_fpu_ops,cmask0=0,umask0=0x7, \
pic1=FR_retired_x86_instr_w_excp_intr,cmask1=0 cc_usr
```

The output should look like this:



In the output, one can see that the `cc_usr` program executed roughly 600 million instructions per second with roughly 160 million floating-point operations per second.

http://blogs.sun.com/partnertech/entry/solaris_performance_primer_tracing_at

◆ ◆ ◆ CHAPTER 5

Developer Tools

This chapter covers some of the other tools that are available to provide insight into application performance. It covers topics such as the SHADE library that can be used to emulate the run of an application on a SPARC processor, as well as how to add DTrace probes into userland applications. The chapter also covers some of the Sun Studio tools like the Performance Analyzer and the probe effect of gathering profile information.

Cool Tools: Using SHADE to Trace Program Execution

Darryl Gove, September 29, 2006

Introduction

The SHADE library is an emulator for SPARC hardware. An emulator is software which "pretends" to be a processor so that the application runs on the emulator, which then runs on the real hardware. Emulators are often used in situations where the original hardware is no longer available, or where the application needs to run on new hardware that has a different processor to the current hardware, or in cases where the target hardware is different to the development platform.

The particular advantage of using SHADE is that it is possible to write an analysis tool which gathers information from the application being emulated. The SHADE library comes with some example analysis tools which track things like the number of instructions executed or the frequency that each type of instruction is executed. A more advanced analysis tool might look at cache misses that the application encounters for a given cache structure.

Obtaining SHADE

The SHADE tools (for Solaris OS releases 9 and 10 on SPARC processors) are available from the Cool Tools site (`http://cooltools.sunsource.net/shade/index.html`). The file is downloaded as a compressed tar file, and when it is decompressed it will have a directory structure as follows:

```
shade-32/bin
shade-32/doc
shade-32/eg
shade-32/inc
shade-32/lib
shade-32/man
```

The include files are located in the `inc` directory, and the library files are located in the `lib` directory. There are a number of examples located in the `eg` directory, and documentation is provided in the `doc` directory.

For the purposes of this article it is assumed that the SHADE code has been installed in `/export/home/shade-32/`.

SHADE Architecture

SHADE works by emulating a number of instructions from the target application, and recording each instruction in a buffer -- this set of recorded instructions is often referred to as a "trace" (of the run of the application). The SHADE library does all the work of emulating the application. Once it has gathered a trace of instructions, it hands this trace over to the "analyzer." The analyzer has to be written in order to take the trace generated by the SHADE library and analyze it.

SHADE ships with the source code to several analyzers in the `eg` subdirectory. The simplest analyzer provides an instruction count; a more complex one is a cache simulator. At the heart of a SHADE analyzer is a simple loop that iterates over records from a trace of the application. This loop takes each individual record and does what ever processing is necessary.

Internally, SHADE works by splitting the application into short snippets of code. These snippets do two things: first of all they do the same thing as the original application would have done. The second thing that they do is that they record, into the trace record, what happened as they executed the code.

Writing a SHADE Analyzer

There are a number of routines which have to be provided when writing a SHADE analyzer:

- The `shadeuser_initialize` function gets called before the program getting traced is loaded. This gives the analyzer the opportunity to select which instructions are to be traced, and also to set up any necessary data structures.
- The `shadeuser_analyze` function gets called repeatedly until the application being traced ends. Each time this function is called, it should ask SHADE to get the next set of traced instructions.
- The `shadeuser_report` routine is called after the application being traced has exited, or when a signal requesting a report is received.
- The `shadeuser_terminate` routine is called after the application being traced has terminated and allows the analyzer to perform any necessary clean up.
- The `shadeuser_analusage` routine needs to return usage information for the analyzer.
- The `shadeuser_analversion` character string needs to be defined to identify the analyzer.

The Basics of Writing an Analyzer

The two functions that need to be provided that are not related to the tracing of the application are to set up the name of the analyzer by assigning a value to the string `shadeuser_analversion` and to write a routine (`shadeuser_analversion`) which outputs usage information for the analyzer. In this case we will call the analyzer `trace` and there are no user-configurable parameters.

```
const char shadeuser_analversion[] = "trace ";

const shade_options_t shadeuser_opts[]={
 {0, "", NULL, NO_PARAM}
};

void shadeuser_analusage(void)
{
 shade_print_opt_info(shadeuser_opts, 0);
}
```

Selecting What to Trace

There are two steps to defining what data is to be collected.

The first step is to define the trace record type that is to be used to store the information. In this case we are using the `SHADE_SPARCV9_TRACE` record type. This type is defined in the `shade_sparcv9.h` header file and includes the `l` record which holds a text representation of the instruction which will be used when the trace is printed out. There are other trace records

defined in the shade_sparcv9.h header file which can hold information about the values held in the source and destination registers. There is a simpler trace record called SHADE_TRACE that has space for trace information such as the PC of the instruction, or the Effective Address (EA) of a load or store instruction.

```
/*
 * Define a standard trace record.
 */
struct shade_trace_s {SHADE_SPARCV9_TRACE};
```

The second step is to set up SHADE so that it delivers just the information that is required by the analyzer.

```
int shadeuser_initialize(
 int argc,
 char ** argv,
 char ** envp)
{
 shade_iset_t * piset;
 shade_tset_t * ptset;


 /*
 * We want to be notified of fork() operations, so we know if the
 * application spawns a child process.
 */
 shade_setopt(SHADE_OPT_FORKNOTIFY);

 /*
 * Set up the trace control parameters
 */
 shade_trsize(sizeof(shade_trace_t));

 /*Trace all instructions*/
 piset = shade_iset_newclass(SHADE_ICLASS_ANY, -1);

 /*Information needed*/
 /*Instruction details and the PC*/
 ptset = shade_tset_new(SHADE_SPARCV9_TRCTL_I, SHADE_TRCTL_PC, -1);

 shade_trctl(piset, SHADE_TRI_ISSUED, ptset);
 return(0);
}
```

The call shade_setopt determines how the SHADE library should handle forks and execs. With the SHADE_OPT_FORKNOTIFY setting, the SHADE analyzer will be notified when the program being traced forks, and SHADE will also fork a new process to trace the process forked by the program being traced.

It is necessary to tell SHADE the size of the trace record using the shade_trsize call. The variable shade_trace_t is equivalent to the previously declared struct shade_trace_s.

The next step is to select which instructions are to be traced, and what is to be recorded about those instructions.

First of all, the routine `shade_iset_newclass` is called to select the set of instructions to be traced. This routine takes a list of the instruction types to trace. The list is terminated by the value -1. For the purposes of this program, it is necessary to select all instructions using the `SHADE_ICLASS_ANY` specifier. However a different analyzer might chose to just trace loads and stores, or other subsets of all instructions.

The routine `shade_tset_new` is used to select the data that will be recorded in the trace records. Again the routine takes a list of all the data to be recorded, and terminates the list with the value -1. For the purposes of tracing the executed instructions it is necessary to record the instruction that was executed (`SHADE_SPARCV9_TRCTL_I`) and the address of that instruction (`SHADE_TRCTL_PC`). Using other specifiers, it is possible to record other data, such as the effective address of memory operations.

Finally the `shade_trctl` interface is used to inform SHADE of the decisions. This interface also requires a specifier to determine whether instructions should be traced only when they are executed, only when they are annulled, or when they are either executed or annulled. The specifier `SHADE_TRI_ISSUED` tells SHADE to trace instructions both when they are executed and when they are annulled.

Analyzing Program Execution

The routine `shadeuser_analyze` gets called repeatedly by the SHADE library to fetch more instructions from the running application. This routine needs to call the `shade_run` routine to get a fresh set of trace records, and then should process those records. The simplest format for this routine would be to just count the total number of instructions executed, as shown in this example:

```
int shadeuser_analyze(void)
{
 shade_trace_t atr[5*1024];
 int ntr;

 ntr = shade_run(atr, sizeof(atr)/sizeof(*atr));
 total+=ntr;
}
```

The routine declares a local array of records which is filled by the call to the SHADE routine `share_run`. This routine returns the number of records placed in the array as the result of the call.

It is actually necessary to have a bit more complexity than this to handle the forking of the traced application. When this happens the variable total will get copied into the new environment, so it is necessary to reset this variable in the new process but not the parent. An example of doing this can be found in the example `icount.c` which is shipped with SHADE.

The objective for the analyzer being written here is to print out the traced instructions. There are routines available in the SPIX library (which ships as part of SHADE) to convert the information recorded in the trace records into disassembly.

```
int shadeuser_analyze(void)
{
 shade_trace_t atr[5*1024];
 shade_status_t status;
 int ntr;
 int i;
 char buf[256];

 ntr = shade_run(atr, sizeof(atr)/sizeof(*atr));

 if (ntr == 0)
 return 0;



 for (i = 0; i < ntr; i++)
 {
   if (spix_sparc_dis32(buf, 256,
       spix_sparc_iop(SPIX_SPARC_V9,&atr[i].tr_i),
       &atr[i].tr_i, atr[i].tr_pc) < 0)
   {
     printf("%llx %-25s\n",(long long)atr[i].tr_pc, "???");
   }
   else
   {
     printf("%llx %-25s\n",(long long)atr[i].tr_pc, buf);
   }
 }
 return(0);
 }
```

The routine `spix_sparc_iop` returns opcode for the given assembly language instruction. This is fed into the routine `spix_sparc_dis32` together with assembly language instruction and its address. This routine writes the disassembly of the instruction into a buffer. If the instruction is successfully disassembled, it is printed out.

Cleaning Up at the End of the Run

The routine `shadeuser_report` is called either when all the traced applications have completed running, or when a signal is received by SHADE. This routine should produce a report of the results so far. In this particular case there is no report to produce as the output is produced during the run. Clean up after the application terminates is done in the `shadeuser_terminate` routine. The return value from this routine is used as the return value from the SHADE analyzer.

Building the Trace Tool

The complete source code for the example appears at the end of this article. The build instructions are as follows:

```
$ cc -I/export/home/shade-32/inc -xO2 -DSPIX_ADDR=64 -o trace \
 -Wl,-M,/export/home/shade-32/lib/mapfile trace.c \
 /export/home/shade-32/lib/libshade.a \
 /export/home/shade-32/lib/libspix_sparc.a
```

Output From the Trace Tool

Here is the kind of output that this analyzer produces:

```
% trace -- ls

4035190c ba,a 0x40351918
40351910 ba,a 0x4035191c
40351918 or %g0, %g0, %o0
4035191c save %sp, -0xa0, %sp
40351920 call 0x40351928
40351924 sethi %hi(0x38c00), %l7
40351928 or %l7, 0x28c, %l7
4035192c addcc %i0, %g0, %o0
40351930 bne 0x4035198c
```

There is a weakness in this analyzer, as written. For performance reasons, the analyzer gets a block of records from the SHADE library. If the application fails, then SHADE may not return all the instructions up until the failing instruction. If it is important to return the trace up until the point of failure, then the analyzer can be modified to return just one instruction at a time.

Concluding Remarks

Tracing the execution of a program is a relatively simple task, but it does demonstrate the fact that we can see the exact sequence of instructions being executed. It is also possible to retrieve information from the SHADE library about the effective address of a load or store instruction, or even what data is fetched from or stored to memory.

Here is the complete example:

```
#define SPIX_ADDR 64

#include <stdio.h>
#include <stdlib.h>
#include <assert.h>
#include <string.h>
#include <unistd.h>
#include <spix_sparc.h>
#include <shade.h>
#include <shade_sparcv9.h>
#include <shade_anal.h>
#include "getopt.h"
#include <sun_prefetch.h>

/*
```

```
 * Define a standard trace record.
 */
struct shade_trace_s {SHADE_SPARCV9_TRACE};


const char shadeuser_analversion[] = "trace ";


const shade_options_t shadeuser_opts[]={
 {0, "", NULL, NO_PARAM}
};

int shadeuser_initialize(
 int argc,
 char ** argv,
 char ** envp)
{
 shade_iset_t * piset;
 shade_tset_t * ptset;


 /*
 * We want to be notified of fork() operations, so we know if the
 * application spawns a child process.
 */
 shade_setopt(SHADE_OPT_FORKNOTIFY);

 /*
 * Set up the trace control parameters
 */
 shade_trsize(sizeof(shade_trace_t));
 /*Trace all instructions*/
 piset = shade_iset_newclass(SHADE_ICLASS_ANY, -1);

 /*Information needed*/
 /*Instruction details and the PC*/
 ptset = shade_tset_new(SHADE_SPARCV9_TRCTL_I, SHADE_TRCTL_PC, -1);

 shade_trctl(piset, SHADE_TRI_ISSUED, ptset);
 return(0);
}

/*
** Run and trace an application that is loaded into Shade.
*/
int shadeuser_analyze(void)
{
 shade_trace_t atr[5*1024];
 shade_status_t status;
 int ntr;
 int i;
 int val;
 char buf[256];

 memset(atr,0,sizeof(atr));
 status = shade_appstatus(&val);

 switch (status) {
 case SHADE_STATUS_LOADED:
```

```
 break;
 case SHADE_STATUS_FORKED:
 break;
 }

 /*
 * Count all instructions of each opcode.
 */


 ntr = shade_run(atr, sizeof(atr)/sizeof(*atr));

 if (ntr == 0)
 return 0;



 for (i = 0; i < ntr; i++)
 {
   if (spix_sparc_dis32(buf, 256,
       spix_sparc_iop(SPIX_SPARC_V9,&atr[i].tr_i),
       &atr[i].tr_i, atr[i].tr_pc) < 0)
   {
     printf("%llx %-25s\n",(long long)atr[i].tr_pc, "???");
   }
   else
   {
     printf("%llx %-25s\n",(long long)atr[i].tr_pc, buf);
   }
 }
 return(0);
 }

void
shadeuser_report(int reason, void* ptr) {
 switch (reason) {
 case SHADEUSER_REPORT_TERMINATE:
 break;
 case SHADEUSER_REPORT_SIGNAL:
 case SHADEUSER_REPORT_APP_DONE:
 break;
 }
}

/*
** Perform termination activities after all applications have been executed.
*/

int shadeuser_terminate(
 int ret)
{
 int val;

 /*
 * Exit with the termination status from the executed application.
 */
 switch (shade_appstatus(&val))
 {
 case SHADE_STATUS_EXITED:
```

```
 return(val);

 case SHADE_STATUS_SIGNALED:
 return(128 + val);

 case SHADE_STATUS_NOAPP:
 return(0);

 case SHADE_STATUS_LOADED:
 case SHADE_STATUS_RUNNING:
 default:
 assert(0);
 }

}



void shadeuser_analusage(void)
{
 shade_print_opt_info(shadeuser_opts, 0);
}
```

http://developers.sun.com/solaris/articles/shade.html

Performance Analysis Made Simple Using SPOT

Darryl Gove, March 7, 2006

Introduction

Performance analysis is the process of looking at how fast an application runs, determining why it is running as fast as it does, and then finding ways to improve the performance. A further complication is the problem of knowing when performance cannot be improved any further.

An application's performance depends on a combination of hardware and software factors. For example, what events must the hardware deal with, and what degree of optimisation was applied when compiling the application? There are a number of tools that can be used to extract information or collect this kind of information, but knowing which tools to pick for a given application can be tricky.

This paper introduces a new tool that aims to simplify the process of performance analysis. We call it the Simple Performance Optimisation Tool – or SPOT. SPOT is an add-on package to Sun Studio. It is available for both x86 and SPARC based systems. SPOT has been released as part of the Cool Tools (http://opensparc.sunsource.net/cooltools) project. Further information on the tool and how to obtain it is available from http://cooltools.sunsource.net/spot/.

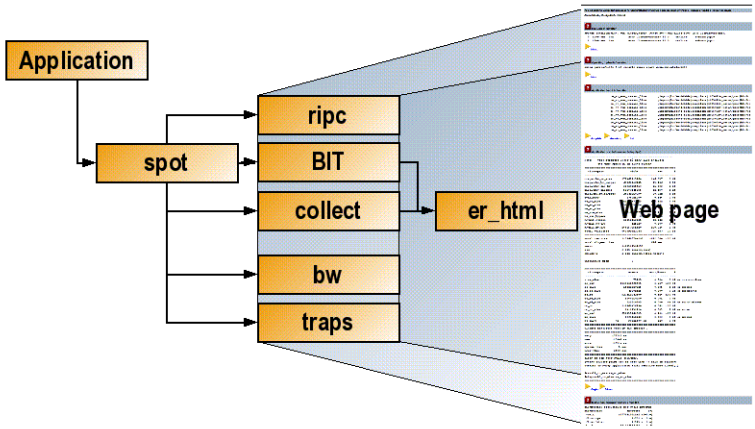
What Is SPOT?

The idea behind SPOT is to make it as simple as possible to produce a report on the performance of a given application. There are two ways of using SPOT to do this. The way that produces the most data is for applications that can be run multiple times. In this case SPOT can run the application under a comprehensive set of tools and collect data for each tool over the entire run of the application. Alternatively, some applications run continuously, and in this case SPOT can attach to the running application and use each tool for a short period of time to gather the data.

One very useful feature of SPOT is that it generates a report on the application's performance as a set of hyperlinked web pages. This addresses a couple of hard-to-solve problems. First of all, the spot-report archives a snapshot of the performance of the application under a given workload, complete with build flags and source code for the parts of the application where the time is spent (assuming that this information is available). Secondly, because the spot-report is a hypertext document, it is possible to pinpoint a particular place in the code with a URL. This feature is very helpful for either working on remote machines, or for collaborating with remote colleagues.

The architecture of SPOT is shown in the following figure.

FIGURE 5–1 Architecture of SPOT



SPOT uses a number of tools, and each tool generates a part of the report. The major tools SPOT uses are:

- The `collect` command from the Sun Studio Performance Analyzer. This command can gather either a profile of where the time is spent in an application, or where processor events occur in the application.

- The command `ripc` gathers information about how much time is spent in processor stalls, and reports this information by the type of stall that causes the problem. For example, it will report the number of seconds spent waiting for data that is located in memory. (The counter for this is named `Re_EC_miss`. The counter names are those that are found in the User's Guide for the processor (`http://www.sun.com/processors/documentation.html`)). This command can also select counters that are worth profiling using `collect` to determine exactly where in the program the particular events are happening. The tool `ripc` will also generate a set of graphs if it finds `gnuplot` to be available on the user's path.
- The BIT tool instruments the application (so long as the application has been compiled appropriately) and counts the number of times each routine and instruction is executed. Using this information it is possible to derive things like the average trip count for a particular loop, or how frequently a condition is true or false.
- The output from the `collect` command and from BIT is fed through another tool called `er_html`. `er_html` uses `er_print` from the Performance Analyzer and renders the data as a set of hyperlinked web pages. These pages show the source (if the application has been compiled with debug) and disassembly for the parts of the code where the time is spent.
- The tool `bw` reports system-wide bandwidth utilization over the entire run of the application. The tool `traps` reports system-wide trap statistics. These tools need root privileges to run. Both of these tools will also generate graphs of the results if `gnuplot` is found on the path.

Running SPOT

SPOT is installed on top of Sun Studio., If Sun Studio is installed in the default place of `/opt/SUNWspro`, then SPOT will be located in `/opt/SUNWspro/extra/bin`.

Note – The Sun Studio Express releases include SPOT. For these releases, the tool can be found in the `bin` directory of the Sun Studio tree.

SPOT can be invoked in one of two ways:

```
/opt/SUNWspro/extra/bin/spot <application> <parameters>
```

will run the target application multiple times, each time under a new tool.

```
/opt/SUNWspro/extra/bin/spot -P <pid>
```

will attach to a running process and rotate through each tool for a short period of time. In both cases the report will be broadly the same.

By default SPOT will produce a short report on the performance of the application, but if SPOT is started with the `-X` flag it will attempt to provide a more detailed set of data (if it has sufficient permissions).

Compiling the Application to Get the Most Data From SPOT

To get the most benefit from SPOT, the application needs to be compiled with the compiler in at least Sun Studio 11. The following flags are recommended:

- Enable debug information using the flag `-g` for C or Fortran or the flag `-g0` for C++. This will enable SPOT to report time attributed to lines of source, not just assembly language instructions.
- Allow SPOT to gather instruction and function count data using BIT by ensuring that the application is compiled with `-O`, `-fast`, or at least `-xO1`, and the `-xbinopt=prepare` flag. (In the Sun Studio Express releases, this flag is unnecessary because it is enabled by default.) The effect of these flags is to record sufficient information in the binary for the tool BIT to be able to instrument it.

An Example

The code test.c (`http://developers.sun.com/solaris/articles/img/spot/test.c`) demonstrates some of the key features of the spot-report. This test code has three separate stages which do the following things:

- Consume memory bandwidth.
- Cause cache misses.
- Cause TLB misses.

For the purposes of being an effective demonstration of the tool, the code should be compiled with the minimum amount of optimization, and the appropriate compiler flags:

```
cc -g -O -xbinopt=prepare -o test test.c
```

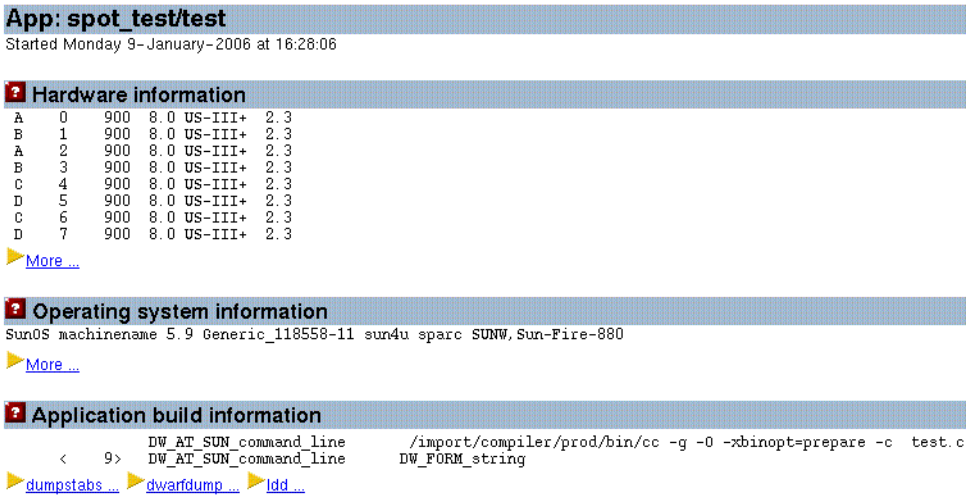
Then it can be run under SPOT using the following command:

```
spot -X test
```

Recording System and Build Information

The first thing that SPOT does is to record information on the test bed that was used to gather the data, and also on the flags that were used to build the application. This is shown in the following figure.

FIGURE 5–2 SPOT System and Build Information



In this particular case the application was built as shown above, and then run on a Sun Fire v880 with eight 900 Mhz UltraSPARC-III+ processors.

Hardware Performance Counter Information

The tool `ripc` then runs the application and gathers data on the relevant events recorded by the performance counters on the processor. The information gathered from the performance counters can be used to guide the process of selecting appropriate compiler options, as suggested in this paper (`http://developers.sun.com/solaris/articles/pcounters.html`). The information from `ripc` is presented as a table, as shown in the next figure.

FIGURE 5–3 `ripc` Performance Counter Information

Application stall information (using ripc)

```
------------------------------------------------------------
NOTE:   Time reported under D$ miss also includes
        the time spent in L2 cache misses
============================================================
  UltraSparc                ticks            sec       %
============================================================
Dispatch0_IC_miss         313828263          0.346    0.3%
Dispatch0_br_target       709780530          0.784    0.7%
Dispatch0_2nd_br           25740897          0.028    0.0%
Dispatch0_mispred          72320941          0.080    0.1%
Dispatch_rs_mispred         3819168          0.004    0.0%
DTLB_miss               14646755195         16.168   13.7%
Re_DC_miss              87805280500         96.926   82.1%
Re_EC_miss              62914203978         69.450   58.9%
Re_PC_miss                 14706602          0.016    0.0%
Re_RAW_miss                15608752          0.017    0.0%
Re_FPU_bypass                 1446          0.000    0.0%
Rstall_storeQ             399062307          0.441    0.4%
Rstall_FP_use                67141          0.000    0.0%
Rstall_IU_use             390244681          0.431    0.4%
Total Stalltime         104397225984        115.242   97.7%
------------------------------------------------------------
Total CPU Time          106895974400        118.000  100.0%
Total Elapsed Time                          120 Sec
Instr                    10621814784
IPC                    0.099 (instr/time)
Grouping               4.251 (instr/(time-total))
------------------------------------------------------------
unfinished fpop              0

============================================================
  UltraSparc               events       evnt/instr     %
============================================================
ITLB_miss                    1281          0.000    0.0% of Instructions
IC_ref                 9095283887          0.856  100.0%
IC_miss                  33497640          0.003    0.4% of IC Ref
EC_ic_miss                  40274          0.000    0.1% of IC misses
DC_rd                  2760498310          0.260  100.0%
DC_rd_miss              667913240          0.063   24.2%
EC_rd_miss              341625055          0.032   51.1% of DC rd misses
DC_wr                  1497628547          0.141  100.0%
DC_wr_miss             1491250098          0.140   99.6% of DC wr
EC_ref                 3289442267          0.310  100.0%
EC_miss                 733592855          0.069   22.3% of Ref
FP_Inst           A=   1053446898 M=       1579    9.9%
============================================================
Maximum Resources Used By The Process :
============================================================
Heap             245768 KB
RSS              246416 KB
Size             246744 KB
System Time           0 Sec
User Time           118 Sec
============================================================
Pairs Of Top Four Stall Counters:
[These counter pairs can be used with -h flag of collect
command to study application stall behavior more closely.]

#Dispatch0_br_target,Re_DC_miss
#Rstall_storeQ,Re_EC_miss
============================================================
```

Graph ... More ...

The table has three major sections. The first section attempts to attributed time that the application spent stalled to the various hardware conditions that caused the stall. In this particular example nearly the entire runtime of the application was spent stalled.

About 14% of the time was lost to Data TLB misses (`DTLB_miss`). This is one condition which can often be corrected using the compiler flag `-xpagesize=4M`.

A further ~80% of the runtime was spent stalled waiting for data that was not resident in the Data Cache (`Re_DC_miss`). About 60% of the runtime was spent waiting for data that was not in the External Cache (`Re_EC_miss`). This means that during the difference, about 20% of the runtime, the data was not in the Data Cache but was in the External Cache.

The next section reports information on IPC (Instructions per Cycle) which is a measure of how much useful work the processor is achieving each cycle. The idea of a Grouped IPC is a measure of how much work would be achieved if the processor did not encounter any stall conditions.

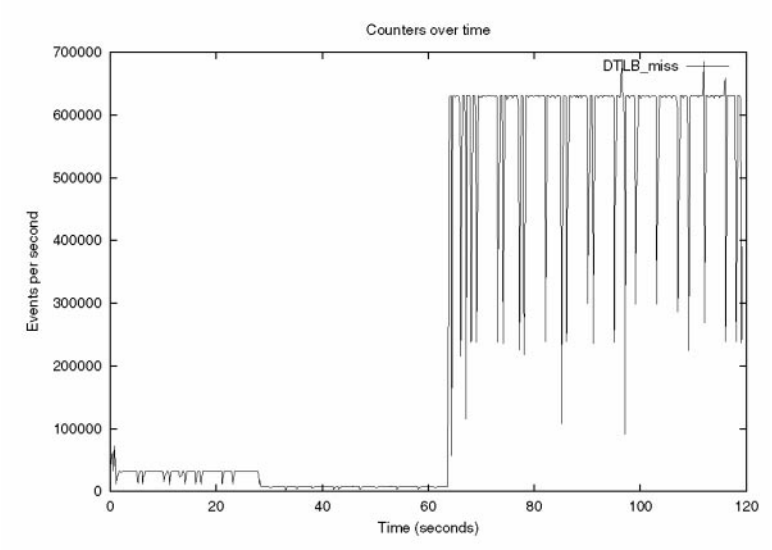
At this point the tool also reports the number of unfinished floating point traps that occurred during the run (on a system-wide basis). Floating-point traps occur very rarely, but if they do happen it can be tricky to realize that it is a problem.

The tool produces a section which reports events as a proportion, for example, cache misses per reference.

The final section reports the memory usage of the application.

If `gnuplot` is available on the path, this information is also presented in a graphical form.

In the following figure, the Data TLB misses are shown over the entire run of the program. In the first part of the run there were very few TLB misses, but in the last part there were significant numbers.



Instruction Frequencies

The tool BIT provides a number of pieces of information. The most useful of these are the number of times each routine is called and each instruction is executed. These will be covered in the section on profiles. However, BIT also gathers data on individual instruction frequencies – for example, how many floating point instructions were executed during the run of the benchmark.

The counts from BIT are for just the instrumented part of the application, so do not include library calls. In some cases not all the code is instrumented, so the resulting counts are not complete.

The following figure shows instruction frequencies for the test program.

Instruction frequency statistics from BIT

Instruction frequencies for whole program

Instruction	Executed	(%)
TOTAL	7963939485	(100.0)
float ops	4194304000	(52.7)
float ld st	3145728000	(39.5)
load store	3502243842	(44.0)
load	2432696322	(30.5)
store	1069547520	(13.4)

Instruction	Executed	(%)	Annulled	In Delay Slot
TOTAL	7963939485	(100.0)		
lddf	2097152000	(26.3)	100	0
add	1415578342	(17.8)	0	5242882
stdf	1048576000	(13.2)	0	262143900
faddd	1048576000	(13.2)	0	0
prefetch	791674576	(9.9)	0	0
br	602931826	(7.6)	0	0
subcc	602931628	(7.6)	0	2
lduw	335544322	(4.2)	0	335544320
stw	20971520	(0.3)	4	8

Whole program... Functions...

The difference between the number of floating-point operations and the number of floating point loads and stores is a count of the number of floating-point calculations (the additions and multiplications). In the above example, 53% of the operations involve floating point, 40% of the instructions are floating-point loads or stores, which means that 13% of the instructions are floating-point calculations.

BIT also correctly counts instructions in delay slots and annulled instructions. The delay slot is the instruction after a branch. Depending on whether the branch is taken or not this instruction may or may not end up being executed. If it is not executed it is an annulled instruction.

Time-Based Profiles

SPOT uses the `collect` command to gather a time based profile of the application. SPOT then renders this as a set of hyperlinked web pages. The profile of the test code is shown below:

```
current filename for subsequent output: ./spot_run4/html/functions.func
Functions sorted by metric: Exclusive User CPU Time

Excl.      Incl.      Excl.      Excl.     Excl. Bit  Excl. Bit   Excl. Bit   Name
User CPU   User CPU   Sys. CPU   Wall      Func       Inst Exec   Inst
  sec.       sec.       sec.       sec.    Count                  Annul
119.834    119.834    0.570      120.684   103        7963939485  204         <Total>
 56.059     56.059    0.         56.219     1          705167424    2         [trimmed] tlb_miss src Caller-callee
 35.815     35.815    0.         35.915     1          705167424    2         [trimmed] cache_miss src Caller-callee
 27.709     27.709    0.         27.729   100        6553603800   200         [trimmed] fp_routine src Caller-callee
  0.250      0.250    0.570       0.821     0                0     0         _memset
  0.       119.834    0.          0.        1              837     0         [trimmed] main src Caller-callee
  0.       119.834    0.          0.        0                0     0         _start
```

The profiles combine the information from collect with the information from BIT. The columns shown are as follows:

- The exclusive user time is the time spent executing code in the named routine.
- The inclusive user time is time spent executing code in a named routine, and all the routines that it calls.
- The system time column is the time that a given routine spends in system code.
- The wall time column is the elapsed time for the master thread. In single-threaded applications this profile is similar to the user and system time. For multi-threaded applications this profile tracks what the master thread is doing, and may not correspond to where the work is performed in the application.
- The exclusive function count column counts the number of times that a given routine is called during the run of the application. This information is provided by BIT and is only available on suitably compiled binaries. This count data is not available for routines that reside in libraries. This can be observed in the example where it appears as if `_memset` is never called.
- The exclusive instructions executed column is a count of the number of instructions executed in each routine during the run of the application. Again this information is not available for all routines – in particular, routines that reside in libraries are excluded.
- The exclusive instructions annulled column is a count of the number of instructions in delay slots that were annulled (not executed).

On the right of the page are hyperlinks that go to the following results:

- The trimmed link goes to disassembly (and source if available) which has been cut down in size so that only the important parts of the code remain. This is usually the best link to use to examine the profile in more detail because the untrimmed disassembly can sometimes be several MB in size.
- The name of the routine links to the untrimmed disassembly (and source if available).
- The src link goes to just the source for the given routine, showing the time and events attributed to each line of source code. This link is only available if the tool has been able to locate the source code for a particular routine.
- The caller-callee link goes to a page that shows which routines call which other routines.

The following figure shows an example of the mix of disassembly with source code, showing the attribution of time to individual assembly language instructions.

```
                                                                  26.   for (int i=0; i<size*16; i++) {cp= (int**)*cp;}
    0.          0              1     0                              [26]    12418:  sll         %o1, 4, %o1
    0.          0              1     0                              [26]    1241c:  cmp         %o1, 0
    0.          0              1     0                              [26]    12420:  ble,pn      %icc, 0x1246c
    0.          0              1     0                              [26]    12424:  add         %o1, -1, %o3
    0.          0              1     0                              [26]    12428:  add         %o3, 1, %g3
    0.          0              1     0                              [26]    1242c:  clr         %o1
    0.          0              1     0                              [26]    12430:  cmp         %g3, 1
    0.          0              1     0                              [26]    12434:  bl,pn       %icc, 0x12458
    0.          0              1     0                              [26]    12438:  mov         %o3, %o5
## 55.209       0      167772160     0                              [26]    1243c:  inc         %o1
    0.          0      167772160     0                              [26]    12440:  cmp         %o1, %o5
    0.          0      167772160     0                              [26]    12444:  ble,pt      %icc, 0x1243c
    0.801       0      167772160     0                              [26]    12448:  ld          [%o2], %o2
    0.          0              1     0                              [26]    1244c:  cmp         %o1, %o3
    0.          0              1     0                              [26]    12450:  bg,pn       %icc, 0x1246c
    0.          0              1     0                              [26]    12454:  nop
    0.          0              0     0                              [26]    12458:  ld          [%o2], %o2
    0.          0              0     0                              [26]    1245c:  inc         %o1
    0.          0              0     0                              [26]    12460:  cmp         %o1, %o3
    0.          0              0     0                              [26]    12464:  ble,a,pt    %icc, 0x1245c
    0.          0              0     0                              [26]    12468:  ld          [%o2], %o2
                                                                  27.   return cp;
```

The columns show the time spent on each assembly language instruction and the number of times that it was executed. Using this information it is possible to determine that the trip count for the loop between lines `0x1243c` and `0x12448` is nearly 170 million times, and that the loop was only entered once.

The time is recorded on the increment instruction at `0x1243c`. The sampling method used by the Performance Analyzer records time spent on the instruction waiting to be executed. So in this case it is necessary to identify the instruction that is executed before the increment; which turns out to be the load (in the delay slot) at `0x12448`. It is this load instruction that is taking all the time.

There are hyperlinks in the disassembly. Those that are in square brackets will jump to the line of source code that the assembly language instruction belongs to. The links that are part of branch instructions will jump to the target of the branch.

Hardware Event Profiles

If SPOT is run with the `-X` option, it will gather the profile using the performance counters that record the most stall time. In the following figure the profile shows where the hardware counter events occur in the test program.

Application HW counter profile output

```
./spot_run4/test.Dispatch0_br_target_Re_DC_miss.er: Experiment has warnings, see header for details
Current metrics: e.Dispatch0_br_target:e.Re_DC_miss:e.bit_fcount:e.bit_instx:e.bit_annul:name
Current Sort Metric: Exclusive Dispatch0_br_target Events ( e.Dispatch0_br_target )
Functions sorted by metric: Exclusive Dispatch0_br_target Events

Excl.                 Excl.         Excl. Bit  Excl. Bit   Excl. Bit   Name
Dispatch0_br_target   Re_DC_miss    Func       Inst Exec   Inst
Events sec.           Events sec.   Count                  Annul
0.826                 80.459        103        7963939485  204         <Total>
0.522                 30.965          1         705167424    2         tlb_miss
0.177                 31.270          1         705167424    2         cache_miss
0.127                 18.224        100        6553603800  200         fp_routine
0.                     0.             1               837    0         main
0.                     0.             0                 0    0         _start
```

More ...

```
./spot_run4/test.Rstall_storeQ_Re_EC_miss.er: Experiment has warnings, see header for details
Current metrics: e.Rstall_storeQ:e.Re_EC_miss:e.bit_fcount:e.bit_instx:e.bit_annul:name
Current Sort Metric: Exclusive Rstall_storeQ Events ( e.Rstall_storeQ )
Functions sorted by metric: Exclusive Rstall_storeQ Events

Excl.           Excl.         Excl. Bit  Excl. Bit   Excl. Bit   Name
Rstall_storeQ   Re_EC_miss    Func       Inst Exec   Inst
Events sec.     Events sec.   Count                  Annul
0.212           60.808        103        7963939485  204         <Total>
0.101            0.             0                 0    0         _memset
0.053            0.758        100        6553603800  200         fp_routine
0.030           30.111          1         705167424    2         cache_miss
0.028           29.939          1         705167424    2         tlb_miss
0.               0.             1               837    0         main
0.               0.             0                 0    0         _start
```

More ...

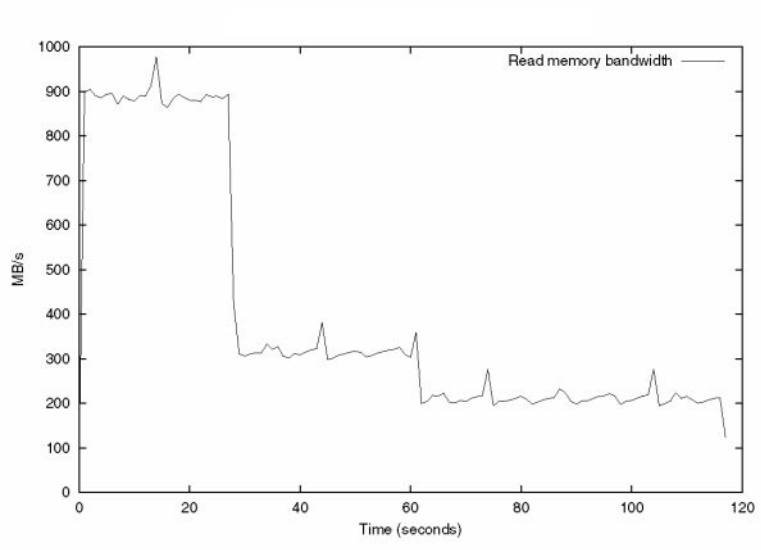
Looking at the first of the two profiles, it is apparent that the cycles spent stalled because of Data Cache miss events occur in all three routines. However, looking at the second profile it can be seen that the External Cache miss events only happen in the `cache_miss` and `tlb_miss` routines.

System-Wide Bandwidth Consumption and Trap Data

If SPOT is run with root permissions and the `-X` flag, it will try to gather system-wide information on the bandwidth consumption and the traps encountered over the entire run of the application. If other applications are also running on the system, these statistics will include bandwidth consumption and traps from those other applications.

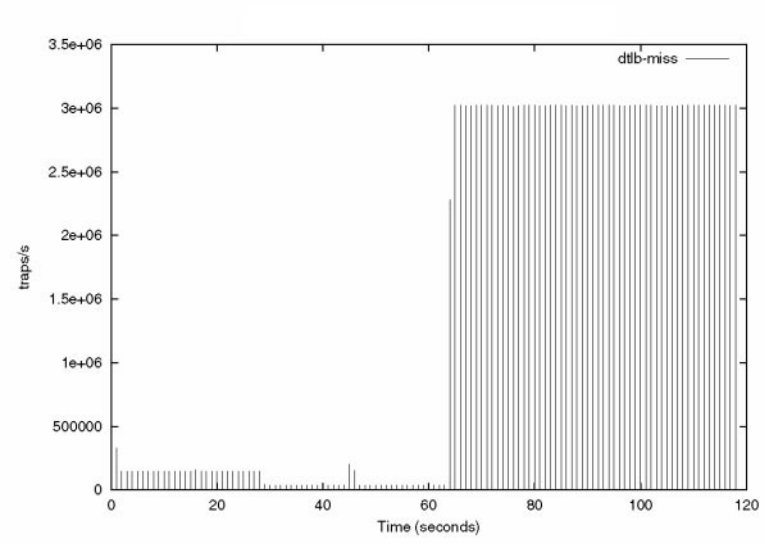
The bandwidth consumption and trap information is presented as a table that summarizes the entire run. If `gnuplot` is present in the user's path, then the results will also be available as graphs over time.

The following figure shows a graph of bandwidth consumption over the run of the application.



The most bandwidth is consumed during the first routine where data is being streamed from memory. The subsequent routines which are more tests of memory latency, consume less bandwidth.

The next figure shows the Data TLB traps reported over the entire run of the application.



As might be expected, the trap data looks very similar to the data reported by the hardware counters for the same event.

Concluding Remarks

SPOT generates a wealth of performance data that in most cases enables the user to rapidly narrow down the places in the code where time is being spent. It also assists in identifying the causes of application slowdown.

For more about SPOT, including downloading information, visit `http://cooltools.sunsource.net/spot/index.html`.

`http://developers.sun.com/solaris/articles/spot.html`

Adding DTrace Probes to User Code

Darryl Gove, November 20, 2007

The process of adding DTrace probes to userland code is described in the *Solaris Dynamic Tracing Guide* (`http://docs.sun.com/app/docs/doc/817-6223/chp-usdt?a=view`). However, there's no better way of learning how to do it than trying it out on a snippet of code.

Here's a short bit of code that calls a function twice, each time with different parameters. The plan is to insert a probe that can report the passed parameters.

```
#include <stdio.h>

void func(int a, int b)
{
  printf("a=%i, b=%i\n",a,b);
}

void main()
{
  func(1,2);
  func(2,3);
}
```

The first change is to add the `<sys/sdt.h>` header file. This file has definitions for the `DTRACE_PROBE<N>` macro. N represents the number of parameters that are to be reported by the probe. In this case we are going to pass two parameters (a and b) to the probe. As well as the parameters that are to be passed to the DTrace probe, the macro takes the name to be used for the application provider (in this case the name will be `myapp`) and the name of the probe (in this case `func_call`). The modified source code looks as follows:

```
#include <stdio.h>
#include <sys/sdt.h>
```

```
void func(int a, int b)
{
  DTRACE_PROBE2(myapp,func_call,a,b);
  printf("a=%i, b=%i\n",a,b);
}

void main()
{
  func(1,2);
  func(2,3);
}
```

The next step is to write a probe description file which DTrace will use to produce the probes. A full file would describe the stability of the probe in more detail, but a lightweight file just describes the probes defined by the provider application:

```
provider myapp
{
  probe func_call(int, int);
};
```

Having completed this, it's necessary to compile and link the application. Initially each source file needs to be compiled, and then before the application is linked, DTrace needs to be invoked to modify the object files, removing the calls to the probes, but leaving space for them to be reinserted. DTrace also needs to compile the probe description file into an object file. Finally the modified object files and the probe description file can be linked to produce the executable.

```
$ cc -c app.c
$ dtrace -G -32 -s probes.d app.o
$ cc probes.o app.o
```

The resulting code in the application looks like:

```
func()
        113a0:  9d e3 bf a0  save       %sp, -96, %sp
        113a4:  f0 27 a0 44  st         %i0, [%fp + 68]
        113a8:  f2 27 a0 48  st         %i1, [%fp + 72]
        113ac:  d0 07 a0 44  ld         [%fp + 68], %o0
        113b0:  01 00 00 00  nop
        113b4:  d2 07 a0 48  ld         [%fp + 72], %o1
        113b8:  11 00 00 45  sethi      %hi(0x11400), %o0
        113bc:  90 12 22 60  bset       608, %o0        ! 0x11660
        113c0:  d2 07 a0 44  ld         [%fp + 68], %o1
        113c4:  40 00 42 c7  call       printf  ! 0x21ee0
        113c8:  d4 07 a0 48  ld         [%fp + 72], %o2
        113cc:  81 c7 e0 08  ret
        113d0:  81 e8 00 00  restore
```

The nop at `0x113b0` is there for DTrace to dynamically patch with a trap instruction that will enable the DTrace probe.

Finally, the following is an example of using the new probe:

```
$ more script.d
myapp$target:::func_call
{
  @[arg0,arg1]=count();
}
$ dtrace -s script.d -c a.out
dtrace: script 'script.d' matched 1 probe
a=1, b=2
a=2, b=3
dtrace: pid 22355 has exited

               1               2               1
               2               3               1
```

The script just aggregates the parameters used in the function call. When the application terminates the aggregation is printed out showing the expected result of two calls to the routine each call with different parameters.

http://blogs.sun.com/d/entry/adding_dtrace_probes_to_user

Adding DTrace Probes to User Code (Part 2)

Darryl Gove, November 27, 2007

Adam Leventhal pointed out in the comments (http://blogs.sun.com/d/entry/adding_dtrace_probes_to_user#comment-1195662302000) to "Adding DTrace Probes to User Code" on page 220 that there is an improved approach to adding userland DTrace probes. He describes this approach on his blog (http://blogs.sun.com/ahl/entry/user_land_tracing_gets_better).

The approach solves two problems. First, that C++ name mangling makes it hard to add DTrace probes for that language. Second, that code with DTrace probes inserted in it will not compile on systems that do not have the necessary dtrace support.

So going back to the example code, I'll try to show the problem and the solution. Here's `app.cc`:

```
#include <stdio.h>
#include <sys/sdt.h>

void func(int a, int b)
{
  DTRACE_PROBE2(myapp,func_call,a,b);
  printf("a=%i, b=%i\n",a,b);
}

void main()
{
  func(1,2);
  func(2,3);
}
```

When compiled with the C compiler the following symbols get defined:

```
$ cc -c app.cc
$ nm app.o
app.o:

[Index]   Value      Size    Type  Bind  Other Shndx   Name
...
[10]    |         0|       0|FUNC |GLOB |0    |UNDEF  |__dtrace_myapp___func_call
...
```

When compiled with the C++ compiler the following happens:

```
$ CC -c app.cc
$ nm app.o
app.o:

[Index]   Value      Size    Type  Bind  Other Shndx   Name
...
[7]     |         0|       0|FUNC |GLOB |0    |UNDEF  |__1cbA__dtrace_myapp___func_call6FLL_v_
...
```

Because the call to the DTrace probe is not declared as being `extern 'C'` the compiler mangles the C++ function name.

The new approach that Adam describes involves DTrace preprocessing the probe description file to generate a header file, and then including the header file in the source code. The big advantage of having the header file is that it's now possible to declare the DTrace probes to have `extern 'C'` linkage, and avoid the name mangling issue. The syntax for preprocessing the probe description file is:

```
$ dtrace -h -s probes.d
```

This generates the following header file:

```
/*
 * Generated by dtrace(1M).
 */

#ifndef _PROBES_H
  #define _PROBES_H

  #include <unistd.h>

  #ifdef  __cplusplus
    extern "C" {
  #endif

  #if _DTRACE_VERSION
    #define MYAPP_FUNC_CALL(arg0, arg1)  __dtrace_myapp___func_call(arg0, arg1)
    #define MYAPP_FUNC_CALL_ENABLED()    __dtraceenabled_myapp___func_call()
    extern void __dtrace_myapp___func_call(int, int);
    extern int __dtraceenabled_myapp___func_call(void);
  #else
    #define MYAPP_FUNC_CALL(arg0, arg1)
    #define MYAPP_FUNC_CALL_ENABLED() (0)
  #endif
```

```
  #ifdef  __cplusplus
    }
  #endif

#endif  /* _PROBES_H */
```

The other advantage is that the header file can protect the definitions of the DTrace probes with `#if _DTRACE_VERSION`, which enables the same source to be compiled on systems which do not support DTrace.

The source code needs to be modified to support this syntax:

```
#include <stdio.h>
#include "probes.h"

void func(int a, int b)
{
  MYAPP_FUNC_CALL(a,b);
  printf("a=%i, b=%i\n",a,b);
}

void main()
{
  func(1,2);
  func(2,3);
}
```

The rest of the process is the same as before (`http://blogs.sun.com/d/entry/adding_dtrace_probes_to_user`).

`http://blogs.sun.com/d/entry/adding_dtrace_probes_to_user1`

Adding DTrace Probes to User Code (Part 3)

Darryl Gove, March 25, 2008

In "Adding DTrace Probes to User Code (Part 2)" on page 222, I discussed how to add DTrace USDT probes into user code. The critical step is to run the object files through DTrace, and for DTrace to record the instrumentation points and to modify the object files prior to linking. The output of this step is an object file that also needs to be linked into the executable. Here's an example:

```
$ cc -O -c app.c
$ cc -O -c app1.c
$ dtrace -G -32 -s probes.d app.o app1.o
$ cc -O probes.o app.o app1.o
```

The results from running the example code under a suitable DTrace script are:

```
$ sudo dtrace -s script.d -c a.out
dtrace: script 'script.d' matched 10 probes
```

```
a=1, b=2
a=1, b=2
a=1, b=2
a=2, b=3
dtrace: pid 20655 has exited

               2               3               1
               1               2               3
```

One question that has come up is whether it's necessary to run a single call to DTrace which instruments all the object files, or whether it's possible to use multiple calls.

The object file that DTrace produces `probes.o` is going to be over written with each call to DTrace, so it's no surprise that the naive approach of multiple calls to `dtrace` with each call generating the same object file does not work:

```
$ dtrace -G -32 -s probes.d app.o
$ dtrace -G -32 -s probes.d app1.o
$ cc -O app.o app1.o probes.o
$ sudo dtrace -s script.d -c a.out
dtrace: script 'script.d' matched 9 probes
a=1, b=2
a=1, b=2
a=1, b=2
a=2, b=3
dtrace: pid 20725 has exited

               2               3               1
               1               2               2
```

The next thing to try is whether changing the generated object file works:

```
$ dtrace -G -32 -s probes.d -o probe0.o app.o
$ dtrace -G -32 -s probes.d -o probe1.o app1.o
$ cc -O probe0.o app.o app1.o probe1.o
$ sudo dtrace -s script.d -c a.out
dtrace: script 'script.d' matched 9 probes
a=1, b=2
a=1, b=2
a=1, b=2
a=2, b=3
dtrace: pid 20673 has exited

               2               3               1
               1               2               2
```

And if we wanted more proof, swapping the order of the object files generates the following:

```
$ cc -O app.o app1.o probe1.o probe0.o
$ sudo dtrace -s script.d -c a.out
dtrace: script 'script.d' matched 1 probe
a=1, b=2
a=1, b=2
a=1, b=2
a=2, b=3
```

```
dtrace: pid 20683 has exited

               1              2              1
```

So the conclusion is that the only way it will work is by putting all the object files onto the command line of a single call to dtrace.

http://blogs.sun.com/d/entry/adding_dtrace_probes_to_user2

Recording Analyzer Experiments Over Long Latency Links (-S off)

Darryl Gove, September 4, 2007

I was recently collecting a profile from an app running on a machine in Europe, but writing the data back to a machine here in California. The application normally ran in 5 minutes, so I was surprised that it had made no progress after 3 hours when run under collect.

The Analyzer experiment looked like:

```
Dir : archives                 08/28/07   23:47:10
File: dyntext                  08/28/07   23:47:12
File: log.xml        598 KB    08/29/07   03:02:52
File: map.xml        3 KB      08/28/07   23:47:22
File: overview       4060 KB   08/29/07   03:02:51
File: profile        256 KB    08/28/07   23:47:56
```

Two of the files (log.xml and overview) had accumulated data since the start of the application, the other files had not. truss output showed plenty of writes to these files:

```
 0.0001 open("/net/remotefs/.../test.1.er/log.xml", O_WRONLY|O_APPEND) = 4
 0.0000 write(4, " < e v e n t   k i n d =".., 74)      = 74
 0.0004 close(4)                                        = 0
```

In fact it looked rather like opening and closing these remote files were taking all the time away from running the application. One of the Analyzer team suggested passing -S off to collect to switch off periodic sampling. Periodic sampling is collecting application state at one-second intervals. Using this flag, the application terminated in the usual 5 minutes and produced a valid profile.

http://blogs.sun.com/d/entry/recording_analyzer_experiments_over_long

Detecting Data TLB Misses

Darryl Gove, April 18, 2007

There are a couple of easy ways that an application can be tested for whether it is encountering DTLB misses.

- One way is to use the command `trapstat`(1M). This command requires administrator privileges to run and either reports trap activity on a system-wide basis, or can be used to follow the traps that a single process encounters
- The second way is to use `cputrack`(1) to track the events recorded by the hardware performance counters on the processor. The particular counters will depend on the hardware. An example using an UltraSPARC III is:

```
cputrack -c pic0=Instr_cnt,pic1=DTLB_miss -p <pid>
```

http://blogs.sun.com/d/entry/detecting_data_tlb_misses

Locating DTLB Misses Using the Performance Analyzer

Darryl Gove, April 19, 2007

DTLB misses typically appear in the Performance Analyzer (`http://docs.sun.com/app/docs/doc/819-0493`) as loads with significant user time. The following code strides through memory in blocks of 8192 bytes, and so encounters many DTLB misses

```
#include<stdlib.h>
void main()
{
  double *a;
  double total=0;
  int i;
  int j;
  a=(double*)calloc(sizeof(double),10*1024*1024+10001);
  for (i=0;i<10000;i++)
   for(j=0;j<10*1024*1024;j+=1024)
    total+=a[j+i];
}
```

A profile can be gathered as follows:

```
$ cc -g -O -xbinopt=prepare -o tlb tlb.c
$ collect tlb
```

Viewing the profile for the main loop using `er_print` produces the following snippet:

```
   Excl.
   User CPU
    sec.
```

```
...
   0.230                [11]    10c70:  prefetch    [%i5 + %i1], #n_reads
   0.                   [11]    10c74:  faddd       %f4, %f2, %f8
   3.693                [11]    10c78:  ldd         [%i5], %f12
## 7.685                [10]    10c7c:  add         %i5, %i3, %o3
   0.                   [11]    10c80:  faddd       %f8, %f0, %f10
   4.183                [11]    10c84:  ldd         [%o3], %f2
## 6.935                [10]    10c88:  add         %o3, %i3, %o1
   0.                   [11]    10c8c:  faddd       %f10, %f12, %f4
   0.                   [10]    10c90:  inc         3072, %i4
   4.123                [11]    10c94:  ldd         [%o1], %f0
## 7.065                [10]    10c98:  cmp         %i4, %i0
   0.                   [10]    10c9c:  ble,pt      %icc,0x10c70
   0.                   [10]    10ca0:  add         %o1, %i3, %i5
```

This can be compared with the situation where `mpss.so.1` has been preloaded to enable the application to get large pages:

```
   0.                   [11]    10c70:  prefetch    [%i5 + %i1], #n_reads
   0.                   [11]    10c74:  faddd       %f4, %f2, %f8
   0.                   [11]    10c78:  ldd         [%i5], %f12
## 7.445                [10]    10c7c:  add         %i5, %i3, %o3
   0.                   [11]    10c80:  faddd       %f8, %f0, %f10
   0.220                [11]    10c84:  ldd         [%o3], %f2
## 6.955                [10]    10c88:  add         %o3, %i3, %o1
   0.                   [11]    10c8c:  faddd       %f10, %f12, %f4
   0.                   [10]    10c90:  inc         3072, %i4
   0.340                [11]    10c94:  ldd         [%o1], %f0
   0.                   [10]    10c98:  cmp         %i4, %i0
   0.                   [10]    10c9c:  ble,pt      %icc,0x10c70
   0.                   [10]    10ca0:  add         %o1, %i3, %i5
```

The difference between the two profiles is the appearance of user time attributed directly to the load instruction (and not the normal instruction after the load).

It is possible to confirm that these are DTLB misses using the Performance Analyzer's ability to profile an application using the hardware performance counters:

```
$ collect -h DTLB_miss tlb
...
   Excl.
   DTLB_miss
   Events
...
          0             [11]    10c70:  prefetch    [%i5 + %i1], #n_reads
          0             [11]    10c74:  faddd       %f4, %f2, %f8
   30000090             [11]    10c78:  ldd         [%i5], %f12
          0             [10]    10c7c:  add         %i5, %i3, %o3
          0             [11]    10c80:  faddd       %f8, %f0, %f10
## 42000126             [11]    10c84:  ldd         [%o3], %f2
          0             [10]    10c88:  add         %o3, %i3, %o1
          0             [11]    10c8c:  faddd       %f10, %f12, %f4
          0             [10]    10c90:  inc         3072, %i4
   30000090             [11]    10c94:  ldd         [%o1], %f0
          0             [10]    10c98:  cmp         %i4, %i0
          0             [10]    10c9c:  ble,pt      %icc,0x10c70
          0             [10]    10ca0:  add         %o1, %i3, %i5
```

The events are reported on the load instructions that are causing the DTLB misses.

http://blogs.sun.com/d/entry/locating_dtlb_misses_using_the

Analyzer Probe Effect

Darryl Gove, June 2, 2006

When profiling an application, using the Sun Studio Performance Analyzer, there is some probe effect due to both the interruption of the application to gather profiling data, and the act of recording that data to disk. The following study is an attempt to quantify the probe effect when running on a T2000 (UltraSPARC-T1) system. This system has the capability to run many threads, consequently the relationship between the probe effect and the number of threads is very interesting.

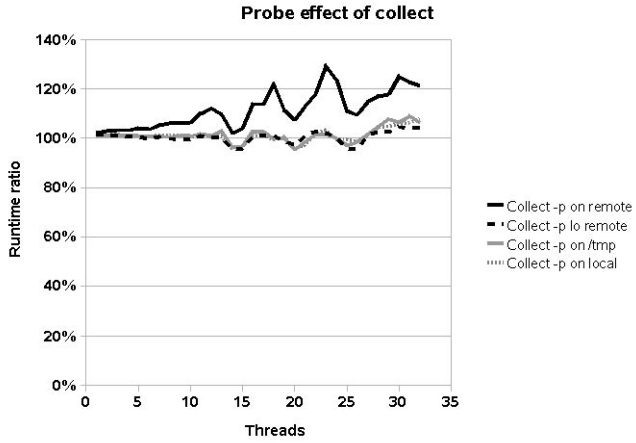
The following test program was used

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/time.h>


int * restrict array;
int array_length;
int round_off;
long threads=1;

#define SIZE 128*1024*1024

int results=0;
pthread_mutex_t results_mutex;
__thread int sum = 0;
__thread int id;

void* thread_code(void* v)
{
  int i;
  id = (int)v;
  for (i=0; i<SIZE; i++)
  {
    sum+=array[i];
  }
  pthread_mutex_lock(&results_mutex);
  results+=sum;
  pthread_mutex_unlock(&results_mutex);
  return 0;
}

void main(int argc,const char** argv)
{
  int i,rtn,id;
```

```
  pthread_t *thread_array;
  hrtime_t start_time;


  array=(int*)malloc(sizeof(int)*SIZE);
  int sum=0;

  if (argc==2) {threads=atoi(argv[1]);}

  pthread_mutex_init(&results_mutex,NULL);

  thread_array=(pthread_t *)malloc(sizeof(pthread_t)*threads);

  array_length=SIZE/threads;
  round_off=SIZE-(array_length*threads);

  for (i=0; i<SIZE;i++)
  {
    array[i]=1;
  }

   /*Multithreaded*/
   rtn=1;
   /*Make the threads*/
   start_time=gethrtime();
   for (i=0; i<10; i++)
{
   for (id=0;id<threads;id++)
   {
     pthread_create(&thread_array[id],NULL,&thread_code,(void*)id);
   }

   /*Join the threads*/
   for (id=0;id<threads;id++)
   {
     pthread_join(thread_array[id],NULL);
   }
}
   printf("Elapsed time (seconds)=%5.3f ",(gethrtime()-start_time)/1000000000.0);
   printf("Total is %i\n",results);

}
```

The study was to look at the runtime of the application as the number of threads was increased, and also when the experiment was recorded to local or remote disk. The other factor that was investigated was the use of low frequency profiling (-p lo).

The results from this can be shown as a graph. The y-axis is the ratio of the runtime under profiling against the runtime without profiling. The x-axis is the number of active threads.



The results suggest that profiling to remote disk can often cause significant probe effect; however, this probe effect may be manageable for low active thread counts. The alternatives of recording experiments to local disk, `/tmp`, or using low frequency profiling (`-p lo`) all result in much lower overhead.

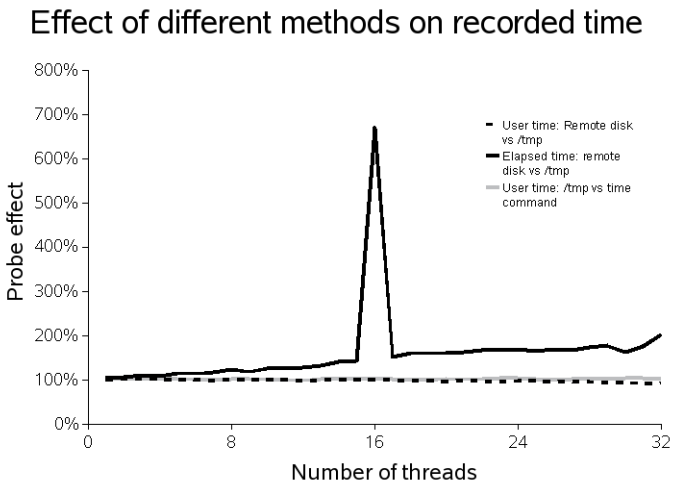
`http://blogs.sun.com/d/entry/analyzer_probe_effect`

Analyzer Probe Effects (Part 2)

Darryl Gove, December 15, 2006

In "Analyzer Probe Effect" on page 229, I looked at the probe effect when running the Performance Analyzer (`http://developers.sun.com/solaris/articles/analyzer_qs.html`) on multithreaded code. The results showed that the probe effect, in terms of difference in runtime, was most pronounced when using multiple threads and recording the experiment to remote disk.

An alternative metric for probe effect was the amount of distortion in the results. To measure this I took the same code and observed the user time recorded by Analyzer when the experiments were recorded to remote disk or to `/tmp`. I also compared this to the user time reported by the command `time`. The results of this are shown in the following figure.



The results show that the runtime when recording the experiment to remote disk was more significantly increased than in my previous experiment. There is also one data point where the runtime was substantially disturbed.

The interesting part of the graph is the recorded user time. For both situations, the recorded user time is pretty much identical to that recorded by the `time` command. Possibly most critically, this is true even when the runtime is hugely disturbed. Anecdotally, this is what we've been assuming for a while, that the Analyzer data is largely correct even in the situation where there is some activity on the machine. (I'm sure that this would not be the case if the machine were placed under an extreme load, for example running two multithreaded benchmarks.)

There is one rather odd observation, that when recording to the remote file system, the user time is slightly underestimated. But it is apparent from the graph that this is not a large effect.

The conclusion is that recording to a remote file system does not cause large distortion to the collected data, although it does cause the application to run more slowly.

`http://blogs.sun.com/d/entry/analyzer_probe_effects_part_2`

Process(or) Forensics

Thomas Bastian, July 21, 2008

Introduction

Optimizing application performance is a multifaceted undertaking. Among all the tools and methods available to us, getting a glimpse at performance metrics from the processor point of

view allows us to highlight yet another facet of the application. Most modern microprocessors include so-called performance counters that provide event-based drill-down of the processor utilization and efficiency. On the Solaris OS, the events and performance counters are made available to a user via two commands: `cpustat`(1M) and `cputrack`(1).
The former looks at system-wide statistics, whereas the latter one looks at process-wide statistics. You can see what events and performance counters are available to you by issuing the command: `cputrack -h`. On my system, the (shortened) output looks like:

```
blog@tbf3400> cputrack -h
Usage:
       cputrack [-T secs] [-N count] [-Defhnv] [-o file]
       -c events [command [args] | -p pid]
       <snip>
       CPU performance counter interface: AMD Opteron & Athlon64

       event specification syntax:
       [picn=]<eventn>[,attr[n][=<val>]][,[picn=]<eventn>[,attr[n][=<val>]],...]

       event[0-3]: FP_dispatched_fpu_ops FP_cycles_no_fpu_ops_retired
       FP_dispatched_fpu_ops_ff LS_seg_reg_load
       LS_uarch_resync_self_modify LS_uarch_resync_snoop
       LS_buffer_2_full LS_retired_cflush LS_retired_cpuid
       DC_access DC_miss DC_refill_from_L2 DC_refill_from_system
       DC_misaligned_data_ref DC_uarch_late_cancel_access
       DC_uarch_early_cancel_access DC_dispatched_prefetch_instr
       DC_dcache_accesses_by_locks BU_memory_requests
       <snip>
       attributes: edge pc inv cmask umask nouser sys

       See Chapter 10 of the "BIOS and Kernel Developer's Guide for the
       AMD Athlon 64 and AMD Opteron Processors," AMD publication #26094
```

`cpustat`(1M) and `cputrack`(1) also provide a reference to processor documentation where one can find more pertinent information about each individual event. For another, excellent tool to collect performance data that can leverage hardware performance counters and much more, please refer to the `collect` command that comes with the free Sun Studio 12 (`http://developers.sun.com/sunstudio/downloads/index.jsp`) suite. Using `cputrack`(1) is really straightforward. To demonstrate its usage, we will use a simple C program that makes lots of misaligned data accesses. On an AMD Opteron based system, we can use the `DC_misaligned_data_ref` event to count misaligned data references.

```
blog@tbf3400> cputrack -t -N 5 -c DC_misaligned_data_ref ./misaligned_access 1
  time lwp event tsc pic0
  1.041 1 tick 791649113 75695259
  2.051 1 tick 790899011 75756743
  3.041 1 tick 779480069 74674120
  4.041 1 tick 791630075 75857721
  5.051 1 tick 798199772 76464257
```

Our little application generates a fair number of misaligned data references (`pic0` column) in relation to the number of clock cycles (`tsc` column) elapsed (~ 9.5 %).

As a follow up, one would probably be interested in finding out where these misaligned data references happen. Unfortunately, this question is more tricky to answer on x86 platforms. On the SPARC platform, the Sun Studio performance analyzer (`http://docs.sun.com/app/docs/doc/819-5264`) includes capabilities and tools that will help out. (Have a look also at SPOT (`http://cooltools.sunsource.net/spot/`).)

A simple way to answer the above question is to make use of the CPU performance counter library (`libcpc`(3LIB)) directly by writing our own code. We will use this technique to write a standalone shared library object that can be dynamically loaded into existing applications (using the `LD_PRELOAD` environment variable) to perform simple profiling based on processor performance counters.

First, we use the shared object initialization hook to prepare our runtime environment:

```
#pragma init(dprof_init)
void dprof_init() {
```

Next, we set up a `SIGEMT` signal handler (more later):

```
/* SIGEMT */
bzero(&emt_action, sizeof(emt_action));
emt_action.sa_sigaction = dprof_emt_handler;
emt_action.sa_flags = SA_RESTART | SA_SIGINFO;
if (sigaction(SIGEMT, &emt_action, NULL) == -1) {
  fprintf(stderr, "dprof: sigaction: %s\n", strerror(errno));
  exit(1);
}
```

Then, we initialize the CPU performance counter library:

```
if ((cpc = cpc_open(CPC_VER_CURRENT)) == NULL) {
  fprintf(stderr, "dprof: perf. counters unavailable: %s\n", strerror(errno));
  exit(1);
}
if ((set = cpc_set_create(cpc)) == NULL) {
  fprintf(stderr, "dprof: could not create set: %s", strerror(errno));
  exit(1);
}
```

So we are pretty much done with initialization. Now let's set up the event of interest:

```
/* AMD Opteron DC_misaligned_data_ref */
event_name ="DC_misaligned_data_ref";
       preset = 0xffffffffffffffff - 100;
nattrs = 0;
```

And add it to the list of events to watch. We also would like to be notified by a signal when the counter overflows (`CPC_OVF_NOTIFY_EMT`):

```
if ((ckey = cpc_set_add_request(cpc, set, event_name, preset,
                                CPC_COUNT_USER | CPC_OVF_NOTIFY_EMT,
                                nattrs, cpc_attrs)) != 0) {
```

```
                fprintf(stderr, "dprof: cannot add request to set: %s\n", strerror(errno));
                exit(1);
            }
```

And finally bind the performance counters to the current light weight process (LWP):

```
        if (cpc_bind_curlwp(cpc, set, 0) == -1) {
            fprintf(stderr, "dprof: cannot bind lwp%d: %s\n",
                            _lwp_self(), strerror(errno));
            exit(1);
        }
```

We're done. We should actually be getting `SIGEMT` signals every time the underlying performance counter overflows (that is with our current settings after 100 events). So let's see what we need to do inside the signal handler:

```
void dprof_emt_handler(
    int sig,
    siginfo_t *sip,
    void *arg
) {
    /* sig should be SIGEMT and sip->si_code should be EMT_CPCOVF */
    if (dladdr(sip->si_addr, &dlip)) {
        printf("|%p|%s|%p|%s|%p|\n",
                sip->si_addr,
                dlip.dli_fname, dlip.dli_fbase, dlip.dli_sname, dlip.dli_saddr);
    } else {
        printf("|%p|||||\n", sip->si_addr);
    }
    /* Restart counter */
    if (cpc_set_restart(cpc, set) != 0) {
        fprintf(stderr,
                "dprof: cannot restart lwp%d: %s\n",
                _lwp_self(), strerror(errno ));
        exit(1);
    }
}
```

So our very basic signal handler tries to resolve the *sip->si_addr* given to us by the system and print the information out (really, one would probably want to aggregate events into a buffer and process them later). Finally, before exiting the signal handler, the performance counter is reset to its preset value. With the little C program that generates misaligned data references, we see following output (shortened):

```
|400da4|misaligned_access|400000|fa|400d98|
```

Using the modular debugger (`mdb`), we disassemble the code at the given address:

```
blog@tbf3400> mdb misaligned_access
> 400da4::dis
fa: movlpd (%rdi),%xmm0
fa+4: mulsd (%rsi),%xmm0
fa+8: movlpd %xmm0,(%rdi)
fa+0xc: movlpd (%rdi),%xmm0
```

```
fa+0x10: movlpd %xmm0,(%rsi)
fa+0x14: ret
0x400dad: nop
0x400db0: nop
0x400db4: nop
fb: movlpd (%rdi),%xmm0
fb+4: mulsd (%rsi),%xmm0
fb+8: movlpd %xmm0,(%rdi)
fb+0xc: movlpd (%rdi),%xmm0
fb+0x10: movlpd %xmm0,(%rsi)
>
```

and find out that the function named *fa()* is likely to be the source of misaligned data references.

A couple of words of caution with this technique: it is based on statistical sampling. The processor gets interrupted whenever the underlying performance counter overflows. Expect delays between the time the processor fires the interrupt and the time the operating system services it. That is, the program counter made available to the signal handler may be slightly off.

(http://blogs.sun.com/partnertech/entry/process_or_forensics)

Runtime Checking With bcheck

Darryl Gove, November 3, 2006

Reading uninitialized memory, reading past the end of arrays, or freeing memory twice are some of a number of problems which are hard to detect from browsing the source code of an application. This problems usually cause runtime behavior which is random and hard to debug – the very act of running the program under the debugger can change the behavior of the program (usually causing it to work perfectly ;-).

The tool bcheck (http://docs.sun.com/source/819-3683/RunTCheck.html#24560) is an easy way of testing a program for uninitialized variables, or a number of other common runtime problems. The command is a convenient wrapper for dbx (http://docs.sun.com/app/docs/doc/819-3683). The command line looks like:

```
bcheck -all <app> <params>
```

Consider this snippet of code:

```
void test()
{
  int a,b;
  a=b;
}

void main()
{
  int i,j;
  i=j;
  test();
}
```

This code has a couple of variables (j and b) which are used without being initialized first. Compiling the code with no optimization and then running under bcheck produces the following output.

```
$ cc -g uninit.c
$ bcheck -all a.out
Reading a.out
Reading ld.so.1
Reading rtcapihook.so
Reading libc.so.1
Reading libdl.so.1
Reading libc_psr.so.1
Reading rtcaudit.so
Reading libmapmalloc.so.1
Reading libgen.so.1
Reading rtcboot.so
Reading librtc.so
access checking - ON
memuse checking - ON
Running: a.out
(process id 25160)
RTC: Enabling Error Checking...
RTC: Running program...

Checking for memory leaks...
  errors are being redirected to file 'a.out.errs'

Actual leaks report    (actual leaks:            0  total size:          0 bytes)
Possible leaks report  (possible leaks:          0  total size:          0 bytes)

Checking for memory use...
  errors are being redirected to file 'a.out.errs'
Blocks in use report   (blocks in use:           0  total size:          0 bytes)

RTC output redirected to logfile 'a.out.errs'

execution completed, exit code is 1
```

The more complete report is recorded to the file a.out.errs:

```
$ more a.out.errs
<rtc>
 Read from uninitialized (rui):
Attempting to read 4 bytes at address 0xffbff9f0
    which is 96 bytes above the current stack pointer
Variable is 'j'
=>[1] main(), line 10 in "uninit.c"

<rtc>
 Read from uninitialized (rui):
Attempting to read 4 bytes at address 0xffbff988
    which is 96 bytes above the current stack pointer
Variable is 'b'
=>[1] test(), line 4 in "uninit.c"
  [2] main(), line 11 in "uninit.c"
```

```
Actual leaks report    (actual leaks:            0  total size:          0 bytes)

Possible leaks report  (possible leaks:          0  total size:          0 bytes)
```

Since the program has been compiled with debug and low optimization, `bcheck` is able to report both the line number and the name of the variable causing the problem. When debug is enabled, the line number is usually available, but the variable name is normally only available at low optimization.

`http://blogs.sun.com/d/entry/runtime_checking_with_bcheck`

Locating Memory Access Errors With the Sun Memory Error Discovery Tool

Darryl Gove, September 2007

Summary

The Sun Memory Error Discovery Tool detects and reports common memory access errors such as accessing uninitialized memory, writing past the end of an array, or accessing memory after it has been freed.

Introduction

Memory access errors are one of the hardest types of error to detect. The reason for this is that the symptoms of the error occur arbitrarily far from the point where the error occurred. The Sun Memory Error Discovery Tool (the Discovery tool) is designed to detect and report common memory access errors such as accessing uninitialized memory, writing past the end of an array, or accessing memory after it has been freed.

The Discovery tool is included in the CMT Developer Tools. These tools work with Sun Studio 12 and are a free download. See the installation instructions (`http://cooltools.sunsource.net/cmtdt/install.html`) for more information.

Using the Discovery Tool

The target application needs to be compiled with Sun Studio 12 together with the `-xbinopt=prepare` compiler flag, an optimization level of at least `-xO1` and the `-g` compiler option to get better debugging information from the tool. The following example code has an error where the array element `a[1]` is read without having first been initialized.

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

void main()
{
  int * a=malloc(100*sizeof(int));
  a[1]++;
  free(a);
}
```

The steps necessary to compile, instrument, and run the code are shown in the following example.

```
$ cc -g -O -xbinopt=prepare -o mem mem.c
$ discover mem
$ mem
```

At the end of the run, the instrumented application reports any memory errors that were encountered during the run. Each reported error can have multiple sections. The first section always describes where the error occurred, as shown in the following example.

```
ERROR (UMR): accessing uninitialized data
             from address 0x5000c (4 bytes) at:
        main() + 0x54 [/tmp/mem:0x30054]
          <mem.c:7>:
                 4:    void main()
                 5:    {
                 6:      int * a=malloc(100*sizeof(int));
                 7:=>    a[1]++;
                 8:      free(a);
                 9:    }
```

There might be other sections that report where the memory was allocated, and if necessary, where the memory was freed. It is also possible to generate the output as an HTML report (`http://cooltools.sunsource.net/nonav/discover/discover_example.html`) by setting the environment variable `DISCOVER_HTML` to `1`.

The following code shows an example of reading memory beyond the end of an allocated array. Another error message is reported for the same line because it is both a read and a write of the 101st element of the array.

```
ERROR (ABR): reading memory beyond array bounds
             at address 0x50198 (4 bytes) at:
        main() + 0x54 [/tmp/mem:0x30054]
          <mem.c:7>:
                 4:    void main()
                 5:    {
                 6:      int * a=malloc(100*sizeof(int));
                 7:=>    a[100]++;
                 8:      free(a);
                 9:    }
```

Concluding Remarks

Although memory access errors can be hard to locate, the Sun Memory Error Discovery Tool should make the process of locating these errors significantly easier, leading to a shorter development time and a more robust final product. More information, including links to installation instructions, the man page, and a detailed user's guide, is available online (`http://cooltools.sunsource.net/discover`).

`http://developers.sun.com/solaris/articles/discovery_tool.html`

◆ ◆ ◆ CHAPTER 6

Libraries and Linking

This chapter covers topics associated with writing libraries and finding libraries at runtime. It explains why using the `LD_LIBRARY_PATH` environment variable is a hazardous approach, as well as details on various ways that the user can fix and debug runtime issues involving libraries and symbols.

Calling Libraries

Darryl Gove, June 26, 2008

I've previously blogged about measuring the performance of calling library code (`http://blogs.sun.com/d/entry/snippet_from_book_cost_of`). Let's quickly cover where the costs come from, and what can be done about them.

The most obvious cost is that of making the call. Probably this is a straight-forward call instruction, although calls over indirection can involve loading the address from memory first of all. There's also a linkage table to negotiate. Let's take a look at that:

```
#include <stdio.h>
void f()
{
  printf("Hello again\n");
}

void main()
{
  printf("Hello World\n");
  f();
}
```

There's two calls to `printf` in the code. `libc` is lazy-loaded, so the first call does the setup, and then we can see what happens more generally on the second call.

```
% cc -g p.c
% dbx a.out
```

```
Reading ld.so.1
Reading libc.so.1
(dbx) stop in f
(2) stop in f
(dbx) run
Running: a.out
(process id 63626)
Reading libc_psr.so.1
Hello World
stopped in f at line 4 in file "test.c"
    4     printf("Hello again\n");
(dbx) stepi
stopped in f at 0x00010bc0
0x00010bc0: f+0x0008:   bset     48, %l0
0x00010bc4: f+0x000c:   call     printf [PLT]   ! 0x20ca8
0x00010bc8: f+0x0010:   or       %l0, %g0, %o0
0x00020ca8: printf        [PLT]:        sethi    %hi(0x15000), %g1
0x00020cac: printf+0x0004 [PLT]:        sethi    %hi(0xff31c400), %g1
0x00020cb0: printf+0x0008 [PLT]:        jmp      %g1 + 0x00000024
0x00020cb4: _get_exit_frame_monitor        [PLT]:       sethi    %hi(0x18000), %g1
0xff31c424: printf       :      save     %sp, -96, %sp
```

So the call to `printf` actually jumps to a procedure lookup table, which then jumps to the actual start address of the library code.

So that's the additional costs of libraries. But just doing a call instruction also has some costs:

- For SPARC processors, there's the possibility of hitting a register windows (`http://blogs.sun.com/d/entry/flush_register_windows`) spill/fill trap.
- The compiler does not know whether the routine being called will read or write to memory. So all variables need to be stored back to memory before the call, and read from memory afterwards. This can get quite ugly, particularly for floating-point codes where there may be quite a few active registers at any one time. This behavior can be avoided using the pragmas `does_not_read_global_data`, `does_not_write_global_data`, `no_side_effect`. The `no_side_effect` pragma means that the compiler can eliminate the call to the routine if the return value is not used.
- There are also ABI issues. For example, the SPARC V8 ABI (`http://blogs.sun.com/d/entry/32_bits_good_64_bits`) requires floating-point parameters to be passed in the integer registers. Doing this requires storing the fp registers to the stack and then loading the values into the integer registers, and doing the opposite on the other side of the call!

So generally calling routines can be time consuming, but what can be done?

- Check to see whether you might use intrinsics such as `fsqrt` rather than calling `sqrt` in `libc` (`-xlibmil`)
- Compiling with `-x04` enables the compiler to avoid calls by inlining within the same source file.
- Compiling and linking with `-xipo` enables the compiler to do cross-file inlining.
- Make sure that every call that is made does substantial work - not just a handful of instructions.

- Profile the application to confirm that there is real work being done in library code, and that the library routines called do perform substantial numbers of instructions on every invocation.

http://blogs.sun.com/d/entry/calling_libraries

LD_LIBRARY_PATH – Just Say No

Rod Evans, July 10, 2004

A recent email discussion reminded me of how fragile, and prevalent, LD_LIBRARY_PATH use is. Within a *development* environment, this variable is very useful. I use it all the time to experiment with new libraries. But within a *production* environment, use of this environment variable can be problematic. See Directories Searched by the Runtime Linker (http://docs.sun.com/app/docs/doc/819-0690/chapter6-63352?a=view) for an overview of LD_LIBRARY_PATH use at runtime.

People use this environment variable to establish search paths for applications whose dependencies do not reside in constant locations. Sometimes wrapper scripts are employed to set this variable, other times users maintain an LD_LIBRARY_PATH within their .profile. This latter model can often get out of hand - try running:

```
% ldd -s /usr/bin/date

...
find object=libc.so.1; required by /usr/bin/date
search path=/opt/ISV/lib     (LD_LIBRARY_PATH)
```

If you have a large number of LD_LIBRARY_PATH components specified, you'll see libc.so.1 being wastefully searched for, until it is finally found in /usr/lib. Excessive LD_LIBRARY_PATH components don't help application startup performance.

Wrapper scripts attempt to compensate for inherited LD_LIBRARY_PATH use. For example, a version of acroread reveals:

```
LD_LIBRARY_PATH="`prepend "$ACRO_INSTALL_DIR/$ACRO_CONFIG/lib:\
$ACRO_INSTALL_DIR/$ACRO_CONFIG/lib" "$LD_LIBRARY_PATH"`
```

The script is prepending its LD_LIBRARY_PATH requirement to any inherited definition. Although this provides the necessary environment for acroread to execute, we're still wasting time looking for any system libraries in the acroread subdirectories.

When 64-bit binaries came along, we had a bit of a dilemma with how to interpret LD_LIBRARY_PATH. But, because of its popularity, it was decided to leave it applicable to both class of binaries (64 and 32-bit), even though it's unusual for a directory to contain both 64 and 32-bit dependencies. We also added LD_LIBRARY_PATH_64 and LD_LIBRARY_PATH_32 as a means of specifying search paths that are specific to a class of objects. These class-specific environment variables are used *instead* of any generic LD_LIBRARY_PATH setting.

Which leads me back to the recent email discussion. Seems a customer was setting both the _64 and _32 variables as part of their startup script, because both 64 and 32–bit processes could be spawned. However, one spawned process was acroread. Its LD_LIBRARY_PATH setting was being overridden by the _32 variable, and hence it failed to execute. Sigh.

Is there a solution to this mess? I guess we could keep bashing LD_LIBRARY_PATH into submission some way, but why not get rid of the LD_LIBRARY_PATH requirement altogether? This can be done. Applications and dependencies can be built to include a *runpath* using ld(1), and the -R option. This path is used to search for the dependencies of the object in which the runpath is recorded. If the dependencies are not in a constant location, use the $ORIGIN (http://docs.sun.com/app/docs/doc/817-1984/6mhm7pl38?a=view) token as part of the pathname.

Is there a limitation to $ORIGIN use? Yes, as directed by the security folks, expansion of this token is not allowed for secure applications. But then again, for secure applications, LD_LIBRARY_PATH components are ignored for non-secure directories anyway. See the "Security" description in the *Linker and Libraries Guide* (http://docs.sun.com/app/docs/doc/817-1984/chapter3-9?a=view).

For a flexible mechanism of finding dependencies, use a runpath that includes the $ORIGIN token, and try not to create secure applications :-)

http://blogs.sun.com/rie/entry/tt_ld_library_path_tt

Dependencies – Define What You Need, and Nothing Else

Rod Evans, July 15, 2004

I recently attended Usenix (http://www.usenix.org/events/usenix04/), where Bryan Cantrill (http://blogs.sun.com/bmc) explained how DTrace (http://www.sun.com/bigadmin/content/dtrace/) had been used to uncover some excessive system load brought on by the behavior of one application. A member of the audience asked whether the application was uncovering a poorly implemented part of the system. Bryan responded that in such cases the system will always be analyzed to determine whether it could do better. But there comes a point where if an application requests an expensive service, that's what it will get. Perhaps the application should be reexamined to see if it needs the service in the first place?

This observation is very applicable to the runtime linking environment. Over the years we've spent countless hours pruning the cost of ld.so.1(1), only to see little improvement materialize with real applications. Alternatively, there's no better way of reducing the overhead of servicing a particular operation, than not requesting the operation in the first place :-)

Think about the work the runtime linker has to do to load an object. It has to find the object (sometimes through a plethora of LD_LIBRARY_PATH components), load the object, and relocate it. The runtime linker then repeats this process for any dependencies of the loaded object. That's a lot of work. So why do so many applications load dependencies they don't need?

Perhaps it's sloppiness, too much `Makefile` cut-and-pasting, or the inheritance of global build flags. Or, perhaps the developer doesn't realize a dependency isn't required. One way to discover dependency requirements is with `ldd`(1) and the `-u` option. For example, neither this application, nor any of its dependencies, make reference to `libmd5.so.1`:

```
% ldd -u -r app
    ...
    unused object=/lib/libmd5.so.1
```

Note the use of the `-r` option. We want to force `ldd`(1) to bind all relocations, data and functions. However, here we're wastefully loading `libmd5.so.1`. This should be removed as a dependency.

The `-u` option uncovers totally unused objects, but there can still be wasteful references. For example, the same application reveals that a number of objects have wasteful dependencies:

```
% ldd -U -r app
    ...
    unreferenced object=/usr/openwin/lib/libX11.so.4; unused dependency of app
    unreferenced object=/usr/openwin/lib/libXt.so.4; unused dependency of app
    unreferenced object=/lib/libw.so.1; unused dependency of app
```

Although the `X` libraries are used by some dependency within the process, they're not referenced by the application. There are data structures maintained by the runtime linker that track dependencies. If a dependency isn't required, we've wasted time creating these data structures. Also, should the object that requires the `X` libraries be redelivered in a form that no longer needs the `X` libraries, the application is still going to cause them to be wastefully loaded.

To reduce system overhead, only record those dependencies you need, and nothing else. As part of building the core OS, we run scripts that perform actions such as `ldd -U` in an attempt to prevent unnecessary dependency loading from creeping in.

Note, you can also observe unused object processing using the runtime linkers debugging capabilities (`LD_DEBUG=unused`). Or, you can uncover unused objects during a link-edit using the same debugging technique (`LD_OPTIONS=-Dunused`). Another way of pruning unwanted dependencies is to use the `-z ignore` option of `ld`(1) when building your application or shared object.

`http://blogs.sun.com/rie/entry/tt_dependencies_tt_define_what`

Dependencies – Perhaps They Can Be Lazily Loaded

Rod Evans, July 27, 2004

In "Dependencies – Define What You Need, and Nothing Else" on page 244, I stated that you should only record those dependencies you need, and nothing else. There's another step you can take to reduce start-up processing overhead.

Dynamic objects need to resolve symbolic references from each other. Function calls are typically implemented through an indirection that allows the function binding to be deferred until the function call is first made. See "When Relocations Are Performed" in *Linker and Libraries Guide* (http://docs.sun.com/app/docs/doc/817-1984/chapter3-4?a=view). Because of this deferral, it is also possible to cause the defining dependency to be loaded when the function call is first made. This model is referred to as Lazy Loading (http://docs.sun.com/app/docs/doc/817-1984/chapter3-7?a=view).

To establish lazy loading, you must pass the `-z lazyload` option to `ld`(1) when you build your dynamic object. In addition, the association of a symbol reference to a dependency requires that the dependency is specified as part of the link-edit. It is recommended that you use the link-editor's `-z defs` option to ensure that all dependencies are specified when you build your dynamic object. The following example establishes lazy dependencies for the references `foo()` and `bar()`.

```
% cat wally.c

    extern void foo(), bar();

    void wally(int who)
    {
        who ? foo() : bar();
    }
% cc -o wally.so wally.c -G -Kpic -zdefs -zlazyload -R'$ORIGIN' foo.so bar.so
```

The lazy loading attribute of these dependencies can be displayed with `elfdump`(1).

```
% elfdump -d wally.so | egrep "NEEDED|POSFLAG"

        [0]  POSFLAG_1        0x1               [ LAZY ]
        [1]  NEEDED           0x66              foo.so
        [2]  POSFLAG_1        0x1               [ LAZY ]
        [3]  NEEDED           0x6d              bar.so
```

By default, `ldd`(1) displays all dependencies, in that it will force lazy loaded objects to be processed. To reveal lazy loading, use the `-L` option. For example, when a dynamic object is loaded into memory, all data relocations are performed before the object can gain control. Thus the following operation reveals that neither dependency is loaded.

```
% ldd -Ld wally.so
%
```

Once function relocations are processed, both dependencies are loaded to resolve the function reference.

```
% ldd -Lr wally.so
        foo.so =>       ./foo.so
        bar.so =>       ./bar.so
```

ldd(1) becomes a convenient tool for discovering whether lazy loading might be applicable. Suppose we rebuilt wally.so without the -z lazyload option. And recall from "Dependencies – Define What You Need, and Nothing Else" on page 244 that the -u option can be used to discover unused dependencies.

```
% cc -o wally.so wally.c -G -Kpic -zdefs -R'$ORIGIN' foo.so bar.so
% ldd -Ldu wally.so

        foo.so =>        ./foo.so
        bar.so =>        ./bar.so

      unused object=./foo.so
      unused object=./bar.so
```

This has revealed that loading wally.so and relocating it as would occur at process startup, did not require the dependencies foo.so or bar.so to be loaded. This confirms that these two dependencies can be lazily loaded when reference to them is first made.

Lazy loading can be observed at runtime using the runtime linkers debugging capabilities (LD_DEBUG=files). For example, if wally() was called with a zero argument, we'd see bar.so lazily loaded.

```
% LD_DEBUG=files main
   .....
   25670: 1: transferring control: ./main
   .....
   25608: 1: file=bar.so;  lazy loading from file=./wally.so: symbol=bar
   .....
```

Note, not only does lazy loading have the potential of reducing the cost of start-up processing, but if lazy loading references are never called, the dependencies will never be loaded as part of the process.

http://blogs.sun.com/rie/entry/dependencies_perhaps_they_can_be

Lazy Loading – There's Even a Fallback

Rod Evans, August 1, 2004

In "Dependencies – Perhaps They Can Be Lazily Loaded" on page 245, I described the use of lazy loading. Of course, when we initially played with an implementation of this technology, a couple of applications immediately fell over. It turns out that a fallback was necessary.

Let's say an application developer creates an application with two dependencies. The developer wishes to employ lazy loading for both dependencies.

```
% ldd main
        foo.so =>        ./foo.so
        bar.so =>        ./bar.so
        ...
```

The application developer has no control over the dependency `bar.so`, as this dependency is provided by an outside party. In addition, this shared object has its own dependency on `foo.so`, however it does *not* express the required dependency information. If we were to inspect this dependency, we would see that it is not `ldd` clean.

```
% ldd -r bar.so
        symbol not found: foo     (./bar.so)
```

The only reason this library has been successfully employed by any application is because the application, or some other shared object within the process, has made the dependency `foo.so` available. This is probably more by accident than design, but sadly it is an all too common occurrence.

Now, suppose the application `main` makes reference to a symbol that causes the lazy loading of `bar.so` *before* the application makes reference to a symbol that would cause the lazy loading of `foo.so` to occur.

```
% LD_DEBUG=bindings,symbols,files main
    .....
    07683: 1: transferring control: ./main
    .....
    07683: 1: file=bar.so;  lazy loading from file=./main: symbol=bar
    .....
    07683: 1: binding file=./main to file=./bar.so: symbol ‘bar’
```

When control is passed to `bar()`, the reference it makes to its implicit dependency `foo()` is not going to be found, because the shared object `foo.so` is not yet available. Because this scenario is so common, the runtime linker provides a fallback. If a symbol cannot be found and lazy loadable dependencies are still pending, the runtime linker will process these pending dependencies in a final attempt to locate the symbol. This can be observed from the remaining debugging output.

```
    07683: 1: symbol=foo;  lookup in file=./main  [ ELF ]
    07683: 1: symbol=foo;  lookup in file=./bar.so  [ ELF ]
    07683: 1:
    07683: 1: rescanning for lazy dependencies for symbol: foo
    07683: 1:
    07683: 1: file=foo.so;  lazy loading from file=./main: symbol=foo
    .....
    07683: 1: binding file=./bar.so to file=./foo.so: symbol ‘foo’
```

Of course, there can be a down side to this fallback. If `main` were to have many lazy loadable dependencies, each will be processed until `foo()` is found. Thus, several dependencies may get loaded that aren't necessary. The use of lazy loading is never going to be more expensive than non-lazy loading, but if this fallback mechanism has to kick in to find implicit dependencies, the advantage of lazy loading is going to be compromised.

To prevent lazy loading from being compromised, always record those dependencies you need (as described in “Dependencies – Define What You Need, and Nothing Else” on page 244 (and nothing else).

http://blogs.sun.com/rie/entry/lazy_loading_there_s_even

Finding Symbols – Reducing dlsym() Overhead

Rod Evans, September 09, 2005

In "Lazy Loading – There's Even a Fallback" on page 247, I explained how lazy loading provides a fallback mechanism. If a symbol search exhausts all presently loaded objects, any pending lazy-loaded objects are processed to determine whether the required symbol can be found. This fall back is required as many dynamic objects exist that do not define all their dependencies. These objects have (probably unknowingly) become reliant on other dynamic objects making available the dependencies they need. Dynamic object developers should define what they need and nothing else (as described in "Dependencies – Define What You Need, and Nothing Else" on page 244).

dlsym(3C) can also trigger a lazy load fall back. You can observe such an event by enabling the runtime linker's diagnostics (as described in "Tracing a Link-Edit" on page 257). Here, we're looking for a symbol in libelf from an application that has a number of lazy dependencies.

```
% LD_DEBUG=symbols,files,bindings  main
    .....
    19231: symbol=elf_errmsg;  dlsym() called from file=main  [ RTLD_DEFAULT ]
    19231: symbol=elf_errmsg;  lookup in file=main  [ ELF ]
    19231: symbol=elf_errmsg;  lookup in file=/lib/libc.so.1  [ ELF ]
    19231:
    19231: rescanning for lazy dependencies for symbol: elf_errmsg
    19231:
    19231: file=libnsl.so.1;  lazy loading from file=main: symbol=elf_errmsg
    ......
    19231: file=libsocket.so.1;  lazy loading from file=main: symbol=elf_errmsg
    ......
    19231: file=libelf.so.1;  lazy loading from file=main: symbol=elf_errmsg
    ......
    19231: symbol=elf_errmsg;  lookup in file=/lib/libelf.so.1  [ ELF ]
    19231: binding file=main to file=/lib/libelf.so.1: symbol 'elf_errmsg'
```

Exhaustively loading lazy dependencies to resolve a symbol isn't always what you want. This is especially true if the symbol may not exist. In Solaris 10 we added RTLD_PROBE. This flag results in the same lookup semantics as RTLD_DEFAULT, but does not fall back to an exhaustive loading of pending lazy objects. This handle can be thought of as the light weight version of RTLD_DEFAULT.

Therefore, if we wanted to test for the existence of a symbol within the objects that were presently loaded within a process, we could use dlsym() to probe the process:

```
% LD_DEBUG=symbols,files,bindings  main
    .....
    19251: symbol=doyouexist;  dlsym() called from file=main  [ RTLD_PROBE ]
    19251: symbol=doyouexist;  lookup in file=main  [ ELF ]
```

```
19251: symbol=doyouexist;  lookup in file=/lib/libc.so.1  [ ELF ]
......
19251: ld.so.1: main: fatal: doyouexist: can't find symbol
```

When `dlsym()` is used to locate symbols from a handle returned by `dlopen`(3C), all the dependencies associated with the handle are available to the symbol lookup. I always thought this was rather odd, and that `dlsym()` should only look at the initial object of a handle. In other words, if you:

```
if ((handle = dlopen("foo.so", RTLD_LAZY)) != NULL) {
        fprt = dlsym(handle, "foo");
```

then intuitively the search for `foo` would be isolated to `foo.so`, and not include the multitude of dependencies also brought in by `foo.so`. But, that is not how our founding fathers developed `dlsym()`. I think we even considered changing the behavior once so that `dlsym()` would only search the initial object. But we soon found that a number of applications fell over as they could no longer find the symbols they were used to finding.

In Solaris 9 8/03 we provided an extension to `dlopen()` with the new flag `RTLD_FIRST`. By using this flag the same series of objects are opened and associated with a handle. However, only the first object on the handle is made available for `dlsym()` searches.

Perhaps `RTLD_PROBE` and `RTLD_FIRST` can reduce your `dlsym()` overhead.

http://blogs.sun.com/rie/entry/finding_symbols_reducing_dlsym_overhead

Using and Redistributing Sun Studio Libraries in an Application

Steve Clamage and Darryl Gove, May 2008

Introduction

The Sun Studio software suite provides a number of libraries that can be incorporated into an application to provide functionality and reduce development time. These libraries are redistributable, meaning that they can be freely distributed along with the applications that depend on them. The full list of Sun Studio 12 redistributable libraries and files can be found at http://docs.sun.com/source/820-4155/runtime.libraries.html

This article presents the best practices for redistributing these libraries and for maintaining applications that depend on them.

The Problem

There are two ways that an application can be linked to a library:

- *Static* linking combines code from the library with the executable forming a single file. The advantage is that the library version used at runtime can be known explicitly, and cannot be changed. The disadvantage is that if the library has to be modified, for example to fix a bug, then the whole application has to be relinked and a new version provided.
- *Dynamic* linking assumes the library is external to the application and will be loaded by the application at runtime. The advantages here are that the library can be updated or replaced without having to recompile or relink the application, and if multiple processes use the same library, only a single copy will be shared between them, thereby reducing the physical memory used. But there are some disadvantages: if multiple copies of a library resides on a system, it is possible for an application to choose the wrong library to load; and should a library used by an application be updated while the application is running, the behaviour could be unpredictable.

Dynamic linking is preferred, especially for the libraries supplied by the compiler, as well as for user libraries that are required by the application. This enables better management of updates to the library code.

Example

This code example uses the vector class from the C++ Standard Library

```
#include <vector>

int main()
{
   std::vector<int> v;
   v.push_back(10);
   return(0);
}
```

By default the compiler will use the C++ Standard Library, `libCstd`, that is provided as part of the Solaris OS. The library is installed in `/usr/lib` as part of the OS, and hence does not need to be packaged separately. Use the `ldd` utility to determine which libraries the application will link to, as shown below.

The following example uses the default standard library.

```
% CC v.cc
% ldd a.out
        libCstd.so.1 =>  /usr/lib/libCstd.so.1
        libCrun.so.1 =>  /usr/lib/libCrun.so.1
        libm.so.1 =>     /usr/lib/libm.so.1
        libc.so.1 =>     /usr/lib/libc.so.1
        libdl.so.1 =>    /usr/lib/libdl.so.1
        /usr/lib/cpu/sparcv8plus/libCstd_isa.so.1
        /usr/platform/SUNW,Sun-Fire-880/lib/libc_psr.so.1
```

The developer can decide to use the alternative `stlport4` library, which provides better standards conformance, and often better performance, than the default library. This library is not shipped as part of the Solaris OS, but is part of the Sun Studio distribution.

The next example shows the same code compiled with the flag `-library=stlport4`, which tells the compiler to use the `stlport4` library instead of the default `libCstd`. The output from `ldd` shows that the application links in the library located in the directory that is part of the Sun Studio distribution.

```
% CC -library=stlport4 v.cc
% ldd a.out
        libstlport.so.1 =>       /opt/SUNWspro/lib/stlport4/libstlport.so.1
        librt.so.1 =>    /usr/lib/librt.so.1
        libCrun.so.1 =>  /usr/lib/libCrun.so.1
        libm.so.1 =>     /usr/lib/libm.so.1
        libc.so.1 =>     /usr/lib/libc.so.1
        libaio.so.1 =>   /usr/lib/libaio.so.1
        libmd5.so.1 =>   /usr/lib/libmd5.so.1
        libdl.so.1 =>    /usr/lib/libdl.so.1
        /usr/platform/SUNW,Sun-Fire-880/lib/libc_psr.so.1
        /usr/platform/SUNW,Sun-Fire-880/lib/libmd5_psr.so.1
```

Running this program on another system without the Sun Studio software installed, the linker will not be able to locate the file `libstlport.so.1`. This situation is shown in the next example with the output from `ldd` on a system without Sun Studio installed.

```
% ldd a.out

        libstlport.so.1 =>       (file not found)
        librt.so.1 =>    /usr/lib/librt.so.1
        libCrun.so.1 =>  /usr/lib/libCrun.so.1
        libm.so.1 =>     /usr/lib/libm.so.1
        libc.so.1 =>     /usr/lib/libc.so.1
        libaio.so.1 =>   /usr/lib/libaio.so.1
        libmd.so.1 =>    /usr/lib/libmd.so.1
        libm.so.2 =>     /usr/lib/libm.so.2
        /platform/SUNW,Sun-Fire-T200/lib/libc_psr.so.1
        /platform/SUNW,Sun-Fire-T200/lib/libmd_psr.so.1
```

One workaround for this situation that is too frequently adopted is to set the environment variable `LD_LIBRARY_PATH` to point to the directory containing the missing library. Although this does work, it is not a recommended fix because it is fragile and requires the user's environment to be correctly set.

For example, the library `libstlport.so.1` might have been copied into the current directory and `LD_LIBRARY_PATH` set to "." (dot). This is shown in the next example. But if the application is invoked from any directory other than the current directory, the application will not be able to locate the `stlport4` library.

```
% cd test
% export LD_LIBRARY_PATH=.
% ldd a.out
```

```
        libstlport.so.1 =>       ./libstlport.so.1
        librt.so.1 =>    /usr/lib/librt.so.1
        libCrun.so.1 =>  /usr/lib/libCrun.so.1
        libm.so.1 =>     /usr/lib/libm.so.1
        libc.so.1 =>     /usr/lib/libc.so.1
        libaio.so.1 =>   /usr/lib/libaio.so.1
        libmd.so.1 =>    /usr/lib/libmd.so.1
        libm.so.2 =>     /usr/lib/libm.so.2
        /platform/SUNW,Sun-Fire-T200/lib/libc_psr.so.1
        /platform/SUNW,Sun-Fire-T200/lib/libmd_psr.so.1
% cd ..
% ldd test/a.out
        libstlport.so.1 =>       (file not found)
        librt.so.1 =>    /usr/lib/librt.so.1
        libCrun.so.1 =>  /usr/lib/libCrun.so.1
        libm.so.1 =>     /usr/lib/libm.so.1
        libc.so.1 =>     /usr/lib/libc.so.1
        libaio.so.1 =>   /usr/lib/libaio.so.1
        libmd.so.1 =>    /usr/lib/libmd.so.1
        libm.so.2 =>     /usr/lib/libm.so.2
        /platform/SUNW,Sun-Fire-T200/lib/libc_psr.so.1
        /platform/SUNW,Sun-Fire-T200/lib/libmd_psr.so.1
```

The `LD_LIBRARY_PATH` environment variable is useful for special testing, but is not a scalable or maintainable solution for deployed programs. Rod Evans provides a detailed discussion in his blog entry "`LD_LIBRARY_PATH` – Just Say No" on page 243.

The Right Way to Distribute Shared Libraries

Avoid using the `LD_LIBRARY_PATH` environment variable by packaging the application binary along with any additional libraries in a directory structure as shown in the next example.

```
/application
            /bin       Contains executables
            /lib       Contains necessary libraries
```

In the example, the `stlport.so.1` library would be copied into the `/lib` subdirectory. The compiler flag `-library=stlport4` will enable linking the `stlport4` library rather than the default library at build time. Compiling with the `-R dir` option, the linker will locate the library in the application's `/lib` subdirectory at runtime.

Although you could specify an absolute directory path to search for the library, this would restrict the installation to one specific location in the file system, again requiring use of `LD_LIBRARY_PATH` as an ugly workaround. The better approach is to use the token `$ORIGIN` with the `-R` option to tell the application to look in a path relative to the location of the executable. The `$ORIGIN` token may need special treatment to avoid being interpreted by the shell. This is shown in the next example.

```
% CC -library=stlport4 -R'$ORIGIN/../lib' v.cc
```

In a Makefile, an extra `$` escape is needed to avoid `$ORIGIN` being interpreted as a Make variable. This is shown in the next example.

```
v: v.cc
       CC -library=stlport4 -R \$$ORIGIN/../lib v.cc -o v
```

On the target machine this results in the application locating the library in the application's `/lib` directory, as shown in the next example.

```
% ldd a.out
        libstlport.so.1 =>       /export/home/test/bin/../lib/libstlport.so.1
        librt.so.1 =>    /usr/lib/librt.so.1
        libCrun.so.1 =>  /usr/lib/libCrun.so.1
        libm.so.1 =>     /usr/lib/libm.so.1
        libc.so.1 =>     /usr/lib/libc.so.1
        libaio.so.1 =>   /usr/lib/libaio.so.1
        libmd.so.1 =>    /usr/lib/libmd.so.1
        libm.so.2 =>     /usr/lib/libm.so.2
        /platform/SUNW,Sun-Fire-T200/lib/libc_psr.so.1
        /platform/SUNW,Sun-Fire-T200/lib/libmd_psr.so.1
```

Conclusions

Using the `$ORIGIN` token with the `-R` option to locate the libraries on a path relative to the executable is recommended for the following reasons:

- The executable and libraries can be co-located, which ensures that the executable is distributed with and uses the appropriate library version.
- If the support libraries are updated, it is easy to copy over the new updated version to replace an earlier version.
- Each library is used by one application (or one family of applications), so this version of the library can be updated without risk to other applications installed on the system.
- The application and libraries can be installed anywhere and be expected to work without the user having to use workarounds like `LD_LIBRARY_PATH`.
- The technique described here applies equally well to third-party shared libraries or libraries created as part of the application.

`http://docs.sun.com/source/820-5191/stdlibdistr.html`

Dynamic Object Versioning

Rod Evans, August 22, 2004

For some time now, we've been versioning core system libraries. You can display version definitions, and version requirements with `pvs`(1). For example, the latest version of `libelf.so.1` from Solaris 10 provides the following versions:

```
% pvs -d /lib/libelf.so.1
        libelf.so.1;
        SUNW_1.5;
```

```
        SUNW_1.4;
        ....
        SUNWprivate_1.1;
```

So, what do these versions provide? Shared object versioning has often been established with various conventions of renaming the file itself with different major or minor (or micro) version numbers. However, as applications have become more complex, specifically because they are constructed from objects that are asynchronously delivered from external partners, this file naming convention can be problematic.

In developing the core Solaris libraries, we've been rather obsessed with compatibility, and rather than expect customers to rebuild against different shared object file names (`libfoo.so.1`, and later `libfoo.so.2`), we've maintained compatibility by defining fixed interface sets within the same library file. And, the only changes we've made to the library is to add new interface sets. These interface sets are described by version names.

Now you could maintain compatibility by retaining all existing public interfaces, and only adding new interfaces, without the versioning scheme. However, the version scheme has a couple of advantages:

- consumers of the interface sets record their requirements on the version name they reference.
- establishing interface sets removes unnecessary interfaces from the namespace.
- the version sets provide a convenient means of policing interface evolution.

When a consumer references a versioned shared object, the version name representing the interfaces the consumer references are recorded. For example, an application that references the `elf_getshnum`(3ELF) interface from `libelf.so.1` will record a dependency on the `SUNW_1.4` version:

```
% cc -o main main.c -lelf
% pvs -r main
    libelf.so.1 (SUNW_1.4);
```

This version name requirement is verified at runtime. Therefore, should this application be executed in an environment consisting of an older `libelf.so.1`, one that perhaps only offers version names up to `SUNW_1.3`, then a fatal error will result when `libelf.so.1` is processed:

```
% pvs -dn /lib/libelf.so.1
    SUNW_1.3;
        SUNWprivate_1.1;
% main
ld.so.1: ./main: fatal: libelf.so.1: version ‘SUNW_1.4’ not found \
    (required by file ./main)
```

This verification might seem simplistic, and won't the application be terminated anyway if a required interface can't be located? Well yes, but function binding normally occurs at the time the function is first called. And this call can be some time after an application is started (think scientific applications that can run for days or weeks). It is far better to be informed that an

interface can't be located when a library is first loaded, than for an application to be killed some time later when a specific interface can't be found.

Defining a version typically results in the demotion of many other global symbols to local scope. This localization can prevent unintended symbol collisions. For example, most shared objects are built from many relocatable objects, each referencing one another. The interface that the developer wishes to export from the shared object is normally a subset of the number of global symbols that would normally remain visible.

Version definitions can be defined using a `mapfile`. For example, the following `mapfile` defines a version containing two interfaces. Any other global symbols that would normally be made available by the objects that contribute to the shared object are demoted, and hence hidden as locals:

```
% cat mapfile
    ISV_1.1 {
        global:
            foo1();
            foo2();
        local:
            *;
    };
 % cc -o libfoo.so.1 -G -Kpic -Mmapfile foo.c bar.c ...
 % pvs -dos libfoo.so.1
    libfoo.so.1 -       ISV_1.1: foo1;
    libfoo.so.1 -       ISV_1.1: foo2;
```

The demotion of unnecessary global symbols to locals greatly reduces the relocation requirements of the object at runtime, and can significantly reduce the runtime startup cost of loading the object.

Of course, interface compatibility requires a disciplined approach to maintaining interfaces. In the previous example, should the signature of `foo1()` be changed, or `foo2()` be deleted, then the use of a version name is meaningless. Any application that had built against the original interfaces will fail at runtime when the new library is delivered, even though the version name verification will have been satisfied.

With the core Solaris libraries we maintain compatibility as we evolve through new releases by maintaining existing public interfaces and only adding new version sets. Auditing of the version sets help catch any mistaken interface deletions or additions. Yeah, we fall foul of cut-and-paste errors too :-)

For more information on versioning refer to the *Linker and Libraries Guide* appendix "Versioning Quick Reference" (`http://docs.sun.com/app/docs/doc/817-1984/appendixb-45356?a=view`). Or, for a detailed description, refer to the chapter "Application Binary Interfaces and Versioning" (`http://docs.sun.com/app/docs/doc/817-1984/chapter5-84101?a=view`).

`http://blogs.sun.com/rie/entry/dynamic_object_versioning`

Tracing a Link-Edit

Rod Evans, September 29, 2004

Since Solaris 2.0, the link-editors have provided a mechanism for tracing what they're doing. As this mechanism has been around for so long, plus I've used some small examples in previous postings, I figured most folks knew of its existence. I was reminded the other day that this isn't the case. For those of you unfamiliar with this tracing, here's an introduction, plus a glimpse of a new analysis tool available with Solaris 10.

You can set the environment variable `LD_DEBUG` to one or more pre-defined tokens. This setting causes the runtime linker, `ld.so.1`(1), to display information regarding the processing of any application that inherits this environment variable. The special token `help` provides a list of token capabilities without executing any application.

One of the most common tracing selections reveals the binding of a symbol reference to a symbol definition.

```
% LD_DEBUG=bindings  main
    .....
    00966: binding file=main to file=/lib/libc.so.1 symbol ‘_iob’
    .....
    00966: binding file=/lib/libc.so.1 to file=main: symbol ‘_end’
    .....
    00966: 1: transferring control: main
    .....
    00966: 1: binding file=main to file=/lib/libc.so.1: symbol ‘atexit’
    .....
    00966: 1: binding file=main to file=/lib/libc.so.1: symbol ‘exit’
```

Those bindings that occur before transferring to `main` are the immediate (data) bindings. These bindings must be completed before any user code is executed. Those bindings that occur after the transfer to `main` are established when the associated function is first called. These are lazy bindings.

Another common tracing selection reveals what files are loaded.

```
% LD_DEBUG=files  main
    .....
    16763: file=libc.so.1;  needed by main
    16763: file=/lib/libc.so.1  [ ELF ]; generating link map
    .....
    16763: 1: transferring control: ./main
    .....
    16763: 1: file=/lib/libc.so.1;  \
        filter for /platform/$PLATFORM/lib/libc_psr.so.1
    16763: 1: file=/platform/SUNW,Sun-Blade-1000/lib/libc_psr.so.1;  \
        filtered by /lib/libc.so.1
    16763: 1: file=/platform/SUNW,Sun-Blade-1000/lib/libc_psr.so.1  [ ELF ]; \
        generating link map
    .....
    16763: 1: file=libelf.so.1;  dlopen() called from file=./main \
```

```
        [ RTLD_LAZY  RTLD_LOCAL  RTLD_GROUP  RTLD_WORLD ]
    16763: 1: file=/lib/libelf.so.1  [ ELF ]; generating link map
```

This reveals initial dependencies that are loaded prior to transferring control to `main`. It also reveals objects that are loaded during process execution, such as filters and `dlopen`(3C) requests.

Note, the environment variable `LD_DEBUG_OUTPUT` can be used to specify a file name to which diagnostics are written (the file name gets appended with the pid). This is helpful to prevent the tracing information from interfering with normal program output, or for collecting large amounts of data for later processing.

In "Dependencies – Define What You Need, and Nothing Else" on page 244, I described how you could discover unused, or unreferenced dependencies. You can also discover these dependencies at runtime.

```
% LD_DEBUG=unused  main
    .....
    11143: 1: file=libWWW.so.1  unused: does not satisfy any references
    11143: 1: file=libXXX.so.1  unused: does not satisfy any references
    .....
    11143: 1: transferring control: ./main
    .....
```

Unused objects are determined prior to calling `main` and after any objects are loaded during process execution. The two libraries above aren't referenced before `main`, and thus make ideal lazy-loading candidates (that's if they are used at all).

Lastly, there are our old friends `.init` sections. Executing these sections in an attempt to fulfill the expectations of modern languages (I'm being polite here) and expected programming techniques has been, shall we say, challenging. `.init` tracing is produced no matter what debugging token you chose.

```
% LD_DEBUG=basic  main
    .....
    34561: 1: calling .init (from sorted order): libYYY.so.1
    34561: 1: calling .init (done): libYYY.so.1
    .....
    34561: 1: calling .init (from sorted order): libZZZ.so.1
    .....
    34561: 1: calling .init (dynamically triggered): libAAA.so.1
    34561: 1: calling .init (done): libAAA.so.1
    .....
    34561: 1: calling .init (done): libZZZ.so.1
```

Note that in this example, the topologically sorted order established to fire `.init`'s has been interrupted. We dynamically fire the `.init` of `libAAA.so.1` that has been bound to while running the `.init` of `libZZZ.so.1`. Try to avoid this. I've seen bindings cycle back into dependencies whose `.init` hasn't completed.

The debugging library that provides these tracing diagnostics is also available to the link-editor, `ld`(1). This debugging library provides a common diagnostic format for tracing both linkers. Use the link-editor's `-D` option to obtain tracing info. As most compilers have already laid claim to this option, the `LD_OPTIONS` environment variable provides a convenient setting. For example, to see all the gory details of the symbol resolution undertaken to build an application, try:

```
% LD_OPTIONS=symbols,detail cc -o main $(OBJS) $(LIBS) ...
```

and stand back ... the output can be substantial.

Although tracing a process at runtime can provide useful information to help diagnose process bindings, the output can be substantial. Plus, it only tells you what bindings have occurred. This information lacks the full symbolic interface data of each object involved, which in turn can hide what you think *should* be occurring. In Solaris 10, we added a new utility, `lari`(1), which provides the Link Analysis of Runtime Interfaces.

This `perl`(1) script analyzes a debugging trace, together with the symbol tables of each object involved in a process. `lari`(1) tries to discover any *interesting* symbol relationships. "Interesting" typically means that a symbol name exists in more than one dynamic object, and interposition is at play. Interposition can be your friend, or your enemy - `lari`(1) doesn't know which. But historically, a number of application failures or irregularities have boiled down to some unexpected interposition which at the time was hard to track down.

For example, a typical interposition might show up as:

```
% lari  main
   [2:3]: foo(): /opt/ISV.I/lib/libfoo.so.1
   [2:0]: foo(): /opt/ISV.II/lib/libbar.so.1
   [2:4]: bar[0x80]: /opt/ISV.I/lib/libfoo.so.1
   [2:0]: bar[0x100]: /opt/ISV.II/lib/libbar.so.1
```

Here, two versions of function `foo()`, and two versions of the data item `bar[]` exist. With interposition, all bindings have resolved to the first library loaded. Hopefully the 3 callers of `foo()` expect the signature and functionality provided by `ISV.I`. But you have to wonder, do the 4 users of `bar[]` expect the array to be `0x80` or `0x100` in size?

`lari`(1) also uncovers direct bindings or symbols that are defined with protected visibility. These can result in multiple instances of a symbol being bound to from different callers:

```
% lari  main
   [2:1D]: foo(): ./libA.so
   [2:1D]: foo(): ./libB.so
```

Again, perhaps this is what the user wants to achieve, perhaps not .. but it is interesting.

There are many more permutations of symbol diagnostic that can be produced by `lari`(1), including the identification of explicit interposition (such as preloading, or objects built with `-z interpose`), copy relocations, and `dlsym`(3C) requests. Plus, as `lari`(1) is effectively

discovering the interfaces used by each object within a process, it can create versioning `mapfiles` that can be used as templates to rebuild each object.

http://blogs.sun.com/rie/entry/tracing_a_link_edit

Shared Object Filters

Rod Evans, October 22, 2004

Filters are a class of shared objects that are available with the Solaris OS. These objects allow for the redirection of symbol bindings at runtime. They have been employed to provide standards interfaces and to allow the selection of optimized function implementations at runtime. For those of you unfamiliar with filters, here's an introduction, plus a glimpse of some new techniques available with Solaris 10.

Filters can exist in two forms, standard filters and auxiliary filters. At runtime, these objects redirect bindings from themselves to alternative shared objects that are known as `filtees`. Shared objects are identified as filters by using the link-editor's `-F` and `-f` options.

Standard filters provide no implementation for the interfaces they define. Essentially they provide a symbol table. When used to build a dynamic object, they satisfy the symbol resolution requirements of the link-editing process. However at runtime, standard filters redirect a symbol binding from themselves to a filtee.

Standard filters have been used to implement `libdl.so.1`, the library that offers the dynamic linking interfaces. This library has no implementation details; it simply points to the real implementation within the runtime linker:

```
% elfdump -d /usr/lib/libdl.so.1

    Dynamic Section:  .dynamic
      index  tag       value
      [0]  SONAME      0x138  libdl.so.1
      [1]  FILTER      0x143  /usr/lib/ld.so.1
      ...
```

Auxiliary filters work in much the same way, however, if a filtee implementation can't be found, the symbol binding is satisfied by the filter itself. Basically, auxiliary filters allow an interface to find a better alternative. If an alternative is found it will be used, and if not, the generic implementation provided by the filter provides a fallback.

Auxiliary filters have been used to provide platform specific implementations. Typically, these are implementations that provide a performance improvement on various platforms. `libc.so.1` has used this technique to provide optimized versions of the `memcpy()` family of routines:

```
% elfdump -d /usr/lib/libc.so.1

    Dynamic Section:  .dynamic
```

```
index  tag        value
...
[3]  SONAME      0x6280  libc.so.1
[4]  AUXILIARY   0x628a  /usr/platform/$PLATFORM/lib/libc_psr.so.1
...
```

You can observe that a symbol binding has been satisfied by a filtee by using the runtime linker's tracing. The following output shows the symbol `memset` being searched for in the application `date`, the dependency `libc.so.1`, and then in `libc`'s filtee, `libc_psr.so.1`.

```
% LD_DEBUG=symbols,bindings  date

    .....
    11055: symbol=memset;  lookup in file=/usr/bin/date
    11055: symbol=memset;  lookup in file=/usr/lib/libc.so.1
    11055: symbol=memset;  lookup in file=/usr/platform/SUNW,Sun-Fire/lib/libc_psr.so.1
    11055: binding file=/usr/lib/libc.so.1 to \
        file=/usr/platform/SUNW,Sun-Fire/lib/libc_psr.so.1: symbol ‘memset’
    .....
```

Until now, a filter has been an identification applied to a whole shared object. With Solaris 10, per-symbol filtering has been introduced. This allows individual symbol table entries to identify themselves as standard or auxiliary filters. These filters provide greater flexibility, together with less runtime overhead than whole object filters. Individual symbols are identified as filters using `mapfile` entries at the time an object is built.

For example, `libc.so.1` now provides a number of per-symbol filters. Each filter is defined using a `mapfile` entry:

```
% cat mapfile
   SUNW_1.22 {
       global:
            ....
            dlopen = FUNCTION FILTER /usr/lib/ld.so.1;
            dlsym = FUNCTION FILTER /usr/lib/ld.so.1;
            ....
   };
   ....
   SUNW_0.7 {
      global:
           ....
           memcmp = AUXILIARY /platform/$PLATFORM/lib/libc_psr.so.1;
           memcpy = AUXILIARY /platform/$PLATFORM/lib/libc_psr.so.1;
           ....
   };
```

The above definitions provide a couple of advantages. First, you no longer need to link against `libdl` to obtain the dynamic linking family of routines. Second, the overhead of searching for a filtee will only occur if a search for the associated symbol is requested. With whole object filters, any reference to a symbol within the filter would trigger a filtee lookup, plus every interface offered by the filter would be searched for within the filtee.

Per-symbol filters have also proved useful in consolidating existing interfaces. For example, for historic standards compliance, `libc.so.1` has offered a small number of math routines.

However, the full family of math routines are provided in `libm.so.2`, the library that most math users link with. Providing the small family of duplicate math routines in `libc` was a maintenance burden, plus there was always the chance of them getting out of sync. With per-symbol filtering, the `libc` interfaces are maintained, while pointing at the one true implementation. `elfdump`(1) can be used to reveal the filter symbols (F) offered by a shared object:

```
% elfdump -y /lib/libc.so.1

   Syminfo Section:  .SUNW_syminfo
     index  flgs          bound to           symbol
      ....
      [93]  F         [1] libm.so.2          isnand
      [95]  F         [1] libm.so.2          isnanf
      ....
```

The definition of a filtee has often employed runtime tokens such as `$PLATFORM`. These tokens are expanded to provide pathnames specific to the environment in which the filter is found. A new capability provided with Solaris 10 is the `$HWCAP` token. This token is used to identify a directory in which one or more hardware capability libraries can be found.

Shared objects can be built to record their hardware capabilities requirements. Filtees can be constructed that use various hardware capabilities as a means of optimizing their performance. These filtees can be collected in a single directory. The `$HWCAP` token can then be employed by the filter to provide the selection of the optimal filtee at runtime:

```
% elfdump -H /opt/ISV/lib/hwcap/*

   /opt/ISV/lib/hwcap/libfoo_hwcap1.so.1:

   Hardware/Software Capabilities Section:  .SUNW_cap
     index  tag              value
       [0]  CA_SUNW_HW_1     0x869  [ SSE  MMX  CMOV  SEP  FPU ]

   /opt/ISV/lib/hwcap/libfoo_hwcap2.so.1:

   Hardware/Software Capabilities Section:  .SUNW_cap
     index  tag              value
      [0]  CA_SUNW_HW_1     0x1871  [ SSE2  SSE  MMX  CMOV  AMD_SYSC  FPU ]

   % elfdump -d /opt/ISV/lib/libfoo.so.1

    Dynamic Section:  .dynamic
      index  tag       value
      ...
      [3]  SONAME      0x138  libfoo.so.1
      [4]  AUXILIARY   0x4124 /opt/ISV/lib/hwcap/$HWCAP
      ...
```

The hardware capabilities of a platform are conveyed to the runtime linker from the kernel. The runtime linker then matches these capabilities against the requirements of each filtee. The filtees are then sorted in descending order of their hardware capability values. These sorted filtees are used to resolve symbols that are defined within the filter.

This model of file identification provides greater flexibility than the existing use of `$PLATFORM`, and is well suited to filtering use.

As usual, examples and all the gory details of filters and the new techniques outlined above can be found in the *Linker and Libraries Guide* (`http://docs.sun.com/app/docs/doc/817-1984`).

Discussions on how to alter any hardware capabilities established by the compilers can be found in "Reading or Modifying Hardware Capabilities Information" on page 263, and in "Modifying Hardware Capabilities" (`http://developers.sun.com/solaris/articles/hwcap_modification.html`).

`http://blogs.sun.com/rie/entry/shared_object_filters`

Reading or Modifying Hardware Capabilities Information

Alfred Huang, January 13, 2006

Ever experienced the following case before? You created an app targeted for certain "newer" platforms, then one day came across an "older" machine and incautiously tried to run your app on it. What happened? When a new instruction of the newer machine was executed on an older one, the older one did not recognize it and simply ended with a rude message from the OS, "Invalid Instruction". At this stage, it may take some effort to figure out the culprit.

Solaris 10 has a new feature, the Hardware Capabilities checking mechanism which stops this type of occurrence with the runtime linker before running it.

Let's say we have an object file `pack.o` which contains x86's SSE2 and AMD's 3DNow instructions. You can check it using a `file` command:

```
% file pack.o
pack.o:         ELF 32-bit LSB relocatable 80386 Version 1 [SSE2 AMD_3DNow]
```

Apparently, any apps linking in `pack.o` will not be able to run under a non-AMD machine. The runtime linker under Solaris 10 will return an error:

```
% a.out
ld.so.1: a.out: fatal: hardware capability unsupported: 0x100  [ AMD_3DNow ]
```

The trick is that the compiler during the assembly phase puts out a section marking all the encountered instructions as listed in `/usr/include/sys/auxv_386.h`. The runtime linker will check the combined bits of all instructions against the underlying platform.

But exceptions are necessary. Some code is written to have different fragments of code relating to different platforms and a choice is made based on the runtime checking of the CPUID bits.

In this case a possible resolution may be

- Don't mark it with the compiler in the first place. You can achieve this by turning on the `-nH` option in the assembler:

```
% cc -S -xarch=sse2 file.c
% fbe -nH file.s        <-- file.o will have no marking
```

- Change the marking using the mapfile of the linker:

  For a relocatable object file:

```
% more mapfile
hwcap_1 = SSE2 OVERRIDE;
% cc -c -xarch=sse2 file.c -o file_pre.o
% ld -r -Mmapfile -o file.o file_pre.o

% file file_pre.o
  file_pre.o:   ELF 32-bit LSB relocatable 80386 Version 1 [SSE2 AMD_3DNow]
% file file.o
  file.o:       ELF 32-bit LSB relocatable 80386 Version 1 [SSE2]
```

  For a shared library or executable file:

```
% more mapfile
  hwcap_1 = SSE2 OVERRIDE;
% cc -Mmapfile -G -o file.so file.o

% file file.o
  file.o:    ELF 32-bit LSB relocatable 80386 Version 1 [SSE2 AMD_3DNow]
% file file.so
  file.so:   ELF 32-bit LSB dynamic lib 80386 Version 1 [SSE2],
     dynamically linked, not stripped
```

One last trick, if you have a `.s` assembly file, you can assemble it with `-showimap` for verbose instruction category output.

```
% fbe -showimap pack.s
  line 47: movdl => SSE2
  line 51: movdl => SSE2
  line 56: prefetchw => 3DNow
  line 57: movq => SSE2
     ...
```

http://blogs.sun.com/alblog/entry/ridding_or_modifying_hardware_capabilities

Interface Creation – Using the Compilers

Rod Evans, January 2, 2005

In "Dynamic Object Versioning" on page 254, I covered how the interface of a dynamic object could be established by defining the interface symbols within a `mapfile`, and feeding this file to the link-edit of the final object. Establishing an object's interface in this manner hides all non-interface symbols, making the object more robust and less vulnerable to symbol name space pollution. This symbol reduction also goes a long way to reducing the runtime relocation cost of the dynamic object.

This `mapfile` technique, while useful with languages such as C, can be challenging to exploit with languages such as C++. There are two major difficulties.

First, the link-editor only processes symbols in their mangled form. For example, even a simple interface such as:

```
void foo(int bar)
```

has a C++ symbolic representation of:

```
% elfdump -s foo.o
    ....
    [2]  .... FUNC GLOB  D   0 .text  __1cDfoo6Fi_v_
```

As no tool exists that can determine a symbol's mangled name other than the compilers themselves, trying to establish definitions of this sort within a `mapfile` is no simple task.

The second issue is that some interfaces created by languages such as C++ provide implementation details of the language itself. These implementation interfaces often must remain global within a group of similar dynamic objects, as one interface must *interpose* on all the others for the correct execution of the application. As users generally are not aware of what implementation symbols are created, they can blindly demote these symbols to local when applying any interface definitions with a `mapfile`. Even the use of linker options like `-B symbolic` are discouraged with C++, as these options can lead to implementation symbols being created that are non-interposable.

Thankfully, some recent ELF extension work carried out with various UNIX vendors has established a set of visibility attributes that can be applied to ELF symbol table entries. These attributes are maintained within the symbol entries `st_other` field, and are fully documented in the table "ELF Symbol Visibility" in the "Object File Format" chapter of the *Linker and Libraries Guide* (`http://docs.sun.com/app/docs/doc/817-1984`).

The compilers, starting with Sun ONE Studio 8, are now capable of describing symbol visibility. These definitions are then encoded in the symbol table, and used by `ld`(1) in a similar manner as reading definitions from a `mapfile`. Using a combination of code "definitions" and command-line options, you can now defined the runtime interface of a C++ object.

As with any interface definition technique, this compilation method can greatly reduce the number of symbols that would normally be employed in runtime relocations. Given the number and size of C++ symbols, this technique can produce runtime relocation reductions that far exceed those that would be found in similar C objects. In addition, as the compiler knows what implementation symbols must remain global within the final object, these symbols are given the appropriate visibility attribute to ensure their correct use.

Presently there are two recommendations for establishing an object's interface. The first is to define all interface symbols using the `__global` directive, and reduce all other symbols to local using the `-xldscope=hidden` compiler option. This model provides the most flexibility. All global symbols are interposable, and allow for any copy relocations[1] to be processed correctly.

The second model is to define all interface symbols using the `__symbolic` directive, and again reduce all other symbols to local using the `-xldscope=hidden` compiler option. Symbolic

symbols (also termed “protected”), are globally visible, but have been internally bound to. This means that these symbols do not require symbolic runtime relocation, but cannot be interposed upon, or have copy relocations against them.

In practice, I'd expect to see significant savings in the runtime relocation of any modules that used either model. However, the savings between using the `__global` or the `__symbolic` model may be harder to measure. In a nutshell, if you do not want a user to interpose upon your interfaces and don't export data items, you can probably go with `__symbolic`. If in doubt, stick with the more flexible use of `__global`.

The following examples uses C++ code that was furnished to me as being representative of what users may develop.

```
% cat interface.h

    class item {
    protected:
        item();
    public:
        virtual void method1() = 0;
        virtual void method2() = 0;
        virtual ~item();
    };

    extern item *make_item();

    % cat implementation.cc

    #include "interface.h"

    class __global item; /* Ensures global linkage for any
                            implicitly generated members. */

    item::item() { }
    item::~item() { }

    class item_impl : public item {
        void method1();
        void method2();
    };

    void item_impl::method1() { }
    void item_impl::method2() { }

    void helper_func() { }

    __global item *make_item() {
        helper_func();
        return new item_impl;
    }
```

All interface symbols have employed the `__global` attribute. Compiling this module with `-xldscope=hidden` reveals the following symbol table entries.

```
% elfdump -CsN.symtab implementation.so.1

    ...
    [31]  .... FUNC LOCL  H  0 .text  void helper_func()
    [32]  .... OBJT LOCL  H  0 .data  item_impl::__vtbl
    [35]  .... FUNC LOCL  H  0 .text  void item_impl::method1()
    [36]  .... FUNC LOCL  H  0 .text  void item_impl::method2()

    [54]  .... OBJT GLOB  D  0 .data  item::__vtbl
    [55]  .... FUNC GLOB  D  0 .text  item::item()
    [58]  .... FUNC GLOB  D  0 .text  item::~item #Nvariant 1()
    [59]  .... FUNC GLOB  D  0 .text  item*make_item()
    [61]  .... OBJT GLOB  D  0 .data  _edata
    [67]  .... FUNC GLOB  D  0 .text  item::~item()
    [77]  .... FUNC GLOB  D  0 .text  item::item #Nvariant 1()
```

Notice that the first 4 local (`LOCL`) symbols would normally have been defined as global without using the symbol definitions and compiler option. This is a simple example. As implementations get more complex, expect to see a larger fraction of symbols demoted to locals.

For a definition of other related compiler options, at least how they relate to C++, see "Linker Scoping" in the *C++ User's Guide* (`http://docs.sun.com/source/817-5070/Language_Extensions.html`) [2].

[1] Copy relocations are a technique employed to allow references from non-pic code to external data items, while maintaining the read-only permission of a typical text segment. This relocation use, and overhead, can be avoided by designing shared objects that do not export data interfaces.

[2] It's rumored the compiler folks are also working on `__declspec` and GCC `__attribute__` clause implementations. These should aid porting code and interface definitions from other platforms.

An Update - Sunday May 29, 2005

The following article by Giri Mandalika on this topic, "Reducing Symbol Scope With Sun Studio C/C++" on page 267, includes the `__declspec` implementation.

`http://blogs.sun.com/rie/entry/interface_creation_using_the_compilers`

Reducing Symbol Scope With Sun Studio C/C++

Giri Mandalika, March 22, 2006

Summary

Hiding non-interface symbols of a library within the library makes the library more robust and less vulnerable to symbol collisions from outside the library. This symbol scope reduction also improves the performance of an application by reducing the runtime relocation costs of the dynamic libraries. To indicate the appropriate linker scoping in a source program, you can now use language extensions built into the Sun Studio C/C++ compilers as described here.

Note – Definitions for footnoted italicized words can be found in "Glossary" on page 290.

Introduction

Until the release of the Sun Studio 8 compilers, linker mapfiles were the only way to change the default symbol processing by the linker. With the help of mapfiles, all non-interface[1] symbols of an object can be hidden within a load module[2], thereby making the object more robust and less vulnerable to symbol collisions. This symbol scope reduction helps improving the performance of an application by reducing the runtime relocation costs of the dynamic objects. The other reason for symbol scoping is to ensure that clients only use the intended interface to the library, and not the functions that are internal to the library.

The mapfile mechanism is useful with languages such as C, but difficult to exploit with languages such as C++. There are two major hurdles:

1. The link-editor[3] only processes symbols in their mangled form. For example, even a simple interface such as `void printstring(char *str)` has a C++ symbolic representation something like `__1cLprintstring6Fpc_v_`. As no tool exists that can determine a symbol's mangled name other than the compilers themselves, trying to establish definitions of this sort within a mapfile, is not a simple task.

   Also, changes to a function's signature, or to a `typedef` that a function signature uses, can invalidate the mapfile which was produced because the mangled name of the symbols could have changed. For versioned libraries, this invalidation is a good thing because the function signature has changed. The fact that the mapfiles survive changes in parameter types in C is a problem.
2. Compilers can generate some implicit symbol definitions. These implementation interfaces must often remain global within a group of similar dynamic objects, as one interface must interpose on all the others for the correct execution of the application. As users generally are not aware of what implementation symbols the compiler creates, they can blindly demote these symbols to local when applying any interface definitions with a mapfile.

We can avoid the specification problems in mapfiles by specifying the linker symbol scope within the source program. Sun Studio 8 introduced new syntax for specifying this scope, and a new option for controlling default scope behavior. With these new features, programmers do not need mapfiles for linker scoping.

There are reasons programmers might still need mapfiles. The primary reason is library versioning. For other reasons, see the *Linker and Libraries Guide* (`http://docs.sun.com/app/docs/doc/817-1984`) for the full list of mapfile capabilities. Compiler-assisted linker scoping also helps with construction of mapfiles, because the set of globally visible symbols in a candidate library becomes the actual set of globally visible symbols, and the only remaining task is to assign symbols to versions.

This article introduces linker scoping with simple examples, and outlines the benefits of this feature for developing end-user applications. All the content of this article is equally applicable to C and C++, unless otherwise specified. Note that the terms *shared library*, *dynamic library*, *load module* and *dynamic module* are used interchangeably throughout the article.

Linker Scoping

Sun added *linker scoping* as a language extension with the release of the Sun Studio 8 C/C++ compilers. Using this feature, the programmer can indicate the appropriate symbol scoping within the source program. This section briefly explains the need for such a feature.

Default Behavior of the Solaris Linker

With Solaris (and UNIX in general), external names (symbols) will have global scope by default, in a dynamic object. This is due to the fact that the static linker makes all symbols global in scope without linker scoping mechanism. That is, it puts all the symbols into the dynamic symbol table of the resulting binary object so other binary modules can access those symbols. Such symbols are called *external* or *exported* symbols

At program startup, the dynamic linker[4] (also referred to as the *runtime* linker) loads up all dynamic libraries specified at link time before starting execution of the application. Because shared libraries are not available to the executable until runtime, shared library calls get special treatment in executable objects. To do this, the dynamic linker maintains a linked list of the link maps in the address space of the executing process, one for each dynamically linked object. The symbol search mechanism traverses this list to bind the objects of an application. The Procedure Linkage Table (PLT) facilitates this binding.

Relocation Processing

The PLT can be used to redirect function calls between the executable and a shared object, or between different shared objects, and is purely an optimization strategy designed to permit lazy symbol resolution at runtime.

Once all the dependencies for the executable were loaded, the runtime linker updates the memory image of the executable and its dependencies to reflect the real addresses for data and function references. This is also known as *relocation processing*.

The dynamic relocations that the dynamic linker performs are only necessary for the global (sometimes referred to as *external* or *exported*) symbols. The static linker resolves references to local symbols (for example, names of static functions) statically when it links the binary. So, when an application is made out of dynamic libraries with the default global scoping, it will pay some penalty during application startup time and the performance may suffer during runtime due to the overhead of PLT processing.

A considerable amount of startup time is spent performing symbolic relocations[5]. Generally a lot more time is spent relocating symbols from dependency objects than relocating symbols from the executable itself. To gain noticeable reduction in startup time, we have to somehow decrease the amount of relocation processing.

As stated earlier, the dynamic linker maintains a linked list of the link maps in the memory of the executing process, one for each dynamically linked object. So, the symbol search mechanism requires the runtime linker to traverse the whole link-map list, looking in each object's symbol table to find the required symbol definition. This is known as a *symbolic relocation*. Because there can be many link maps containing many symbols, symbolic relocations are time consuming and expensive. The process of looking up symbol values needs to be done only for symbolic relocations that reference data. Symbolic entries from the `.plt` section are not relocated at startup because they are relocated on demand. However, non-symbolic relocations do not require a lookup and thus are not expensive and do not affect the application startup time. Because relocation processing can be the most expensive operation during application startup, it is desirable to have fewer symbols that can be relocated. See "Appendix" on page 287 for instructions about how to estimate the number of relocations on a library.

This process can be summarized as follows:

Each global symbol has a runtime overhead for binding the symbol. This overhead may occur for all symbols at program startup, or it may occur only for referenced symbols upon first reference. In addition, each use of a symbol will have a runtime overhead for the indirection of the binding tables.

A symbol that needs binding is visible in the library as a relocation. Reducing the number of relocations will reduce both forms of overhead, and yield faster libraries.

Reducing the Number of Relocations

One way of reducing the relocations is to have fewer symbols visible outside the application or library. This can be done by declaring locally used functions and global data private to the application/library. Using the `static` keyword as a function type in C/C++ programs, makes the function local to the module and the symbol will not appear in the dynamic symbol table (`.dynsym`). `elfdump`[6] or `nm`[7] utilities can be used to examine the symbol table of an object file.

Another way is to use the mapfile option to control the scope of functions and symbols. But due to the overhead of maintaining map files with the changes in source code and compiler versions explained earlier, it is not a preferable scheme to be used with C++ applications.

Yet another way is to indicate the appropriate linker scoping within the source program with the help of language extensions in the Sun Studio C/C++ compilers, as described in the following section.

Linker Scoping With Sun Studio Compilers

With the release of Sun Studio 8 compilers, C and C++ are now capable of describing symbol visibility. Although the symbol visibility is specified in the source file, it actually defines how a symbol can be accessed once it has become part of an executable or shared object. The default visibility of symbol is specified by the symbol's binding type.

By using a combination of linker scope specifier directives and command-line options, the programmer can define the runtime interface of a C/C++ object. These definitions are then encoded in the symbol table, and used by the link-editor in a similar manner as reading definitions from a mapfile. With this interface definition technique, the compilation method can greatly reduce the number of symbols that would normally be employed in runtime relocations. In addition, as the compiler knows what implementation symbols must remain global within the final object, these symbols are given the appropriate visibility attribute to ensure their correct usage.

The language/compiler extensions include a new compiler flag and a new C/C++ source language interface.

The new compiler flag is `-xldscope={global| symbolic|hidden}`

`-xldscope` accepts one of the values: `global`, `symbolic`, or `hidden`. This command-line option sets the default linker scope for user-defined external symbols. The compiler issues an error if you specify `-xldscope` without an argument. Multiple instances of this option on the command line override each other until the rightmost instance is reached. Symbols with explicit linker scope qualifiers, declarations of external symbols, static symbols, and local symbols are not affected by the `-xldscope` option.

The new C/C++ source language interface is:

`__global`, `__symbolic`, and `__hidden` declaration specifiers were introduced to specify symbol visibility at declarations and definitions of external symbols and class types. These specifiers are applicable to external functions, variables and classes; and these specifiers takes precedence over the command line (`-xldscope`) option.

With no specifier, the symbol linker scoping remains unchanged from any prior declarations. If the symbol has no prior declaration, the symbol will have the default linker scoping.

Global Scoping

The `__global` specifier can be used to make the symbol definition global in linker scope. This is the default scoping for `extern` symbols. With global scope, all references to the symbol bind to the definition in the first dynamic load module (shared library) that defines the symbol. To make all symbols global in scope, the programmer need not use any special flags, as it is the default. Note that `-xldscope=global` is the default assumed by the compiler; so, specifying `-xldscope=global` explicitly on the command line has no additional effect beyond overriding a previous `-xldscope` on the same command line.

Symbolic Scoping

Symbolic scoping (also known as *protected*) is more restrictive than global linker scoping; all references within a library that match definitions within the library will bind to those definitions. Outside of the library, the symbol appears as though it was global. That is, at first the link-editor tries to find the definition of the symbol within the library. If found, the symbol will be bound to the definition during link time; otherwise, the search continues outside the library as the case with global symbols. For variables, there is an extra complication of copy relocations[8].

Symbolic scoping ensures that the library uses its own versions of specific functions no matter what might appear elsewhere in the program. There are times when symbolic scoping of a set of symbols is exactly what we want. For instance, symbolic scoping fits well in a scenario, where there is an encryption function, with the requirement that it must not be overridden by any other function from any other library irrespective of the link order of those libraries during link time.

On the downside, we lose the flexibility of library interposition, as the resulting symbols are non-interposable. Library interposition is a useful technique for tuning performance, collecting runtime statistics, or debugging applications. For example, if `libc` was built with symbolic scoping, then we cannot take advantage of faster memory allocator libraries like `libmtmalloc.so` for multi-threaded applications by simply preloading `libmtmalloc` and interposing `malloc`. To do so, the symbol `malloc` must be interposable with global binding.

With the `__symbolic` specifier, symbol definitions will have symbolic linker scope. With `-xldscope=symbolic` on the command line and without any linker scoping specifiers in the source code, all the symbols of the library get symbolic scoping. This linker scoping corresponds to the linker option, `-Bsymbolic`.

Be aware that with symbolic scoping, you can wind up with multiple copies of an object or function in a program when only one should be present. For example, suppose a symbol X is defined in library L scoped *symbolic*. If X is also defined in the main program or another library that is linked ahead of L, library L will use its own copy of X, but everything else in the program will use a different copy of X. When using the `-Bsymbolic` linker option or

`-xldscope=symbolic` compiler option, this potential problem extends to every symbol defined in the library, not just the ones you intend to be symbolic.

Which one to choose: `-Bsymbolic` or compiler-supported symbolic mechanism?

Some interfaces created by languages such as C++, provide implementation details of the language itself. These implementation interfaces often must remain global within a group of similar dynamic objects, as one interface must interpose on all the others for the correct execution of the application. As users generally are not aware of what implementation symbols are created, they can blindly demote these symbols to local with options like `-Bsymbolic`. For this reason, `-Bsymbolic` has never been supported with C++, and its use was discouraged with C++.

Using linker scoping specifiers is the preferred way to specify symbolic scoping to a symbol. If the source code changes are not feasible, compile the source with `-xldscope=symbolic`. `-xldscope=symbolic` is considerably safer than `-Bsymbolic` at link time. `-Bsymbolic` is a big hammer that affects every non-local symbol. With the compiler option, certain compiler-generated symbols that need to be global remain global. Also the compiler options do not break exception handling, whereas the linker `-Bsymbolic` option can break exception handling.

A linker mapfile is an alternative solution. Check the introductory paragraphs for the problems associated with linker mapfiles.

Hidden Scoping

Symbols with `__hidden` specifier will have hidden linker scoping. Hidden linker scoping is the most restrictive scope of all. All references within a dynamic load module bind to a definition within that module and the symbol will not be visible outside of the module. That is, the symbol will be local to the library in which it was defined and other libraries may not know the existence of such symbol.

Using `-xldscope=hidden` requires using at least a `__global` or `__symbolic` declaration specifier. Otherwise the instructions result in a library that is completely unusable. The mixed use of `-xldscope=hidden` and `__symbolic` will yield the same effect as `__declspec(dllexport)` in DLLs on Windows (explained in the later part of the article).

Summary of Linker Scoping

The following table provides a summary of linker scoping.

Declaration Specifier	-xldscope Value	Reference Binding	Visibility of Definitions
__global	global	First Module	All Modules
__symbolic	symbolic	Same Module	All Modules
__hidden	hidden	Same Module	Same Module only

The linker will choose the most restrictive scoping specified for all definitions.

Linker scoping specifiers are applicable to `struct`, `class` and `union` declarations and definitions. Consider the following example:

```
__hidden struct __symbolic BinaryTree node;
```

The declaration specifier before the `struct` keyword applies to variable `node`. The class key modifier after the `struct` keyword applies to the type `BinaryTree`.

Rules for using these specifiers are as follows:

- A symbol definition may be redeclared with a more restrictive specifier, but may not be redeclared with a less restrictive specifier. This definition corresponds well with the ELF[9] definition, which says that the symbol scoping chosen is the most restrictive.
- A symbol may not be declared with a different specifier once the symbol has been defined. This is due to the fact that C++ class members cannot be redeclared. In C++, an entity must be defined exactly once; repeated definitions of the same entity in separate translation units result in a error.
- All virtual functions must be visible to all compilation units that include the class definition because the declaration of virtual functions affects the construction and interpretation of virtual tables.

Note these additional guidelines::

- Declaration specifiers apply to all declarations as well as definitions.
- Function and variable declarations are unaffected with `-xldscope` flag, only the definitions are affected.
- With Sun Studio 8 compilers, out-of-line inline functions were static, and thus always hidden. With Sun Studio 9, out-of-line inline functions are global by default, and are affected by linker scoping specifiers and `-xldscope`.
- C does not have (or need) `struct` linker scoping.
- Library functions declared with the `__hidden` or `__symbolic` specifiers can be generated inline when building the library. They are not supposed to be overridden by clients. If you intend to allow a client to override a function in a library, you must ensure that the function is not generated inline in the library.

  The compiler inlines a function if you:

- specify the function name with `-xinline`
- compile at `-xO4` or higher in which case inlining can happen automatically
- use the `inline` specifier
- use the `#pragma inline`

- Library functions declared with the `__global` specifier should not be declared inline, and should be protected from inlining by use of the `-xinline` compiler option.
- `-xldscope` option does not apply to tentative[10] definitions; tentative definitions continue to have global scope.
- If the source file with static symbols is compiled with `-xldscope=symbolic` and if the same object file is used in building more than one library, dynamically loading/unloading, referencing the common symbols from those libraries may lead to a crash during runtime due to the possible symbol conflict. This is due to the globalization of static symbols to support "fix and continue" debugging. These global names must be interposable for "fix and continue" to work.

  If the same object file, say `x.o`, has to be linked in creating more than one library, use object file (`x.o`) with a different timestamp each time you build a new library, that is, compile the original source again just before building a new library. Or, compile the original source to create object files with different names, say, `x_1.o`, `x_2.o`, etc., and use those unique object file names in building new libraries.
- The scoping restraints that we specify for a static archive or an object file will not take effect until the file is linked into a shared library or an executable. This behavior can be seen in the following C program with mixed specifiers:

```
% cat employee.c
__global const float lversion = 1.2;
__symbolic int taxrate;

__hidden struct employee {
 int empid;
 char *name;
 } Employee;

 __global void createemployee(int id, char *name) { }
 __symbolic void deleteemployee(int id) { }
 __hidden void modifyemployee(int id) { }

 % cc -c employee.c
 % elfdump -s employee.o | egrep -i "lver|tax|empl" | grep -v "employee.c"
 [5] 0x00000004 0x00000004 OBJT GLOB P 0 COMMON taxrate
 [6] 0x00000004 0x00000008 OBJT GLOB H 0 COMMON Employee
 [7] 0x00000068 0x00000018 FUNC GLOB H 0 .text modifyemployee
 [8] 0x00000040 0x00000018 FUNC GLOB P 0 .text deleteemployee
 [9] 0x00000010 0x0000001c FUNC GLOB D 0 .text createemployee
 [10] 0x00000000 0x00000004 OBJT GLOB D 0 .rodata lversion
```

In this example, though different visibility was specified for all the symbols, scoping restraints were not in effect in the ELF relocatable object. Due to this, all symbols have global (`GLOB`) binding. However the object file is holding the corresponding ELF symbol visibility attributes for all the symbols according to their binding type.

Variable `lversion` and function `createemployee` have attribute `D`, which stands for `DEFAULT` visibility (that is, `__global`). So those two symbols are *visible* outside of the defining component, the executable file or shared object.

`taxrate` and `deleteemployee` have attribute `P`, which stands for `PROTECTED` visibility (`__symbolic`). A symbol that is defined in the current component is *protected* if the symbol is visible in other components but cannot be pre-empted. Any reference to such a symbol from within the defining component must be resolved to the definition in that component. This resolution must occur, even if a symbol definition exists in another component that would interpose by the default rules.

Function `modifyemployee` and structure `Employee` were `HIDDEN` with attribute `H` (`__hidden`). A symbol that is defined in the current component is *hidden* if its name is not visible to other components. Such a symbol is necessarily protected. This attribute is used to control the external interface of a component. An object named by such a symbol can still be referenced from another component if its address is passed outside.

A hidden symbol contained in a relocatable object is either removed or converted to local (`LOCL`) binding when the object is included in an executable file or shared object. It can be seen in the following example:

```
% cc -G -o libempl.so employee.o
% elfdump -sN.dynsym libempl.so | egrep -i "lver|tax|empl"
  [5] 0x00000298 0x00000018 FUNC GLOB P 0 .text deleteemployee
  [6] 0x00010360 0x00000004 OBJT GLOB P 0 .bss taxrate
  [9] 0x000002f4 0x00000004 OBJT GLOB D 0 .rodata lversion
  [11] 0x00000268 0x0000001c FUNC GLOB D 0 .text createemployee
% elfdump -sN.symtab libempl.so | egrep -i "lver|tax|empl" \
    | grep -v "libempl.so" | grep -v "employee.c"
  [19] 0x000002c0 0x00000018 FUNC LOCL H 0 .text modifyemployee
  [20] 0x00010358 0x00000008 OBJT LOCL H 0 .bss Employee
  [36] 0x00000298 0x00000018 FUNC GLOB P 0 .text deleteemployee
  [37] 0x00010360 0x00000004 OBJT GLOB P 0 .bss taxrate
  [40] 0x000002f4 0x00000004 OBJT GLOB D 0 .rodata lversion
  [42] 0x00000268 0x0000001c FUNC GLOB D 0 .text createemployee
```

Because of the `__hidden` specifier, `Employee` and `modifyemployee` were locally bound (`LOCL`) with hidden (`H`) visibility and didn't show up in dynamic symbol table; hence `Employee` and `modifyemployee` cannot go into the procedure linkage table (PLT), and the run-time linker need only deal with four out of six symbols.

Default Scope

At this point, it is worth looking at the default scope of symbols without the linker scoping mechanism in force, to practically observe the things we learned so far:

```
% cat employee.c
const float lversion = 1.2;
int taxrate;
```

```
struct employee {
 int empid;
 char *name;
 } Employee;

 void createemployee(int id, char *name) { }
 void deleteemployee(int id) { }
 void modifyemployee(int id) { }

 % cc -c employee.c
 % elfdump -s employee.o | egrep -i "lver|tax|empl" | grep -v "employee.c"
 [5] 0x00000004 0x00000004 OBJT GLOB D 0 COMMON taxrate
 [6] 0x00000004 0x00000008 OBJT GLOB D 0 COMMON Employee
 [7] 0x00000068 0x00000018 FUNC GLOB D 0 .text modifyemployee
 [8] 0x00000040 0x00000018 FUNC GLOB D 0 .text deleteemployee
 [9] 0x00000010 0x0000001c FUNC GLOB D 0 .text createemployee
 [10] 0x00000000 0x00000004 OBJT GLOB D 0 .rodata lversion

 % cc -G -o libempl.so employee.o
 % elfdump -sN.dynsym libempl.so | egrep -i "lver|tax|empl"
 [1] 0x00000344 0x00000004 OBJT GLOB D 0 .rodata lversion
 [4] 0x000103a8 0x00000008 OBJT GLOB D 0 .bss Employee
 [6] 0x000002e8 0x00000018 FUNC GLOB D 0 .text deleteemployee
 [9] 0x00000310 0x00000018 FUNC GLOB D 0 .text modifyemployee
 [11] 0x000103b0 0x00000004 OBJT GLOB D 0 .bss taxrate
 [13] 0x000002b8 0x0000001c FUNC GLOB D 0 .text createemployee

 % elfdump -sN.symtab libempl.so | egrep -i "lver|tax|empl" \
   | grep -v "libempl.so" | grep -v "employee.c"

 [30] 0x00000344 0x00000004 OBJT GLOB D 0 .rodata lversion
 [33] 0x000103a8 0x00000008 OBJT GLOB D 0 .bss Employee
 [35] 0x000002e8 0x00000018 FUNC GLOB D 0 .text deleteemployee
 [38] 0x00000310 0x00000018 FUNC GLOB D 0 .text modifyemployee
 [40] 0x000103b0 0x00000004 OBJT GLOB D 0 .bss taxrate
 [42] 0x000002b8 0x0000001c FUNC GLOB D 0 .text createemployee
```

From the above `elfdump` output, all the six symbols were having global binding. So, PLT will be holding at least six symbols.

Establishing an Object Interface

You can use either of these two suggestions on establishing an object interface:

- Define all interface symbols using the `__global` directive, and reduce all other symbols to local using the `-xldscope=hidden` compiler option. This model provides the most flexibility. All global symbols are interposable, and allow for any copy relocations to be processed correctly.
- Define all interface symbols using the `__symbolic` directive, data objects using the `__global` directive, and reduce all other symbols to local using the `-xldscope=hidden` compiler option. Symbolic symbols are globally visible, but have been internally bound to. This means that these symbols do not require symbolic runtime relocation, but cannot be interposed upon, or have copy relocations against them. Note that the problem of copy

relocations only applies to data, but not to functions. This mixed model in which functions are symbolic and data objects are global will yield more optimization opportunities in the compiler.

In short: if we do not want a user to interpose upon our interfaces, and don't export data items, the second model, that is, the mixed model with `__symbolic` and `__global`, is the best. If in doubt, better stick to the more flexible use of `__global` (the first model).

Examples

Exporting of symbols in dynamic libraries can be controlled with the help of `__global`, `__symbolic`, and `__hidden` declaration specifiers. Look at the following header file:

```
% cat tax.h
int taxrate = 33;
float calculatetax(float);
```

If the `taxrate` is not needed by any code outside of the module, we can hide it with the `__hidden` specifier and compile with the `-xldscope=global` option. Or leave `taxrate` to the default scope, make `calculatetax()` visible outside of the module by adding `__global` or `__symbolic` specifiers and compile the code with the `-xldscope=hidden` option. Let's have a look at both approaches.

First approach:

```
% more tax.h
__hidden int taxrate = 33;
float calculatetax(float);

% more tax.c
#include "tax.h"

float calculatetax(float amount) {
 return ((float) ((amount * taxrate)/100));
 }
 % cc -c -KPIC tax.c
 % cc -G -o libtax.so tax.o
 % elfdump -s tax.o | egrep "tax"
 [8] 0x00000010 0x00000068 FUNC GLOB D 0 .text calculatetax
 [9] 0x00000000 0x00000004 OBJT GLOB H 0 .data taxrate

 % elfdump -s libtax.so | egrep "tax"
 [6] 0x00000240 0x00000068 FUNC GLOB D 0 .text calculatetax
 [23] 0x00010350 0x00000004 OBJT LOCL H 0 .data taxrate
 [42] 0x00000240 0x00000068 FUNC GLOB D 0 .text calculatetax
```

Second approach:

```
% more tax.h
int taxrate = 33;
__global float calculatetax(float);
```

```
% more tax.c
#include "tax.h"

float calculatetax(float amount) {
 return ((float) ((amount * taxrate)/100));
 }

 % cc -c -xldscope=hidden -Kpic tax.c
 % cc -G -o libtax.so tax.o

 % elfdump -s tax.o | egrep "tax"
 [8] 0x00000010 0x00000068 FUNC GLOB D 0 .text calculatetax
 [9] 0x00000000 0x00000004 OBJT GLOB H 0 .data taxrate

 % elfdump -s libtax.so | egrep "tax"
 [6] 0x00000240 0x00000068 FUNC GLOB D 0 .text calculatetax
 [23] 0x00010350 0x00000004 OBJT LOCL H 0 .data taxrate
 [42] 0x00000240 0x00000068 FUNC GLOB D 0 .text calculatetax
```

Now it is clear that the same effect of symbol visibility can be achieved by changing either the specifier and/or the command-line interface through `-xldscope` flag. (The first entry in "Appendix" on page 287 shows the binding types with all possible source interfaces (specifiers) and command line option, `-xldscope`)

Let's try to build a driver that invokes `calculatetax()` function. But at first, let's modify `tax.h` slightly to make `calculatetax()` non-interposable (as described in the second suggestion in "Establishing an Object Interface" on page 277) and build `libtax.so`.

```
% cat tax.h
int taxrate = 33;
__symbolic float calculatetax(float);
% cc -c -xldscope=hidden -Kpic tax.c
% cc -G -o libtax.so tax.o

% cat driver.c
#include <stdio.h>
#include "tax.h"

int main() {
 printf("** Tax on $2525 = %0.2f **\n", calculatetax(2525));
 return (0);
 }

 % cc -R. -L. -o driver driver.c -ltax
 Undefined first referenced
 symbol in file
 calculatetax driver.o (symbol scope specifies local binding)
 ld: fatal: Symbol referencing errors. No output written to driver
 % elfdump -s driver.o | egrep "calc"
 [7] 0x00000000 0x00000000 FUNC GLOB P 0 UNDEF calculatetax
```

Building the driver program failed, because even the client program (`driver.c`) is trying to export (`__symbolic`) the definition of `calculatetax()`, instead of importing (`__global`) it. The declarations within header files shared between the library and the clients must ensure that clients and implementation have different values for the linker scoping of public symbols. So,

the simple fix is to export the symbol while the library being built and import it when the client program needs it. This can be done by either copying the `tax.h` to another file and changing the specifier to `__global`, or by using preprocessor conditionals (with `-D` option) to alter the declaration depending on whether the header file is used in building the library or by a client.

Using separate header files for clients and the library leads to code maintenance problems. Even though using a compiler directive eases the pain of writing and maintaining two header files, unfortunately it places lots of implementation details in the public header file. The following example illustrates this by introducing the compiler directive `BUILDLIB` for building the library.

```
% more tax.h
int taxrate = 33;

#ifdef BUILDLIB
 __symbolic float calculatetax(float);
#else
 __global float calculatetax(float);
#endif
```

When the library was built, the private compiler directive defines `BUILDLIB` to be non-zero, so the symbol `calculatetax` will be exported. While building a client program with the same header file, the `BUILDLIB` variable is set to zero, and `calculatetax` will be made available to the client, that is, the symbol will be imported.

You may want to emulate this system by defining macros for your own libraries. This implies that you have to define a compiler switch (analogous to `BUILDLIB`) yourself. This can be done with the `-D` flag of Sun Studio C/C++ compilers. Using `-D` option at the command line is equivalent to including a `#define` directive at the beginning of the source. Set the switch to non-zero when you're building your library, and then set it to zero when you publish your headers for use by library clients.

Let's continue with the example by adding the directive `BUILDLIB` to the compile line that builds `libtax.so`.

```
% make
Compiling tax.c ..
cc -c -xldscope=hidden -Kpic tax.c

Building libtax library ..
cc -G -DBUILDLIB -o libtax.so tax.o

Building driver program ..
cc -ltax -o driver driver.c

Executing driver ..
./driver
** Tax on $2525 = 833.25 **
```

The following is an alternative implementation for the above example, with a simple interface in the public header file. The idea behind this approach is to use a second header file that redeclares symbols with a more restrictive linker scope for use within the library.

```
% cat tax_public.h
float calculatetax(float);

% cat tax_private.h
#include "tax_public.h"
int taxrate = 33;

% cat tax_private.c
#include "tax_private.h"

__symbolic float calculatetax(float amount) {
 return ((float) ((amount * taxrate)/100));
 }
```

This code makes the symbol `calculatetax` symbolic when compiling the `tax_private.c` file; and the compiler makes optimizations knowing that `calculatetax` is symbolic and is only available within the one object file. To make these optimizations known to the entire library, the function would be redeclared in the private header.

```
% cat tax_private.h
#include "tax_public.h"
int taxrate = 33;
__symbolic float calculatetax(float);
```

To export the symbol `calculatetax`, a private header should be used while building the library.

```
% cat tax_private.c
#include "tax_private.h"

float calculatetax(float amount) {
 return ((float) ((amount * taxrate)/100));
 }

% cc -c -xldscope=hidden -KPIC tax_private.c
% cc -G -o libtax_private.so tax_private.o
```

A public header should be used while building the client program so the client can access `calculatetax`, since it will have global visibility.

```
% cat driver.c
#include <stdio.h>
#include "tax_public.h"

int main() {
 printf("** Tax on $2525 = %0.2f **\n", calculatetax(2525));
 return (0);
 }

 % cc -ltax_private -o driver driver.c
 % ./driver
 ** Tax on $2525 = 833.25 **
```

The trade-off with this alternate approach is that we need two sets of header files, one for exporting the symbols and the other for importing them.

Windows Compatibility With `__declspec`

Sun Studio 9 compilers introduced a new keyword called `__declspec` and supports `dllexport` and `dllimport` storage-class attributes (or *specifiers*) to facilitate the porting of applications developed using Microsoft Windows compilers to the Solaris OS.

Syntax:

```
storage... __declspec( dllimport ) type declarator...
storage... __declspec( dllexport ) type declarator...
```

On Windows, these attributes define the symbols exported (the library as a *provider*) and imported (the library as a *client*).

On the Solaris OS, `__declspec(dllimport)` maps to `__global` and `__declspec(dllexport)` maps to the `__symbolic` specifier. Note that the semantics of these keywords are somewhat different on Microsoft and Solaris platforms. So, the applications being developed natively on the Solaris platform are strongly encouraged to stick to Solaris syntax, instead of using Microsoft specific extensions to C/C++.

Just to present the syntax, the following sections, especially `__declspec` (`dllexport`) and `__declspec(dllimport)`, assume that we are going to use the `__declspec` keyword in place of linker scoping specifiers.

`__declspec(dllexport)`

While building a shared library, all the global symbols of the library should be explicitly exported using the `__declspec` keyword. To export a symbol, the declaration will be like:

```
__declspec(dllexport) type name
```

where "`_declspec(dllexport)`" is literal, and *type* and *name* declare the symbol.

For example,

```
__declspec(dllexport) char *printstring();
class __declspec(dllexport) MyClass {...}
```

Data, functions, classes, or class member functions from a shared library can be exported using the `__declspec(dllexport)` keyword.

When building a library, we typically create a header file that contains the function prototypes and/or classes we are exporting, and add `__declspec(dllexport)` to the declarations in the header file.

Sun Studio compilers map `__declspec(dllexport)` to `__symbolic`; hence the following two declarations are equivalent:

```
__symbolic void printstring();
__declspec(dllexport) void printstring();
```

——declspec(dllimport)

To import the symbols that were exported with `__declspec(dllexport)`, a client, that wants to use the library must reverse the declaration by replacing `dllexport` with `dllimport`.

For example,

```
__declspec(dllimport) char *printstring();
class __declspec(dllimport) MyClass{...}
```

A program that uses public symbols defined by a shared library is said to be importing them. While creating header files for applications that use the libraries to build with, `__declspec(dllimport)` should be used on the declarations of the public symbols.

Sun Studio compilers map `__declspec(dllimport)` to `__global`; hence the following two declarations are equivalent:

```
__global void printstring();
__declspec(dllimport) void printstring();
```

Automatic Data Imports

Windows C/C++ compilers may accept code that is declared with `__declspec(dllexport)` but actually imported. Such code will not compile with Sun Studio compilers. The `dllexport/dllimport` attributes must be correct. This constraint increases the effort necessary to port existing Windows code to the Solaris OS, especially for large C++ libraries and applications.

For example, the following code may compile and run on Windows, but doesn't compile on the Solaris platform.

```
% cat util.h
__declspec(dllexport) long multiply (int, int);

% cat util.cpp
#include "util.h"

long multiply (int x, int y) {
 return (x * y);
 }

 % cat test.cpp
 #include <stdio.h>
 #include "util.h"

 int main() {
 printf(" 25 * 25 = %ld", multiply(25, 25));
 return (0);
 }
```

```
% CC -G -o libutil.so util.cpp

% CC -o test test.cpp -L. -R. -lutil
Undefined first referenced
symbol in file
multiply test.o (symbol scope specifies local binding)
ld: fatal: Symbol referencing errors. No output written to test

% elfdump -CsN.symtab test.o | grep multiply
[3] 0x00000000 0x00000000 FUNC GLOB P 0 UNDEF long multiply(int,int)
```

The normal mapping for `__declspec(dllexport)` is `__symbolic`, which requires that the symbol be inside the library, which is not true for imported symbols. The correct solution is for customers to change their Microsoft code to use `__declspec(dllimport)` to declare imported symbols. (The solution with an example, was already discussed in "Windows Compatibility With `__declspec`" on page 282.)

To port such code, Sun provided a solution with an undocumented compiler option that does not involve changing the source base. `-xldscoperef` is the new compiler option, and it accepts two values, `global` or `keyword`. This option is available with Sun Studio 9 and later versions by default; and to Sun Studio 8, as a patch 112760-01 (or later) and 112761-01 (or later) for SPARC and x86 platforms respectively.

The value `keyword` indicates that a symbol will have the linker scoping specified by any keywords given with the symbol's declarations. This value is the current behavior and the default. The value `global` indicates that all undefined symbols should have global linkage[11], even if a linker scoping keyword says otherwise. Note that `__declspec(dllimport)` and `__declspec(dllexport)` still map to `__global` and `__symbolic` respectively; and only undefined symbols are affected with the `-xldscoperef=global` option.

So, compiling the code from the above example with `-xldscoperef=global` would succeed and produce the desired result. Since `-xldscoperef` is not a first class option, it has to be passed to the front end with the help of the `-Qoption` option of the C++ compiler and with the `-W` option of the C compiler. (`-Qoption ccfe -xldscoperef=global` for C++ and `-W0,-xldscoperef=global` for C).

```
% CC -Qoption ccfe -xldscoperef=global -lutil -o test test.cpp

% ./test
 25 * 25 = 625

% elfdump -CsN.symtab test.o | grep multiply
 [3] 0x00000000 0x00000000 FUNC GLOB D 0 UNDEF long multiply(int,int)
```

Let's conclude by stating some of the benefits of reduced linker scoping.

Benefits of Linker Scoping

The following paragraphs explain some of the benefits of the linker scoping feature. We can take advantage of most of the benefits listed just by reducing the scope of all or most of the symbols in our application from global to local.

- *Less chance for name collisions with other libraries.*

  With C++, namespaces are the preferred method for avoiding name collisions. But applications that rely heavily on C style programming and don't use the namespace mechanism are vulnerable to name collisions.

  Name collisions are hard to detect and debug. Third-party libraries can create havoc when some of their symbol names coincide with those in the application. For example, if a third-party shared library uses a global symbol with the same name as a global symbol in one of the application's shared libraries, the symbol from the third-party library may interpose on ours and unintentionally change the functionality of the application without any warning. With symbolic scoping, we can make it hard to interpose symbols and ensure the correct symbol being used during runtime.

- *Improved performance.*

  Reducing the exported interfaces of shared objects greatly reduces the runtime overhead of processing these objects and improves the application startup time and the runtime performance. Due to the reduced symbol visibility, the symbol count is reduced, hence there is less overhead in runtime symbol lookup, and the relocation count is reduced, hence there is less overhead in fixing up the objects prior to their use.

- *Thread-local storage* (TLS).

  Access to thread-local storage can be significantly faster as the compiler knows the inter-object, intra-linker-module relationship between a reference to a symbol and the definition of that symbol. If the backend knows that a symbol will not be exported from a dynamic library or executable, it can perform optimizations which it couldn't perform before when it only knew the scope relative to the relocatable object being built.

- *Position-independent code with* `-Kpic`.

  With most symbols hidden, there are fewer symbols in the library, and the library may be able to use the more efficient `-Kpic` rather than the less efficient `-KPIC`.

  The PIC-compiled code allows the linker to keep a read-only version of the text (code) segment for a given shared library. The dynamic linker can share this text segment among all running processes, referencing it at a given time. PIC helps reducing the number of relocations.

- *Improved security.*

  The `strip`(1) utility is not enough to hide the names of the application's routines and data items. Stripping eliminates the local symbols but not the global symbols.

Dynamically linked binaries (both executables and shared libraries) use two symbol tables: the static symbol table and the dynamic symbol table. The dynamic symbol table is used by the runtime linker. It has to be there even in stripped executables or else the dynamic linker cannot find the symbols it needs. The `strip` utility can only remove the static symbol table.

By making most of the symbols of the application local in scope, the symbol information for such local symbols in a stripped binary is really gone and are not available at runtime, so no one can extract it.

Note that even though linker scoping is an easier mechanism to use, it is not the only one and the same could be done with mapfiles too.

- *Better alignment with the supported interface of the library.*

  By reducing the scope of symbols, the linker symbols that the client can link to are aligned with the supported interface of the library, and the client cannot link to functions that are not supported and may do damage to the operation of the library.

- *Reduced application binary sizes.*

  ELF's exported symbol table format is quite a space hog. Due to the reduced linker scope, there will be a noticeable drop in the sizes of the binaries being built

- *Overcoming 64-bit PLT limit of 32768* (*Solaris 8 or previous versions only*).

  In the 64-bit mode, the linker on Solaris 8 or previous versions currently has a limitation: It can only handle up to 32768 PLT entries. This means that we can't link very large shared libraries in the 64-bit mode. Linker throws the following error message if the limit is exceeded:

  ```
  Assertion failed: pltndx < 0x8000
  ```

  The linker needs PLT entries only for the global symbols. If we use linker scoping to reduce the scope of most of the symbols to local, this limitation is likely to become irrelevant.

- *Substitution to the difficult to use/manage mapfiles mechanism.*

  The use of linker mapfiles for linker scoping is difficult with C++ because the mapfiles require linker names, which are not the same names used in the program source (explained in the introductory paragraphs). Linker scoping is a viable alternative to mapfiles for reducing the scope of symbols. With linker scoping, the header files of the library need not change. The source files may be compiled with the `-xldscope` flag to indicate the default linker scoping, and individual symbols that wish another linker scoping are specified in the source.

  Note that linker mapfiles provide many features beyond linker scoping, including assigning addresses to symbols and internal library versioning.

Appendix

- Linker scope specifiers (`__global`, `__symbolic`, and `__hidden`) will have priority over the command-line `-xldscope` option. The following table shows the resulting binding and visibility when the code was compiled with the combination of specifiers and command-line option.

Declaration Specifier	`-xldscope=global`	`-xldscope=symbolic`	`-xldscope=hidden`	`no -xldscope`
`__global`	GLOB DEFAULT	GLOB DEFAULT	GLOB DEFAULT	GLOB DEFAULT
`__symbolic`	GLOB PROTECTED	GLOB PROTECTED	GLOB PROTECTED	GLOB PROTECTED
`__hidden`	LOCL HIDDEN	LOCL HIDDEN	LOCL HIDDEN	LOCL HIDDEN
`no specifier`	GLOB DEFAULT	GLOB PROTECTED	LOCL HIDDEN	GLOB DEFAULT

- Consider a library with a narrow external interface, but with a wide internal implementation. It would typically be compiled with `-xldscope=hidden` and its interface functions defined with `__global` or `__symbolic`.

```
% cat external.h
 extern void non_library_function();
 inline void non_library_inline() {
 non_library_function();
 }

% cat public.h
 extern void interposable();
 extern void non_interposable();
 struct container {
 virtual void method();
 void non_virtual();
 };

 % cat private.h
 extern void inaccessible();

 % cat library.c
 #include "external.h"
 #include "public.h"
 #include "private.h"
 __global void interposable() { }
 __symbolic void non_interposable() { }
 __symbolic void container::method() { }
 __hidden void container::non_virtual() { }
 void inaccessible() {
 non_library_inline();
 }
```

Compiling `library.c` results in the following linker scopings in `library.o`.

Function	Linker Scoping
non_library_function	undefined
non_library_inline	hidden
interposable	global
non_interposable	symbolic
container::method	symbolic
container::non_virtual	hidden
inaccessible	hidden

- The following example interface shows the usage of symbol visibility specifiers with a class template. With the `__symbolic` specifier in the template class definition, all members of all instances of class `Stack` will have the `symbolic` scope, unless overridden.

```
% cat stack.cpp
template <class Type>
class __symbolic Stack
{
private:
 Type items[25];
 int top;
 public:
 Stack();
 Bool isempty();
 Bool isfull();
 Bool push(const Type & item);
 Bool pop(Type & item);
 };
```

  In order to specify linker scoping to the template class definition, the Sun Studio 8 compilers on SPARC and x86 platforms must be patched with the latest patches of 113817-01 (or later) and 113819-01 (or later) respectively. This facility is available with Sun Studio 9 or later versions, by default.

- A trivial C++ example showing accidental symbol collision with a third-party symbol

```
% cat mylib_public.h
float getlibversion();
int checklibversion();

% cat mylib_private.h
#include "mylib_public.h"
const float libversion = 2.2;

% cat mylib.cpp
#include "mylib_private.h"

float getlibversion() {
 return (libversion);
 }
```

```
int checklibversion() {
return ((getlibversion() < 2.0) ? 1 : 0);
}

% CC -G -o libmylib.so mylib.cpp
% cat thirdpartylib.h
const float libversion = 1.5;

float getlibversion();

% cat thirdpartylib.cpp
#include "thirdpartylib.h"

float getlibversion() {
return (libversion);
}

% CC -G -o libthirdparty.so thirdpartylib.cpp

% cat versioncheck.cpp
#include <stdio.h>
#include "mylib_public.h"
int main() {
if (checklibversion()) {
printf("\n** Obsolete version being used .. Can\'t proceed further! **\n");
} else {
printf("\n** Met the library version requirement .. Good to Go! ** \n");
}
return (0);
}

% CC -o vercheck -lthirdparty -lmylib versioncheck.cpp

%./vercheck

** Obsolete version being used .. Can’t proceed further! **
```

Since `checklibversion()` and `getlibversion()` are within the same load module, `checklibversion()` of `mylib` library is expecting the `getlibversion()` to be called from `mylib` library. However the linker picked up the `getlibversion()` from the `thirdparty` library since it was linked before `mylib` when the executable was built.

To avoid failures like this, it is suggested to bind the symbols to their definition within the module itself with symbolic scoping. Compiling the `mylib` library's source with `-xldscope=symbolic` makes all the symbols of the module to be symbolic in nature. It produces the desired behavior and makes it hard for symbol collisions, by ensuring that the library will use the local definition of the routine rather than a definition that occurs earlier in the link order.

```
% CC -G -o libmylib.so -xldscope=symbolic mylib.cpp
% CC -o vercheck -lthirdparty -lmylib versioncheck.cpp
% ./vercheck

** Met the library version requirement .. Good to Go! **
```

- Estimating the number of relocations

To get the number of relocations that the linker may perform, run the following commands:

For the total number of relocations:

```
% elfdump -r <DynamicObject> | grep -v NONE | grep -c R_
```

For the number of non-symbolic relocations:

```
% elfdump -r <DynamicObject> | grep -c RELATIVE
```

For example

```
% elfdump -r /usr/lib/libc.so | grep -v NONE | grep -c R_
 2562
% elfdump -r /usr/lib/libc.so | grep -c RELATIVE
 1868
```

The number of symbolic relocations is calculated by subtracting the number of non-symbolic relocations from the total number of relocations. This number also includes the relocations in the procedure linkage table.

Glossary

1. An *interface* (API) is a specification of functions and use of a software module. In short, it's a set of instructions for other programmers on what all classes, methods, etc., can be used from the module, which provides the interface.
2. A source file contains one or more variables, function declarations, function definitions or similar items logically grouped together. From source file, compiler generates the *object module*, which is the machine code of the target system. Object modules will be linked with other modules to create the *load module*, a program in machine language form, ready to run on the system.
3. *static linker*, `ld(1)`, also called link-editor, creates load modules from object modules.
4. *dynamic linker*, `ld.so.1(1)` performs the runtime linking of dynamic executables and shared libraries. It brings shared libraries into an executing application and handles the symbols in those libraries as well as in the dynamic executable images, that is, the dynamic linker creates a process image from load modules.
5. A *symbolic relocation* is a relocation that requires a lookup in the symbol table. The runtime linker optimizes symbol lookup by caching successive duplicate symbols. These cached relocations are called "cached symbolic" relocations, and are faster than plain symbolic relocations. A *non-symbolic relocation* is a simple relative relocation that requires the base address at which the object is mapped to perform the relocation. Non-symbolic relocations do not require a lookup in the symbol table.
6. The `elfdump`(1) utility can be used to dump selected parts of an object file, like symbol table, elf header, global offset table.
7. `nm`(1) utility displays the symbol table of an ELF object file.

8. *Copy relocations* are a technique employed to allow references from non-pic code to external data items, while maintaining the read-only permission of a typical text segment. This relocation use, and overhead, can be avoided by designing shared objects that do not export data interfaces.
9. *Executable and Linkable Format* (ELF) is a portable object file format supported by most UNIX vendors. ELF helps developers by providing a set of binary interface definitions that are cross-platform, and by making it easier for tool vendors to port to multiple platforms. Having a standard object file format also makes porting of object-manipulating programs easier. Compilers, debuggers, and linkers are some examples of tools that use the ELF format.
10. *Tentative* symbols are those symbols that have been created within a file but have not yet been sized, or allocated in storage. These symbols appear as uninitialized C symbols.
11. The ability of a name in one translation unit to be used as the definition of the same name in another translation unit is called *linkage*. Linkage can be internal or external. Internal linkage means a definition can only be used in the translation unit in which it is found. External linkage means the definition can be used in other translation units as well; in other words, it can be linked into outside translation units.

Resources

- *Sun Studio C++ User's Guide*, "Language Extensions" (`http://docs.sun.com/source/819-0496/Language_Extensions.html`)
- *Solaris Linker and Libraries Guide* (`http://docs.sun.com/app/docs/doc/817-1984`)
- Programming Languages - C++, ISO/IEC 14882 International Standard
- Enhancing Applications by Directing Linker Symbol Processing (`http://developers.sun.com/solaris/articles/linker_mapfiles.html`) by Greg Nakhimovsky
- Interface Creation Using Compilers (`http://blogs.sun.com/rie/entry/interface_creation_using_the_compilers`) by Rod Evans
- Reducing Application Startup Time (`http://developers.sun.com/solaris/articles/reducing_app.html`) by Neelakanth Nadgir

`http://developers.sun.com/solaris/articles/symbol_scope.html`

My Relocations Don't Fit – Position Independence

Rod Evans, April 26, 2005

A couple of folks have come across the following relocation error when running their applications on AMD64:

```
$ prog
  ld.so.1: prog: fatal: relocation error: R_AMD64_32: file \
      libfoo.so.1: symbol (unknown): value 0xfffffd7fff0cd457 does not fit
```

The culprit, `libfoo.so.1` has been built using position-dependent code (often referred to as *non*-pic).

Shared objects are typically built using position-independent code, using compiler options such as `-Kpic`. This position independence allows the code to execute efficiently at a different address in each process that uses the code.

If a shared object is built from position-dependent code, the text segment can require modification at runtime. This modification allows relocatable references to be assigned to the location that the object has been loaded. The relocation of the text segment requires the segment to be remapped as writable. This modification requires a swap space reservation, and results in a private copy of the text segment for the process. The text segment is no longer sharable between multiple processes. Position-dependent code typically requires more runtime relocations than the corresponding position-independent code. Overall, the overhead of processing text relocations can cause serious performance degradation.

When a shared object is built from position-independent code, relocatable references are generated as indirections through data in the shared object's data segment. The code within the text segment requires no modification. All relocation updates are applied to corresponding entries within the data segment.

The runtime linker attempts to handle text relocations should these relocations exist. However, some relocations cannot be satisfied at runtime.

The AMD64 position-dependent code sequence typically generates code which can only be loaded into the lower 32 bits of memory. The upper 32 bits of any address must all be zeros. Since shared objects are typically loaded at the top of memory, the upper 32 bits of an address are required. Position-dependent code within an AMD64 shared object is therefore insufficient to cope with relocation requirements. Use of such code within a shared object can result in the runtime relocation errors cited above.

This situation differs from the default ABS64 mode that is used for 64-bit SPARCV9 code. This position-dependent code is typically compatible with the full 64-bit address range. Thus, position-dependent code sequences can exist within SPARCV9 shared objects. Use of either the ABS32 mode, or ABS44 mode for 64-bit SPARCV9 code, can still result in relocations that cannot be resolved at runtime. However, each of these modes require the runtime linker to relocate the text segment.

Build all your shared objects using position-independent code.

An Update - Wednesday March 21, 2007

If you believe you have compiled all the components of your shared object using -Kpic and still see an error of this sort, look a little more closely. First, determine if the link-editor thinks the shared object contains text relocations.

```
$ elfdump -d library | fgrep TEXTREL
```

If this flag is found, then the link-editor thinks this file contains non-pic code. One explanation might be that you have included an assembler file in the shared library. Any assembler must be written using position-independent instructions. Another explanation is that you might have included objects from an archive library as part of the link-edit. Typically, archives are built with non-pic objects.

You can track down the culprit from the link-editor's debugging capabilities. Build your shared object.

```
$ LD_OPTIONS=-Dreloc,detail cc -o library .... 2> dbg
```

The diagnostic output in dbg can be inspected to locate the non-pic relocation, and from this you can trace back to the input file that supplied the relocation as part of building your library.

http://blogs.sun.com/rie/entry/my_relocations_don_t_fit

Building Shared Libraries for sparcv9

Darryl Gove, October 16, 2007

By default, the SPARC compiler assumes that SPARCV9 objects are built with -xcode=abs44, which means that 44 bits are used to hold the absolute address of any object. Shared libraries should be built using position independent code, either -xcode=pic13 or -xcode=pic32 (http://docs.sun.com/source/819-3688/cc_ops.app.html#pgfId-1029729) (replacing the deprecated -Kpic (http://docs.sun.com/source/819-3688/cc_ops.app.html#pgfId-1026741) and -KPIC (http://docs.sun.com/source/819-3688/cc_ops.app.html#pgfId-999870) options.

If one of the object files in a library is built with abs44, then the linker will report the following error:

```
ld: fatal: relocation error: R_SPARC_H44: file file.o
: symbol <unknown>:
relocations based on the ABS44 coding model can not be used in building
a shared object
```

Further details on this can be found in the compiler documentation (http://docs.sun.com/source/819-3688/cc_ops.app.html#pgfId-1012989).

http://blogs.sun.com/d/entry/building_shared_libraries_for_sparcv9

Interposing on malloc

Darryl Gove, February 8, 2008

Ended up wanting to look at malloc calls, how much was requested, where the memory was located, and where in the program the request was made. This was on S9, so no DTrace, so the obvious thing to do was to write an interpose library and use that. The code is pretty simple:

```
#include <stdlib.h>
#include <stdio.h>
#include <dlfcn.h>
#include <ucontext.h>

void * malloc(size_t size)
{
  static void* (*func)(size_t)=0;
  void* ret;
  if (!func) {func=(void*(*)(size_t))dlsym(RTLD_NEXT,"malloc");}
  ret=func(size);
  printf("size = %i address=%x\n",size,ret);
  printstack(0);
  return ret;
}
```

The code uses a call to printstack(3C) to print out the stack at the point of the call.

The code is compiled and run with:

```
$ cc -O -G -Kpic -o libmallinter.so mallinter.c
$ LD_PRELOAD=./libmallinter.so ls
size = 17 address=25118
/home/libmallinter.so:malloc+0x5c
/usr/lib/libc.so.1:_strdup+0xc
/usr/lib/libc.so.1:0x73b54
/usr/lib/libc.so.1:0x72d44
/usr/lib/libc.so.1:0x720e4
/usr/lib/libc.so.1:setlocale+0x3c
/usr/bin/ls:main+0x14
/usr/bin/ls:_start+0x108
size = 17 address=25138
/home/libmallinter.so:malloc+0x5c
/usr/lib/libc.so.1:_strdup+0xc
/usr/lib/libc.so.1:0x73b54
/usr/lib/libc.so.1:0x72d44
/usr/lib/libc.so.1:0x720e4
/usr/lib/libc.so.1:setlocale+0x3c
/usr/bin/ls:main+0x14
/usr/bin/ls:_start+0x108
```

http://blogs.sun.com/d/entry/interposing_on_malloc

◆ ◆ ◆ CHAPTER 7

Floating Point

This chapter contains some posts about assessing floating-point instruction, discussions of subnormal numbers, and a discussion of the floating-point multiple accumulate instruction.

Measuring Floating-Point Use in Applications

Darryl Gove, January 19, 2006

MrBenchmark has written a blog entry (`http://blogs.sun.com/mrbenchmark/entry/is_my_workload_recommended_for`) on assessing floating-point performance in codes using the hardware performance counters on the UltraSPARC-III. It's a nice detailed read, so I'm not going to repeat any of what's written there.

However, there's a bit more to be said on the topic - in particular the technique used may overestimate floating-point content in some codes.

The best place to start is a test program

```
double a[1024*1024],b[1024*1024];

void main()
{
  int i,j;
  for (i=0; i<1024*1024;i++) {a[i]=b[i]=0.0;}
  for (j=0;j<100;j++) for (i=0; i<1024*1024;i++) {a[i]+=b[i];}
  for (j=0; j<100; j++) for (i=0; i<1024*1024;i++) {a[i]*=b[i];}
}
```

The program does 100*1024*1024 (approx 100 million) each of floating-point adds and floating-point multiplies. The next step is to compile and run the program.

```
$ cc fp
$ cputrack -c pic0=FA_pipe_completion,pic1=Instr_cnt \
  -c pic0=Instr_cnt,pic1=FM_pipe_completion \
```

```
  a.out | grep exit
 32.687   1       exit  55477739 1104761007 # pic0=FA_pipe_completion,pic1=Instr_cnt
 32.071   1       exit 1001979477  50793545 # pic0=Instr_cnt,pic1=FM_pipe_completion
```

The program has been run under `cputrack`(1), which follows the process and counts the number of the specified events that occur. `cputrack` is set to count:

- `FA_pipe_completion` events - instructions that use the floating-point add pipe.
- `FM_pipe_completion` events - instructions that use the floating-point multiply pipe.
- `Instr_cnt` - the total instruction count

[If you want to find out more about the available performance counters on the UltraSPARC hardware, see the processor manuals (`http://www.sun.com/processors/documentation.html`).]

Since `cputrack` was asked to count two different sets of events, it alternates between them, which means that the number of each type of event that it counts is undercounted by a factor of two (assuming that the distribution of missed events was even). Given this information it is possible to estimate the total number of instructions that used the floating-point add pipe as being 111 million (55,477,739*2), and the total number that used the floating-point multiply pipeline as 101 million (50,793,545*2). The total number of instructions was 2.1 billion (1,104,761,007 + 1,001,979,477). So the floating-point content is about 10% ((101+111)/2100).

The floating-point percentage is artificially low because the code was not compiled with any optimization, so there were plenty of instructions which could have been optimised out.

Now I've carefully written floating-point add *pipeline*, and there is a good reason for doing that. It is not only floating-point add or multiply events that go down the floating-point pipes, it is also VIS (`http://www.sun.com/processors/vis/`) instructions. You may have thought that your code does not use any VIS instructions, but unfortunately most code does use them at some point. Here's a simple example.

```
#include <strings.h>

char a[10*1024*1024],b[10*1024*1024];
void main()
{
   int i;
   for (i=0;i<100; i++){bcopy(a,b,10*1024*1024);}
}
```

And here is it running under `cputrack`:

```
$ cc mov.c
$ cputrack -c pic0=FA_pipe_completion,pic1=Instr_cnt \
  -c pic0=Instr_cnt,pic1=FM_pipe_completion a.out | grep exit
  1.098   1       exit  73756539 212490199 # pic0=FA_pipe_completion,pic1=Instr_cnt
  1.791   1       exit 164927147         0 # pic0=Instr_cnt,pic1=FM_pipe_completion
```

So this little code generated 146 million (73,756,539*2) instructions that used the floating-point add pipe, out of a total of 376 million instructions - pretty much half of the total number of instructions! Now this degree of disturbance is unusual (or contrived), but there is a lot of code out there that does spend significant time in `memcpy` type functions, so care has to be taken when using this approach.

The next obvious question is whether it is possible to check for this. Of course, the thing to do is to fire up the Performance Analyzer from Sun Studio (`http://developers.sun.com/prodtech/cc/products/index.html`), and use that to gather a profile.

```
$ collect a.out
Creating experiment database test.1.er ...
$ er_print -limit 3 -func test.1.er
Functions sorted by metric: Exclusive User CPU Time

Excl.     Incl.      Name
User CPU  User CPU
 sec.      sec.
1.321     1.321      _memcpy
1.321     1.321      <total>
0.        1.321      main
```

From the profile it is possible to be pretty certain that most of the events come in from `_memcpy`. To be sure take a look at the hot disassembly. To do this it is necessary to run `er_print` interactively to pick the correct version of `_memcpy` from `libc_psr`.

```
$ er_print test.1.er
(er_print) dis _memcpy
Available name list:
       0) Cancel
       1) (unknown)(/usr/lib/libc.so.1)‘_memcpy
       2) (unknown)(/usr/platform/sun4u-us3/lib/libc_psr.so.1)‘_memcpy
       3) (unknown)(/usr/lib/ld.so.1)‘_memcpy
Enter selection: 2
Source file: (unknown)
Object file: /usr/platform/sun4u-us3/lib/libc_psr.so.1
Load Object: /usr/platform/sun4u-us3/lib/libc_psr.so.1
...
   0.        0.       [?]   720:  ldd         [%o1], %f2
   0.        0.       [?]   724:  faligndata  %f12, %f14, %f44
   0.        0.       [?]   728:  ldd         [%o1 + 8], %f4
   0.        0.       [?]   72c:  faligndata  %f14, %f0, %f46
   0.090     0.090    [?]   730:  stda        %f32, [%o0] 0xf0
   0.        0.       [?]   734:  ldd         [%o1 + 16], %f6
   0.        0.       [?]   738:  faligndata  %f0, %f2, %f32
   0.100     0.100    [?]   73c:  inc         64, %o0
   0.        0.       [?]   740:  ldd         [%o1 + 24], %f8
   0.        0.       [?]   744:  faligndata  %f2, %f4, %f34
   0.        0.       [?]   748:  ldd         [%o1 + 32], %f10
   0.        0.       [?]   74c:  faligndata  %f4, %f6, %f36
   0.        0.       [?]   750:  ldd         [%o1 + 40], %f12
   0.        0.       [?]   754:  dec         64, %o2
   0.        0.       [?]   758:  faligndata  %f6, %f8, %f38
   0.100     0.100    [?]   75c:  ldd         [%o1 + 48], %f14
```

```
   0.        0.       [?]   760:  faligndata  %f8, %f10, %f40
   0.030     0.030    [?]   764:  ldd         [%o1 + 56], %f0
## 1.001     1.001    [?]   768:  prefetch    [%o1 + 384], #one_read
   0.        0.       [?]   76c:  faligndata  %f10, %f12, %f42
   0.        0.       [?]   770:  cmp         %o2, 72
   0.        0.       [?]   774:  bgu,pt      %icc,0x720
   0.        0.       [?]   778:  inc         64, %o1
...
```

So the reason for all the floating-point add pipeline activity is the `faligndata` VIS instruction.

Well, that's the hypothesis, which can be checked by reading the processor documentation (`http://www.sun.com/processors/documentation.html`). Many people do not want to go that far, and there is an alternative approach - use the Analyzer to sample based on the hardware performance counters:

```
$ collect -h FA_pipe_completion,,FM_pipe_completion a.out
Creating experiment database test.2.er ...
$ er_print -limit 3 -func test.2.er
Functions sorted by metric: Exclusive FA_pipe_completion Events

Excl.               Incl.               Excl.               Incl.               Name
FA_pipe_completion  FA_pipe_completion  FM_pipe_completion  FM_pipe_completion
Events              Events              Events              Events
131072040           131072040           0                   0                   <total>
131000768           131000768           0                   0                   _memcpy
    71272               71272           0                   0                   collector_final_counters
```

Using this approach it is clearly visible that all the floating-point add pipeline events come in from the `_memcpy` routine. It is even possible to check which instructions are generating them:

```
          0    0    [?]      720:  ldd         [%o1], %f2
          0    0    [?]      724:  faligndata  %f12, %f14, %f44
          0    0    [?]      728:  ldd         [%o1 + 8], %f4
          0    0    [?]      72c:  faligndata  %f14, %f0, %f46
    3000012    0    [?]      730:  stda        %f32, [%o0] 0xf0
          0    0    [?]      734:  ldd         [%o1 + 16], %f6
          0    0    [?]      738:  faligndata  %f0, %f2, %f32
    3000015    0    [?]      73c:  inc         64, %o0
          0    0    [?]      740:  ldd         [%o1 + 24], %f8
          0    0    [?]      744:  faligndata  %f2, %f4, %f34
## 30000207    0    [?]      748:  ldd         [%o1 + 32], %f10
          0    0    [?]      74c:  faligndata  %f4, %f6, %f36
## 27000189    0    [?]      750:  ldd         [%o1 + 40], %f12
          0    0    [?]      754:  dec         64, %o2
          0    0    [?]      758:  faligndata  %f6, %f8, %f38
## 33000231    0    [?]      75c:  ldd         [%o1 + 48], %f14
          0    0    [?]      760:  faligndata  %f8, %f10, %f40
          0    0    [?]      764:  ldd         [%o1 + 56], %f0
## 35000114    0    [?]      768:  prefetch    [%o1 + 384], #one_read
          0    0    [?]      76c:  faligndata  %f10, %f12, %f42
          0    0    [?]      770:  cmp         %o2, 72
          0    0    [?]      774:  bgu,pt      %icc,0x720
          0    0    [?]      778:  inc         64, %o1
```

Going back to the first case, we can also profile that:

```
Functions sorted by metric: Exclusive FA_pipe_completion Events

Excl.                Excl.                 Name
FA_pipe_completion   FM_pipe_completion
Events               Events
104857640            104857600             <total>
104000312            104000312             main
   857328               857288             collector_final_counters
```

And then dig down to find where in the code the events are happening:

```
...
           0            0      [?]    10c2c:  sll          %i5, 3, %o0
           0            0      [?]    10c30:  add          %o0, %o5, %i3
           0            0      [?]    10c34:  ldd          [%i3], %f6
## 104000312            0      [?]    10c38:  ldd          [%o0 + %o4], %f4
           0            0      [?]    10c3c:  faddd        %f6, %f4, %f4
           0            0      [?]    10c40:  std          %f4, [%i3]
           0            0      [?]    10c44:  inc          %i5
           0            0      [?]    10c48:  cmp          %i5, %o3
           0            0      [?]    10c4c:  bl           0x10c2c
           0            0      [?]    10c50:  nop
...
           0            0      [?]    10c90:  sll          %i5, 3, %o0
           0            0      [?]    10c94:  add          %o0, %o5, %i3
           0            0      [?]    10c98:  ldd          [%i3], %f6
##         0    104000312      [?]    10c9c:  ldd          [%o0 + %o4], %f4
           0            0      [?]    10ca0:  fmuld        %f6, %f4, %f4
           0            0      [?]    10ca4:  std          %f4, [%i3]
           0            0      [?]    10ca8:  inc          %i5
           0            0      [?]    10cac:  cmp          %i5, %o3
           0            0      [?]    10cb0:  bl           0x10c90
           0            0      [?]    10cb4:  nop
...
```

A quick summary of the points are

- The floating-point event counters count more than just floating-point events, so check the profile of applications which have unexpectedly high counts.
- The VIS instructions that go down the floating-point add pipeline are used in `memcpy`, so be suspicious when there are substantially more of this kind of event than the floating-point multiply pipeline event.
- Use the Performance Analyzer (`http://docs.sun.com/app/docs/doc/819-3687`) to dig down to exactly where the events are happening.

`http://blogs.sun.com/d/entry/measuring_floating_point_use_in`

Obtaining Floating-Point Instruction Count

Darryl Gove, March 30, 2006

In "Measuring Floating-Point Use in Applications" on page 295 I talk about how to determine the floating-point content of an application using the tools that are shipped with the OS and the compiler. Since SPOT (`http://cooltools.sunsource.net/spot/`) has now been released, it is appropriate that I talk about how this tool can make it easier.

I'll run SPOT on the same example code that I used last time:

```
double a[1024*1024],b[1024*1024];

void main()
{
  int i,j;
  for (i=0; i<1024*1024;i++) {a[i]=b[i]=0.0;}
  for (j=0;j<100;j++) for (i=0; i<1024*1024;i++) {a[i]+=b[i];}
  for (j=0; j<100; j++) for (i=0; i<1024*1024;i++) {a[i]*=b[i];}
}
```

Then build the code using the Sun Studio 11 compiler:

```
$ /opt/SUNWspro/bin/cc -O -xbinopt=prepare -o blog_test blog_test.c
```

The flag `-xbinopt=prepare` annotates the binary so that BIT (`http://cooltools.sunsource.net/bit/`) can provide instruction count data. Using BIT also requires that the binary be compiled with some amount of optimization, hence the `-O` flag.

An aside comment here is that using optimization on trivial binaries like this can lead to all of the test code being eliminated when the compiler realizes that the array a (for example) is never actually used. However in this case the `ieee-754` standard for floating-point computation protects us - the standard says that in the compliant mode the calculation has to be done *even if* the results are not used (this is in case the side-effects of the calculation are being evaluated). Anyway, because the code uses floating-point values it is not eliminated with optimization.... back to the main thread.

Next the binary is run under SPOT:

```
$ /opt/SUNWspro/extra/bin/spot blog_test
Collect machine statistics
Collect application details
Collect ipc data using ripc
Collect data using BIT
Output ifreq data from bit
Collect clock-based profiling data
Generating html output for time profile data
Done collecting, tidying up reports
```

The next step is to open up the spot report in a web-browser and take a look at the statistics. Rather than take screen shots, I've put the results in as text. The first set of results is from `ripc`:

```
...
Instr                     1215210112
...
=========================================================
  UltraSparc               events       evnt/instr     %
=========================================================
ITLB_miss                      86        0.000    0.0% of Instructions
IC_ref                  784223162        0.645  100.0%
IC_miss                    622250        0.001    0.1% of IC Ref
EC_ic_miss                   5937        0.000    1.0% of IC misses
DC_rd                   495950962        0.408  100.0%
DC_rd_miss               62371609        0.051   12.6%
EC_rd_miss                 716846        0.001    1.1% of DC rd misses
DC_wr                   341751980        0.281  100.0%
DC_wr_miss              375756728        0.309  110.0% of DC wr
EC_ref                  531196786        0.437  100.0%
EC_miss                  45585877        0.038    8.6% of Ref
FP Inst            A=    104879231 M=  93378848   16.3%
=========================================================
...
```

So `ripc` reports 104M instructions that went down the FP Add pipeline, and 93M instructions that went down the FP Mul pipeline. Out of a total instruction count of 12B. So this is about 16% of the instructions.

Since we compiled with the flag `-xbinopt=prepare`, we also get data from BIT, which gives us aggregate data as well as actual instruction counts:

```
...
Instruction frequencies for whole program
Instruction              Executed     (%)
 TOTAL                 1209797385 (100.0)
 float ops              840957953 ( 69.5)
 float ld st            631242753 ( 52.2)
 load store             631242753 ( 52.2)
 load                   419430401 ( 34.7)
 store                  211812352 ( 17.5)
...
```

So ~70% of the instructions were floating point, and 52% of instructions were floating-point memory operations. This leaves about 18% of instructions that were actual floating-point operations. The next section in the BIT report tells us which instructions they were:

```
Instruction              Executed     (%)        Annulled   In Delay Slot
 TOTAL                 1209797385 (100.0)
 lddf                   419430401 ( 34.7)             200               0
 stdf                   211812352 ( 17.5)               0        52690744
 add                    158073838 ( 13.1)               0               0
 prefetch               105381488 (  8.7)               0               0
 faddd                  104857600 (  8.7)               0               0
 fmuld                  104857600 (  8.7)               0               0
...
```

So ~9% of instructions were fp adds, and ~9% were fp muls.

But the bit data gives us more than that, it actually tells us the execution counts for individual instructions in the profile. So let's take a look at one of the hot loops:

```
...
   0.             100           [?]    1189c:  ldd         [%o3 + 8], %f26
   0.             100           [?]    118a0:  ldd         [%o3 + 16], %f28
## 0.        26214300           [?]    118a4:  prefetch    [%o1 + 144], #n_reads
## 0.020     26214300           [?]    118a8:  prefetch    [%g3 + 144], #n_reads
## 0.        26214300           [?]    118ac:  inc         4, %g5
## 0.        26214300           [?]    118b0:  inc         32, %g3
## 0.030     26214300           [?]    118b4:  ldd         [%o1], %f14
## 0.        26214300           [?]    118b8:  faddd       %f20, %f24, %f6
## 0.        26214300           [?]    118bc:  cmp         %g5, %g1
## 0.        26214300           [?]    118c0:  inc         32, %o1
## 0.010     26214300           [?]    118c4:  ldd         [%g3 - 32], %f16
## 0.010     26214300           [?]    118c8:  std         %f6, [%o1 - 56]
## 0.        26214300           [?]    118cc:  faddd       %f22, %f26, %f8
## 0.040     26214300           [?]    118d0:  ldd         [%o1 - 24], %f20
## 0.        26214300           [?]    118d4:  ldd         [%g3 - 24], %f24
## 0.270     26214300           [?]    118d8:  std         %f8, [%o1 - 48]
## 0.        26214300           [?]    118dc:  faddd       %f12, %f28, %f10
## 0.010     26214300           [?]    118e0:  ldd         [%o1 - 16], %f22
## 0.700     26214300           [?]    118e4:  ldd         [%g3 - 16], %f26
## 0.650     26214300           [?]    118e8:  std         %f10, [%o1 - 40]
## 0.        26214300           [?]    118ec:  faddd       %f14, %f16, %f18
## 0.030     26214300           [?]    118f0:  ldd         [%o1 - 8], %f12
## 0.        26214300           [?]    118f4:  ldd         [%g3 - 8], %f28
## 0.        26214300           [?]    118f8:  ble,pt      %icc,0x118a4
## 0.030     26214300           [?]    118fc:  std         %f18, [%o1 - 32]
   0.             100           [?]    11900:  faddd       %f20, %f24, %f0
...
```

From this we can work out that the loop is entered (and left) 100 times, and the average trip count is 26214300/100=262143 times. It is also possible to see all the `fadd` instructions that contribute to the floating-point add count.

http://blogs.sun.com/d/entry/obtaining_floating_point_instruction_count

Floating-Point Multiple Accumulate

Darryl Gove, June 5, 2008

The SPARC64 VI (http://www.fujitsu.com/downloads/SPARCE/others/sparc64vi-extensions.pdf) processor supports floating-point multiply accumulate instructions, also known as FMA or FMAC. These instructions take 3 input registers and one output register and perform the calculation:

```
dest = src1*src2+src3
```

The advantage of these instructions is that they perform the multiply and add operations in the same time as it normally takes to do either a multiply or an add, so the issue rate of floating-point instructions is potentially doubled. To see how they can do this, look at the following binary multiply example:

```
     1010
    * 111
     ----
     1010
    10100
 + 101000
 -------
 1000110
```

You can see that the multiply is really a sequence of adds. Adding one more addition into the process does not really make much difference.

However, we're really dealing with floating-point numbers. So consider the usual sequence of operations when performing a multiplication followed by an addition:

```
temp = round(src1 * src2)
dst  = round(src3 + temp)
```

The floating-point numbers are typically computed in hardware with additional precision, then rounded to fit the register. A *fused* FMA is equivalent to the following operations:

```
dst = round(src1 * src2 + src3)
```

Which performs rounding only at the end of the FMA operation. The single rounding operation during the computation may cause a difference in the least significant bits of the result when compared with the result from using two rounding operations. In theory, because the hardware does the computation in higher precision, and then rounds down, the result should be more accurate. It is possible to support an *unfused* FMA operation, where the middle rounding step is preserved, and the result is identical to the two-step code.

The SPARC64 VI processor supports fused FMA. As we've discussed, using this instruction may cause minor differences to the output, so you need to explicitly give permission to the compiler to generate it. (By default the compiler avoids doing anything that would alter the results of FP computation.) So the flags necessary to generate fused FMA instructions are:

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

A binary compiled to use the FMA instructions will not run on hardware that does not support these instructions. With that in mind it is also acceptable to specify the target processor in the flags, which avoids needing to specify the architecture:

```
-xtarget=sparc64vi -xfma=fused
```

http://blogs.sun.com/d/entry/floating_point_multiple_accumulate

Using DTrace to Locate Floating-Point Traps

Darryl Gove, August 29, 2008

The easiest way to check whether a system is handling floating point traps is to use `kstat`(1M):

```
$ kstat |grep fpu_unfinished
        fpu_unfinished_traps            178164
```

or

```
$ kstat -s fpu_unfinished_traps
module: unix                            instance: 0
name:   fpu_traps                       class:    misc
        fpu_unfinished_traps            178164
```

This reports the number since boot time, so to see if traps are happening, you need to run the command twice and look at the difference.

An alternative way of doing this is to use DTrace to count the number of traps that occur:

```
$ dtrace -n fbt:genunix:_fp_fpu_simulator:entry'{@a[probefunc]=count();}'
dtrace: description 'fbt:genunix:_fp_fpu_simulator:entry' matched 1 probe
^C

  _fp_fpu_simulator                                                 57050
```

Then it's easy to record the pid of the processes generating the traps together with their frequency:

```
dtrace -n fbt:genunix:_fp_fpu_simulator:entry'{@a[pid]=count();}'
dtrace: description 'fbt:genunix:_fp_fpu_simulator:entry' matched 1 probe
^C

    1686                4
    1526            19670
    1589            20924
```

The final thing to do is to look at the function names:

```
$ dtrace -n fbt:genunix:_fp_fpu_simulator:entry'{@a[ustack(1)]=count();}'
dtrace: description 'fbt:genunix:_fp_fpu_simulator:entry' matched 1 probe
^C


              app`func1
            13426
```

http://blogs.sun.com/d/entry/using_dtrace_to_locate_floating

Subnormal Numbers

Darryl Gove, August 28, 2008

Under IEEE-754, floating-point numbers are represented in binary as:

```
Number = signbit * mantissa * 2^exponent
```

There are potentially multiple ways of representing the same number. Using decimal as an example, the number 0.1 could be represented as $1*10^{-1}$ or $0.1*10^{0}$ or even $0.01 * 10^{1}$. The standard dictates that the numbers are always stored with the first bit as a one. In decimal that corresponds to the $1*10^{-1}$ example.

Now suppose that the lowest exponent that can be represented is -100. So the smallest number that can be represented in normal form is $1*10^{-100}$. However, if we relax the constraint that the leading bit be a one, then we can actually represent smaller numbers in the same space. Taking a decimal example, we could represent $0.1*10^{-100}$. This is called a *subnormal number*. The purpose of having subnormal numbers is to smooth the gap between the smallest normal number and zero.

It is very important to realize that subnormal numbers are represented with less precision than normal numbers. In fact, they are trading reduced precision for their smaller size. Hence calculations that use subnormal numbers are not going to have the same precision as calculations on normal numbers. So an application which does significant computation on subnormal numbers is probably worth investigating to see if rescaling (that is, multiplying the numbers by some scaling factor) would yield fewer subnormals, and more accurate results.

The following program will eventually generated subnormal numbers:

```
#include <stdio.h>

void main()
{
  double d=1.0;
  while (d>0) {printf("%e\n",d); d=d/2.0;}
}
```

Compiling and running this program will produce output that looks like:

```
$ cc -O ft.c
$ a.out
...
3.952525e-323
1.976263e-323
9.881313e-324
4.940656e-324
```

The downside with subnormal numbers is that computation on them is often deferred to software, which is significantly slower. As outlined above, this should not be a problem since computations on subnormal numbers should be both rare and treated with suspicion.

However, sometimes subnormals come out as artifacts of calculations, for example, subtracting two numbers that should be equal, but due to rounding errors are just slightly different. In these cases the program might want to flush the subnormal numbers to zero, and eliminate the computation on them. There is a compiler flag that needs to be used when building the main routine called `-fns` which enables the hardware to flush subnormals to zero. Recompiling the above code with this flag yields the following output:

```
$ cc -O -fns ft.c
$ a.out
...
1.780059e-307
8.900295e-308
4.450148e-308
2.225074e-308
```

Notice that the smallest number when subnormals are flushed to zero is $2e^{-308}$ rather than $5e^{-324}$ that is attained when subnormals are enabled.

http://blogs.sun.com/d/entry/subnormal_numbers

Enabling Floating-Point Non-Standard Mode Using LD_PRELOAD

Darryl Gove, August 28, 2008

Subnormal numbers are explained in a "Subnormal Numbers" on page 305, together with the use of the flag `-fns` to flush these to zero if they cause a performance impact.

Of course, it's possible that parts of the code need to be computed with subnormals and parts with them flushed to zero. There are programmatic controls to do this in libsunmath (http://docs.sun.com/source/820-4180/man3m/nonstandard_arithmetic.3m.html) The routines are `nonstandard_arithmetic` and `standard_arithmetic`.

Even if they are occurring in code that cannot be recompiled, it is still possible to disable them. One approach is to write a `LD_PRELOAD` library, containing the following code:

```
#include <sunmath.h>
#include <stdio.h>

#pragma init(go)

void go()
{
  nonstandard_arithmetic();
  printf("NONSTANDARD MODE\n");
}
```

The `printf` is just to demonstrate that the code is actually called. Taking the same code from the previous post we can confirm that this approach works:

```
$ cc -O ft.c
$ a.out
...
4.940656e-324
$ cc -O -G -Kpic libns.c -o libns.so -lsunmath
$ export LD_PRELOAD=./libns.so
$ a.out
NONSTANDARD MODE
...
2.225074e-308
```

http://blogs.sun.com/d/entry/enabling_floating_point_non_standard

◆ ◆ ◆ CHAPTER 8

Parallel Code

This chapter has some discussion of writing parallel applications using OpenMP directives, as well as some discussion on the topic of the overhead of using multiple threads to complete a unit of work.

The Limits of Parallelism

Darryl Gove, October 31, 2008

We all know Amdahl's law. The way I tend to think of it is if you reduce the time spent in the hot region of code, the most benefit you can get is the total time that you initially spent there. However, the original setting for the "law" was parallelization - the runtime improvement depends on the proportion of the code that can be made to run in parallel.

Aside: When I'm looking at profiles of applications, and I see a hot region of code, I typically consider what the improvement in runtime would be if I entirely eliminated the time spent there, or if I halved it. Then use this as a guide as to whether it's worth the effort of changing the code.

The issue with Amdahl is that it's completely unrealistic to consider parallelization without considering issues of the synchronization overhead introduced when you use multiple threads. So let's do that and see what happens. Assume that:

```
P = parallel runtime
S = Serial runtime
N = Number of threads
Z = Synchronisation cost
```

Amdahl would give you:

```
P = S / N
```

The flaw is that I can keep adding processors, and the parallel runtime keeps getting smaller and smaller - why would I ever stop? A more accurate equation would be something like:

```
P = S / N + Z*log(N)
```

This is probably a fair approximation of the cost of synchronization, some kind of binary tree object that synchronizes all the threads. So we can differentiate that:

```
dP/DN = -S / N^2 + Z / N
```

And then solve:

```
0 = -S / N^2 + Z / N
S / N = Z
N = S / Z
```

Ok, it's a bit disappointing, you start off with some nasty looking equation, and end up with a ratio. But let's take a look at what the ratio actually means. Let's suppose I reduce the synchronization cost (Z). If I keep the work constant, then I can scale to a greater number of threads on the system with the lower synchronization cost. Or, if I keep the number of threads constant, I can make a smaller chunk of work run in parallel.

Let's take a practical example. If I do synchronization between threads on a traditional SMP system, then communication between cores occurs at memory latency. Let's say that's ~200ns. Now compare that with a CMT system, where the synchronization between threads can occur through the second level cache, with a latency of ~20ns. That's a 10x reduction in latency, which means that I can either use 10x the threads on the same chunk of work, or I can run in parallel a chunk of work that is 10x smaller.

The logical conclusion is that CMT is a perfect enabler of Microparallelism (`http://blogs.sun.com/d/entry/multicore_expo_available_microparallelisation`). You have both a system with huge numbers of threads, and the synchronization costs between threads are potentially very low.

Now, that's exciting!

`http://blogs.sun.com/d/entry/the_limits_of_parallelism`

Introducing OpenMP: A Portable, Parallel Programming API for Shared Memory Multiprocessors

Nawal Copty, January 2007

Sun Studio compilers (C/C++/Fortran 95) support OpenMP parallelization natively. OpenMP is an emerging standard model for parallel programming in a shared memory environment. It provides a set of pragmas, runtime routines, and environment variables for programmers to easily parallelize their code. This article provides a brief introduction to OpenMP and OpenMP support in Sun Studio compilers. This article is of particular interest to programmers who are new to OpenMP and parallel programming in Fortran, C, or C++.

What Is OpenMP?

OpenMP is a specification for parallelizing programs in a shared memory environment. OpenMP provides a set of pragmas, runtime routines, and environment variables that programmers can use to specify shared-memory parallelism in Fortran, C, and C++ programs.

When OpenMP pragmas are used in a program, they direct an OpenMP-aware compiler to generate an executable that will run in parallel using multiple threads. Little source code modifications are necessary (other than fine tuning to get the maximum performance). OpenMP pragmas enable you to use an elegant, uniform, and portable interface to parallelize programs on various architectures and systems. OpenMP is a widely accepted specification, and vendors like Sun, Intel, IBM, and SGI support it. The OpenMP web site (`http://www.openmp.org/`) has the latest OpenMP specification document.

OpenMP takes parallel programming to the next level by creating and managing threads for you. All you need to do is insert appropriate pragmas in the source program, and then compile the program with a compiler supporting OpenMP and with the appropriate compiler option (Sun Studio uses the `-xopenmp` compiler option). The compiler interprets the pragmas and parallelizes the code. When using compilers that are not OpenMP-aware, the OpenMP pragmas are silently ignored.

While this article gives examples of using OpenMP with C and C++ programs, equivalent pragmas exist for Fortran 95 as well. See the *OpenMP User's Guide* (`http://docs.sun.com/app/docs/doc/819-3694`) for details.

OpenMP Pragmas

The OpenMP specification defines a set of pragmas. A pragma is a compiler directive specifying how to process the block of code that follows the pragma. The most basic pragma is the `#pragma omp parallel` to denote a parallel region.

OpenMP uses the fork-join model of parallel execution. An OpenMP program begins as a single thread of execution, called the initial thread. When a thread encounters a parallel construct, it creates a new team of threads composed of itself and zero or more additional threads, and becomes the master of the new team. All members of the new team (including the master) execute the code inside the parallel construct. There is an implicit barrier at the end of the parallel construct. Only the master thread continues execution of user code beyond the end of the parallel construct.

The number of threads in the team executing a parallel region can be controlled in several ways. One way is to use the environment variable `OMP_NUM_THREADS`. Another way is to call the runtime routine `omp_set_num_threads()`. Yet another way is to use the `num_threads` clause in conjunction with the `parallel` pragma.

OpenMP supports two basic kinds of work-sharing constructs to specify that work in a parallel region is to be divided among the threads in the team. These work-sharing constructs are *loops* and *sections*. The `#pragma omp for` is used for loops, and `#pragma omp sections` is used for sections (blocks of code that can be executed in parallel).

The `#pragma omp barrier` instructs all threads in the team to wait for each other before they continue execution beyond the barrier. There is an implicit barrier at the end of a parallel region. The `#pragma omp master` instructs the compiler that the following block of code is to be executed by the master thread only. The `#pragma omp single` indicates that only one thread in the team should execute the following block of code; this thread may not necessarily be the master thread. You can use the `#pragma omp critical` pragma to protect a block of code that should be executed by a single thread at a time. Of course, all of these pragmas make sense only in the context of a `parallel` pragma (parallel region).

OpenMP Runtime Routines

OpenMP provides a number of runtime routines can be used to obtain information about threads in the program. These include `omp_get_num_threads()`, `omp_set_num_threads()`, `omp_in_parallel()`, and others. In addition, OpenMP provides a number of lock routines that can be used for thread synchronization.

Examples

Using a simple matrix multiplication program you can see how OpenMP can be used to parallelize the program. Consider the following small code fragment that multiplies 2 matrices. This is a very simple example and, if you really want a good matrix multiply routine, you will have to consider cache effects, or use a better algorithm (Strassen's, or Coppersmith and Winograd's, and so on).

```
for (ii = 0; ii < nrows; ii++) {
 for (jj = 0; jj < ncols; jj++) {
  for (kk = 0; kk < nrows; kk++) {
   array1[ii][jj] += array[ii][kk] * array[kk][jj];
  }
 }
}
```

Note that at each level in the loop nest above, the loop iterations can be executed independently of each other. So parallelizing the above code segment is straightforward: Insert the `#pragma omp parallel for` pragma before the outermost loop (`ii` loop). It is beneficial to insert the pragma at the outermost loop, since this gives the most performance gain. In the parallelized loop, variables `array`, `ncols` and `nrows` are shared among the threads, while variables `ii`, `jj`, and `kk` are private to each thread. The preceding code now becomes:

```
#pragma omp parallel for shared(array, array1, ncols, nrows) private(ii, jj, kk)
for (ii = 0; ii < nrows; ii++) {
 for (jj = 0; jj < ncols; jj++) {
  for (kk = 0; kk < nrows; kk++) {
   array1[ii][jj] = array[ii][kk] * array[kk][jj];
  }
 }
}
```

As another example, consider the following code fragment that finds the sum of `f(x)` for `0 <= x < n`.

```
for (ii = 0; ii < n; ii++) {
 sum = sum + some_complex_long_fuction(a[ii]);
}
```

To parallelize the above fragment, the first step could be:

```
#pragma omp parallel for shared(sum, a, n) private(ii, value)
for (ii = 0; ii < n; ii++) {
 value = some_complex_long_fuction(a[ii]);

 #pragma omp critical
 sum = sum + value;
}
```

or better, you can use the reduction clause to get:

```
#pragma omp parallel for shared(a, n) private(ii) reduction(+: sum)
for (ii = 0; ii < n; ii++) {
 sum = sum + some_complex_long_fuction(a[ii]);
}
```

How to Begin

There are several ways to parallelize a program. First, determine if you need parallelization. Some algorithms are not suitable for parallelization. If you are starting a new project, you could choose an algorithm that can be parallelized. It is very important to be sure that the code is correct (serially) before trying to parallelize it. Be sure to maintain timings of your serial run, so that you can decide if parallelization is useful.

Compile the serial version with several optimization options. The compiler can generally perform many more optimizations than you can.

When you are ready to parallelize your program, there are a number of features and tools in Sun Studio that can help you achieve that goal. They are briefly described below.

Automatic Parallelization

Try using the automatic parallelization option of the compiler (`-xautopar`). Delegating parallelization to the compiler allows you to parallelize a program without any effort on your part. The autoparallelizer can also help you identify pieces of code that can be parallelized using

OpenMP pragmas, or point out things in the code that could prevent parallelization (for example, inter-loop dependencies). You can view compiler commentary by compiling your program with the -g flag and using the `er_src.1` utility in Sun Studio, as follows:

```
% cc -g -O -xautopar -c source.c
% er_src source.o
```

Autoscoping

One common type of error in OpenMP programming is scoping errors, where a variable is erroneously shared when it should be scoped as private, or the other way around. The autoscoping feature in the Sun Studio compilers relieves you of the task of determining the scopes of variables. Two extensions to OpenMP are supported: the `__auto` clause and the `default(__auto)` clause. Refer to the *OpenMP User's Guide* (`http://docs.sun.com/app/docs/doc/819-3694`) for details.

`dbx` Debugger

The Sun Studio debugger, `dbx`, is a thread-aware debugger that can help you debug your OpenMP program. To debug your OpenMP program, first compile the program without any optimization by using the `xopenmp=noopt -g` compiler options, and then run `dbx` on the resulting executable. With `dbx`, you can set breakpoints inside a parallel region and step through the code of a parallel region, examine variables that are private to a thread, and so on.

Performance Analyzer

Identify bottlenecks in your program by using the Sun Studio Performance Analyzer. This tool can help you identify hot routines in your program where a large amount of time is spent. The tool also provides work and wait metrics, attributed to functions, source lines, and instructions, that can help you identify bottlenecks in an OpenMP program.

Mixing OpenMP With MPI

MPI (Message Passing Interface) is another model for parallel programming. Unlike OpenMP, MPI spawns multiple processes that then communicate using TCP/IP. Since these processes do not share the same address space, they can run on remote machines (or a cluster of machines). It is difficult to say whether OpenMP or MPI is better. They both have their advantages and disadvantages. What is more interesting is that OpenMP can be used with MPI. Typically, you would use MPI to coarsely distribute work among several machines, and then use OpenMP to parallelize at a finer level on a single machine.

In summary, Sun Studio compilers and tools support OpenMP natively and have many useful features that can help you parallelize your program.

`http://developers.sun.com/solaris/articles/omp-intro.html`

Debugging OpenMP Code

Darryl Gove, October 10, 2007

The compiler flag `-xopenmp` enables the recognition of OpenMP directives. As a side effect it also raises the optimization level to `-x03`. If you're trying to debug the code, then you'll not want the optimisation level raised then you can use the option `-xopenmp=noopt` (`http://docs.sun.com/source/819-0501/5_compiling.html`) which enables the recognition of OpenMP directives but does not increase the optimisation level.

It's also worth compiling with the flags `-xvpara` (`http://docs.sun.com/source/819-3688/cc_ops.app.html#pgfId-1016119`) and `-xloopinfo` (`http://docs.sun.com/source/819-3688/cc_ops.app.html#pgfId-1014869`) which report parallelization information.

`http://blogs.sun.com/d/entry/debugging_openmp_code`

OpenMP 3.0 Specification Released

Darryl Gove, May 13, 2008

The specification for OpenMP 3.0 (`http://www.openmp.org/mp-documents/spec30.pdf`) has been put up on the OpenMP (`http://openmp.org/wp/`) web site. Using the previous OpenMP 2.5 standard, there's basically two supported modes of parallelisation:

- Splitting a loop over multiple threads - each thread is responsible for a range of the iterations.
- Splitting a serial code into sections - each thread executes a section of code.

The large change with OpenMP 3.0 is the introduction of tasks, where a thread can spawn a task to be completed by another thread at an unspecified point in the future. This should make OpenMP amenable to many more situations. An example of using tasks looks like:

```
node * p = head;
while (p)
{
  #pragma omp task
  {
    process(p);
  }
  p = p->next;
}
```

The master thread iterates the linked list generating tasks for processing each element in the list. The brackets around the call to `process(p)` are unnecessary, but hopefully clarify what's happening.

`http://blogs.sun.com/d/entry/openmp_3_0_specification_released`

Extending the OpenMP Profiling API for OpenMP 3.0

Yuan Lin, Friday November 21, 2008

Last Tuesday at the OpenMP BOF of SC08 (`http://sc08.supercomputing.org/`), Oleg Mazurov presented our work on extending the OpenMP profiling API for OpenMP 3.0 (`http://blogs.sun.com/yuanlin/resource/SC2008.pdf`).

The current existing API (`http://www.compunity.org/futures/omp-api.html`) was first published in 2006 and was last updated in 2007. Since then, two more developments now beg for another update - one is for supporting the new OpenMP tasking feature, and the other is for supporting vendor specific extensions.

The extension for tasking support is straightforward. A few events that correspond to the creation, execution, and termination of tasks are added. Also added are a few requests to get the task ID and other properties.

Vendor-specific extensions are implemented essentially by sending a establishing-extension request with a vendor unique ID from the collector tool to the OpenMP runtime library. The OpenMP runtime library accepts the request if it supports the vendor, otherwise rejects it. After a successful rendezvous, the request establishes a new namespace for subsequent requests and events.

One pending issue is how to support multiple vendor agents in one session. It is not that a solution cannot be engineered, but we are waiting for a use case to emerge.

During the execution of an OpenMP program, any arbitrary program event can be associated with:

- an OpenMP state
- a user callstack
- a node in the thread tree with parallel region ID's and OpenMP thread IDs along the path
- a node in the task tree with task IDs along the path.

Because the execution of an OpenMP task may be asynchronous, and the executing thread may be different from the encountering thread, getting the user callstack of an event that happened within a task becomes tricky.

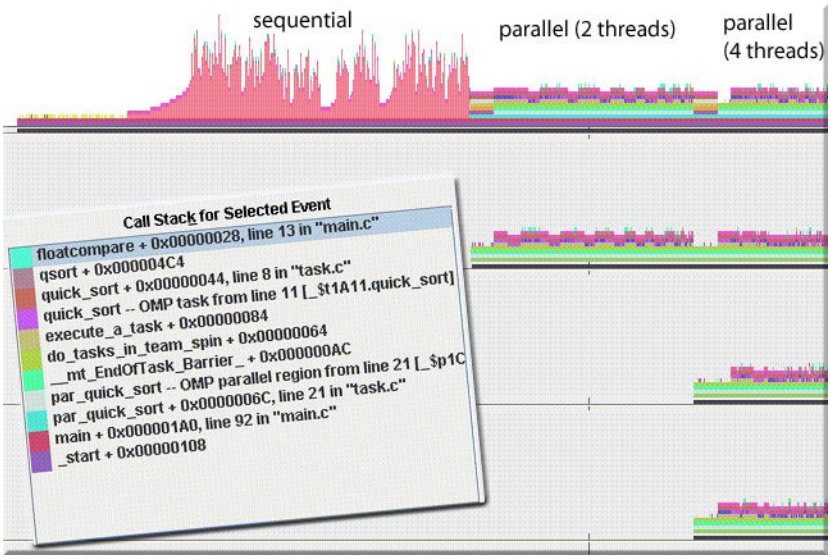
At our Sun booth at SC08, we demoed a prototype Performance Analyzer that can present user callstacks in a cool way when OpenMP tasks are involved.

Take a simple quick sort code for an example.

```
void quick_sort ( int lt,  int rt,  float *data )  {
    int md = partition( lt,  rt,  data );
    #pragma omp task
    quick_sort( lt,  md - 1,  data );
```

```
        #pragma omp task
        quick_sort( md + 1,  rt,  data );
    }
```

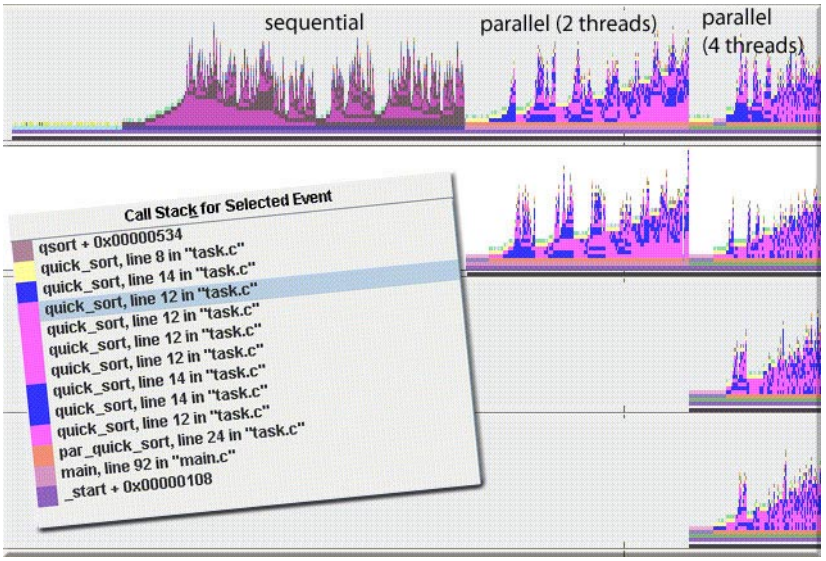
The following figure shows the timeline display of one execution of the program. The same original data are sorted three times, once sequential, once using two threads, and once using four threads.



The spikes in callstacks in the sequential sort show the recursive nature of the quick sort. But when you look at the parallel sort, the callstacks are flat. That's because each call to `quick_sort()` is now a task, and the tasking execution essentially changes the recursive execution into a work-list execution. The low-level callstack in the above figure shows close to what actually happens in one thread.

While these pieces of information are useful in showing the execution details, they do not help answer the question of which tasks are actually being executing. Where was the current executing task created? In the end, the user needs to debug the performance problem in the user's code (not in the OpenMP runtime). Representing information close to the user program logic is crucial.

The following figure shows the time line and user callstacks in the user view constructed by our prototype tool. Notice the callstacks in the parallel run are almost the same as in the sequential run. In the timeline, it is just like the work in the sequential run is being distributed among the threads in the parallel run. Isn't this what happens intuitively when you parallelize a code using OpenMP? :-)



http://blogs.sun.com/yuanlin/entry/extending_the_openmp_profiling_api

Index

H

L

M

R

S

V

W

The SystemDeveloper's Edge
April 2011
Author: Darryl Gove

Oracle Corporation
World Headquarters
500 Oracle Parkway
Redwood Shores, CA 94065
U.S.A.

Worldwide Inquiries:
Phone: +1.650.506.7000
Fax: +1.650.506.7200

oracle.com

Oracle is committed to developing practices and products that help protect the environment

Hardware and Software, Engineered to Work Together