Abstract
This white paper is targeted at users, administrators, and developers of parallel applications on the Solaris™ OS. It describes the current challenges of performance analysis and tuning of complex applications, and the tools and techniques available to overcome these challenges. Several examples are provided, from a range of application types, environments, and business contexts, with a focus on the use of the Solaris Dynamic Tracing (DTrace) utility. The examples cover virtualized environments, multicore platforms, I/O and memory bottlenecks, Open Multi-Processing (MP) and Message Passing Interface (MPI).
# Table of Contents

**Introduction**........................................................................................................1
**Challenges of Tuning Parallel Code** ....................................................................1
**Some Solaris™ OS Performance Analysis Tools** ............................................... 2
   - DTrace .................................................................................................................. 2
   - System Information Tools .................................................................................. 2
   - Sun™ Studio Performance Analyzer .................................................................. 3
   - Combining DTrace with Friends ........................................................................ 3
**Purpose of this Paper** .......................................................................................... 4

**DTrace and its Visualization Tools**.................................................................... 5
   - Chime ..................................................................................................................... 5
   - DLight .................................................................................................................... 6
   - gnuplot ................................................................................................................... 7

**Performance Analysis Examples** ....................................................................... 8
   - Analyzing Memory Bottlenecks with cputrack, busstat, and the Sun Studio
     Performance Analyzer ......................................................................................... 8
   - Improving Performance by Reducing Data Cache Misses ................................ 8
   - Improving Scalability by Avoiding Memory Stalls Using the Sun Studio Analyzer 11
   - Memory Placement Optimization with OpenMP ............................................. 16
   - Analyzing I/O Performance Problems in a Virtualized Environment ................. 19
**Using DTrace to Analyze Thread Scheduling with OpenMP** ............................... 21
**Using DTrace with MPI** ...................................................................................... 27
   - Setting the Correct mpirun Privileges ............................................................... 28
   - A Simple Script for MPI Tracing ....................................................................... 28
   - Tracking Down MPI Memory Leaks ................................................................... 30
   - Using the DTrace mpiperuse Provider ............................................................. 33
   - A Few Simple mpiperuse Usage Examples ....................................................... 35

**Conclusion** ........................................................................................................... 37
**Acknowledgments** .............................................................................................. 37
**References** ........................................................................................................... 38
Chapter 1

Introduction

Most computers today are equipped with multicore processors and manufacturers are increasing core and thread counts, as the most cost effective and energy efficient route to increased processing throughput. While the next level of parallelism — grid computers with fast interconnects — is still very much the exclusive domain of high performance computing (HPC) applications, mainstream computing is developing ever increasing levels of parallelism. This trend is creating an environment in which the optimization of parallel code is becoming a core requirement of tuning performance in a range of application environments. However, tuning parallel applications brings with it complex challenges that can only be overcome with increasingly advanced tools and techniques.

Challenges of Tuning Parallel Code

Single threaded code, or code with a relatively low level of parallelism, can be tuned by analyzing its performance in test environments, and improvements can be made by identifying and resolving bottlenecks. Developers tuning in this manner are confident that any results they obtain in testing can result in improvements to the corresponding production environment.

As the level of parallelism grows, due to interaction between various concurrent events typical of complex production environments, it becomes more difficult to create a test environment that closely matches the production environment. In turn, this means that analysis and tuning of highly multithreaded, parallel code in test environments is less likely to improve its performance in production environments. Consequently, if the performance of parallel applications is to be improved, they must be analyzed in production environments. At the same time, the complex interactions typical of multithreaded, parallel code are far more difficult to analyze and tune using simple analytical tools, and advanced tools and techniques are needed.

Conventional tools used for performance analysis were developed with low levels of parallelism in mind, are not dynamic, and do not uncover problems relating to timing or transient errors. Furthermore, these tools do not provide features that allow the developer to easily combine data from different processes and to execute an integrated analysis of user and kernel activity. More significantly, when these tools are used, there is a risk that the act of starting and stopping, or tracing can affect and obscure the problems that require investigation, thus rendering any results found suspect and creating a potential risk to business processing.

1. In the Solaris Operating System (OS), these include various analysis and debugging tools such as truss(1), apptrace(1) and mdb(1). Other operating environments have similar tools.
Some Solaris™ OS Performance Analysis Tools

In addition to performance and scalability, the designers of the Solaris OS set observability as a core design goal. Observability is achieved by providing users with tools that allow them to easily observe the inner workings of the operating system and applications, and analyze, debug, and optimize them. The set of tools needed to achieve this goal was expanded continuously in each successive version of the Solaris OS, culminating in the Solaris 10 OS with DTrace — arguably the most advanced observability tool available today in any operating environment.

DTrace

DTrace provides the Solaris 10 OS user with the ultimate observability tool — a framework that allows the dynamic instrumentation of both kernel and user level code. DTrace permits users to trace system data safely without affecting performance. Users can write programs in D — a dynamically interpreted scripting language — to execute arbitrary actions predicated on the state of specific data exposed by the system or user component under observation. However, currently DTrace has certain limitations — most significantly, it is unable to read hardware counters, although this might change in the near future.

System Information Tools

There are many other tools that are essential supplements to the capabilities of DTrace that can help developers directly observe system and hardware performance characteristics in additional ways. These tools and their use are described in this document through several examples. There are other tools that can be used to provide an overview of the system more simply and quickly than doing the equivalent in DTrace. See Table 1 for a non-exhaustive list of these tools.

Table 1. Some Solaris system information tools

<table>
<thead>
<tr>
<th>Command</th>
<th>Purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>intrstat(1M)</td>
<td>Gathers and displays run-time interrupt statistics per device and processor or processor set</td>
</tr>
<tr>
<td>busstat(1M)</td>
<td>Report memory bus related performance statistics</td>
</tr>
<tr>
<td>cputrack(1M), cpustat(1M)</td>
<td>Monitor system and/or application performance using CPU hardware counters</td>
</tr>
<tr>
<td>trapstat(1M)</td>
<td>Reports trap statistic per processor or processor set</td>
</tr>
<tr>
<td>prstat(1M)</td>
<td>Report active process statistics</td>
</tr>
<tr>
<td>vmstat(1M)</td>
<td>Report virtual memory statistics</td>
</tr>
</tbody>
</table>

2. See discussion of the proposed cpc DTrace provider and the DTrace limitations page in the Solaris internals wiki through links in “References” on page 38
Sun™ Studio Performance Analyzer

The Sun Studio Performance Analyzer and the Sun Studio Collector are components of the Sun Studio integrated development environment (IDE). The collector is used to run experiments to collect performance related data for an application, and the analyzer is used to analyze and display this data with an advanced GUI.

The Sun Studio Performance Analyzer and Collector can perform a variety of measurements on production codes without special compilation, including clock- and hardware-counter profiling, memory allocation tracing, Message Passing Interface (MPI) tracing, and synchronization tracing. The tool provides a rich graphical user interface, supports the two common programming models for HPC — OpenMP and MPI — and can collect data on lengthy production runs.

Combining DTrace with Friends

While DTrace is an extremely powerful tool, there are other tools that can be executed from the command line, which makes them more appropriate to view a system’s performance. These tools can be better when a quick, initial diagnosis is needed to answer high level questions. A few examples of this type of question are:

• Are there a significant number of cache misses?
• Is a CPU accessing local memory or is it accessing memory controlled by another CPU?
• How much time is spent in user versus system mode?
• Is the system short on memory or other critical resources?
• Is the system running at high interrupt rates and how are they assigned to different processors?
• What are the system’s I/O characteristics?

An excellent example is the use of the `prstat` utility which, when invoked with the `-lm` option, provides microstates per thread that offer a wide view of the system. This view can help identify what type of further investigation is required, using DTrace or other more specific tools, as described in Table 2.
Table 2. Using the `prstat` command for initial analysis

<table>
<thead>
<tr>
<th>prstat column</th>
<th>Meaning</th>
<th>Investigate if high values are seen</th>
</tr>
</thead>
<tbody>
<tr>
<td>USR</td>
<td>% of time in user mode</td>
<td>Profile user mode with DTrace using either <code>pid</code> or <code>profile</code> providers</td>
</tr>
<tr>
<td>SYS</td>
<td>% of time in system mode</td>
<td>Profile the kernel</td>
</tr>
<tr>
<td>LCK</td>
<td>% of time waiting for locks</td>
<td>Use the <code>plockstat</code> DTrace provider or the <code>plockstat(1M)</code> utility to see which user locks are used extensively</td>
</tr>
<tr>
<td>SLP</td>
<td>% of time sleeping</td>
<td>Use the <code>sched</code> DTrace provider and view call stacks with DTrace to see why the threads are sleeping</td>
</tr>
<tr>
<td>TFL or DFL</td>
<td>% of time processing text or data page faults</td>
<td>Use the <code>vminfo</code> DTrace provider to identify the source of the page faults</td>
</tr>
</tbody>
</table>

Once the various tools are used to develop an initial diagnosis of a problem, it is possible to use DTrace, Sun Studio Performance Analyzer, and other more specific tools for an in-depth analysis. This approach is described in further detail in the examples to follow.

Purpose of this Paper

This white paper is targeted at users, administrators, and developers of parallel applications. The lessons learned in the course of performance tuning efforts are applicable to a wide range of similar scenarios common in today's computing environments in different public sector, business, and academic settings.
Chapter 2
DTrace and its Visualization Tools

DTrace is implemented through thousands of probes (also known as tracing points) embedded in the Solaris 10 kernel, utilities, and in other software components that run on the Solaris OS. The kernel probes are interpreted in kernel context, providing detailed insight into the inner workings of the kernel. The information exposed by the probes is accessed through scripts written in the D programming language. Administrators can monitor system resources to analyze and debug problems, developers can use it to help debugging and performance tuning, and end-users can analyze applications for performance and logic problems.

The D language provides primitives to print text to a file or the standard-output of a process. While suitable for brief, simple analysis sessions, direct interpretation of DTrace output in more complex scenarios can be quite difficult and lengthy. A more intuitive interface that allows the user to easily analyze DTrace output is provided by several tools — Chime, DLight, and gnuplot — that are covered in this paper.

For detailed tutorials and documentation on DTrace and D, see links in “References” on page 38.

Chime

Chime is a standalone visualization tool for DTrace and is included in the NetBeans DTrace GUI plug-in. It is written in the Java™ programming language, supports the Python and Ruby programming languages, includes predefined scripts and examples, and uses XML to define how to visualize the results of a DTrace script. The Chime GUI provides an interface that allows the user to select different views, to easily drill down into the data, and to access the underlying DTrace code.

Chime includes the following additional features and capabilities:

- Graphically displays aggregations from arbitrary DTrace programs
- Displays moving averages
- Supports record and playback capabilities
- Provides an XML based configuration interface for predefined plots

When Chime is started it displays a welcome screen that allows the user to select from a number of functional groups of displays through a pull-down menu, with Initial Displays the default group shown. The Initial Displays group contains various general displays including, for example, System Calls, that when selected brings up a graphical view showing the application executing the most system calls, updated
every second (Figure 8, in the section “Chime Screenshots” on page 39). The time interval can be changed with the pull-down menu at the bottom left corner of the window.

Figure 1. File I/O displayed with the rfileio script from the DTraceToolkit

Another useful choice available from the main window allows access to the probes based on the DTraceToolkit, that are much more specific than those in Initial Displays. For example, selecting rfileio brings up a window (Figure 1) with information about file or block read I/O. The output can be sorted by the number of reads, helping to identify the file that is the target of most of the system read operations at this time — in this case, a raw device. Through the context menu of the suspect entry, it is possible to bring up a window with a plotting of the reads from this device over time (Figure 9, on page 39).

For further details on Chime, see links in “References” on page 38.

DLight

DLight is a GUI for DTrace, built into the Sun Studio 12 IDE as a plug-in. DLight unifies application and system profiling and introduces a simple drag-and-drop interface that interactively displays how an application is performing. DLight includes a set of DTrace scripts embedded in XML files. While DLight is not as powerful as the DTrace
command line interface, it does provide a useful way to see a quick overview of standard system activities and can be applied directly to binaries using the DTrace scripts included with it.

After launching DLight, the user selects a Sun Studio project, a DTrace script, or a binary executable to apply DLight to. For the purpose of this discussion, the tar utility was used to create a load on the system and DLight was used to observe it. The output of the tar command is shown in the bottom right section of the display (Figure 2). The time-line for the selected view (in this case File System Activity) is shown in the middle panel of the display. Clicking on a sample prints the details associated with it in the lower left panel of the display. The panels can be detached from the main window and their size adjusted.

![Figure 2. DLight main user interface window](image)

For further details on DLight, see links in “References” on page 38.

**gnuplot**

gnuplot is a powerful, general purpose, open-source, graphing tool that can help visualize DTrace output. gnuplot can generate two- and three-dimensional plots of DTrace output in many graphic formats, using scripts written in gnuplot’s own text-based command syntax. An example using gnuplot with the output of the cputrack and busstat commands is provided in the section “Analyzing Memory Bottlenecks with cputrack, busstat, and the Sun Studio Performance Analyzer” on page 8, in the “Performance Analysis Examples” chapter.
Chapter 3

Performance Analysis Examples

In the following sections, several detailed examples of using DTrace and other Solaris observability tools to analyze the performance of Solaris applications are provided. These use-cases include the following:

1. Use `busstat`, `cputrack`, and the Sun Studio Performance Analyzer to explain performance of different SPARC® processors and scalability issues resulting from data cache misses and memory bank stalls
2. Use `cputstat` to analyze memory placement optimization with OpenMP code on non-uniform memory access (NUMA) architectures
3. Use DTrace to resolve I/O related issues
4. Use DTrace to analyze thread scheduling with Open Multi-Processing (MP)
5. Use DTrace to analyze MPI performance

Analyzing Memory Bottlenecks with `cputrack`, `busstat`, and the Sun Studio Performance Analyzer

Most applications require fast memory access to perform optimally. In the following sections, several use-cases are described where memory bottlenecks cause sub-optimal application performance, resulting in sluggishness and scalability problems.

Improving Performance by Reducing Data Cache Misses

The `cputstat(1M)` and `cputrack(1M)` commands provide access to CPU hardware counters. `busstat(1M)` reports system-wide bus-related performance statistics. These commands are passed sets of platform dependent event-counters to monitor.

In this example, the performance of two functionally identical instances of a small program that sums two matrices are analyzed — `add_col` and `add_row`. `add_col` uses the data-cache inefficiently, with many cache misses, while `add_row` uses the data-cache efficiently. The insights provided by these examples are applicable in a wide range of contexts.
The following listing, add_col.c, shows the implementation of add_col:

```c
#include <stdlib.h>
#define SIZE 12000L
double a[SIZE][SIZE], b[SIZE][SIZE], c[SIZE][SIZE];
int main(int argc, char *argv[]) {
  size_t i, j;
  for (i = 0; i < SIZE; i++)
    for (j = 0; j < SIZE; j++)
      c[j][i] = a[j][i] + b[j][i];
}
```

add_row.c is identical to add_col.c, except that line 8 — the line that executes the addition of the matrix cells, runs across rows and takes the following form:

```c
8   c[i][j] = a[i][j] + b[i][j];
```

The programs are compiled without optimization or prefetching, to side-step the compiler’s ability to optimize memory access (that would make the differences between the programs invisible) and with large pages to avoid performance problems resulting from the relatively small TLB cache available to the UltraSPARC T1® processor and the UltraSPARC T2® processor:

```
# cc -xpagesize=256m -xprefetch=no -o add_row add_row.c
# cc -xpagesize=256m -xprefetch=no -o add_col add_col.c
```

### Using cputrack to Count Data Cache Misses

The programs are first run with cputrack, while tracing data cache misses as reported by the CPU’s hardware counters. The output is piped through the sed command to remove the last line in the file, since it contains the summary for the run and should not be plotted, and directed to cputrack_add_col.data and cputrack_add_row.data, as appropriate:

```
# cputrack -n -ef -c pic0=DC_miss,pic1=Instr_cnt .\add_row | \sed \$d\ > cputrack_add_row.data
# cputrack -n -ef -c pic0=DC_miss,pic1=Instr_cnt .\add_col | \sed \$d\ > cputrack_add_col.data
```

The --n option is used to prevent the output of column headers, --e instructs cputrack to follow all exec(2), or execve(2) system calls, --f instructs cputrack to follow all child processes created by the fork(2), fork1(2), or vfork(2) system calls. The --c option is used to specify the set of events for the CPU performance instrumentation counters (PIC) to monitor:

- **pic0=DC_miss** — counts the number of data-cache misses
- **pic1=Instr_cnt** — counts the number of instructions executed by the CPU

Plotting the results of the two invocations with gnuplot (Figure 3) shows that the number of cache-misses in the column-wise matrix addition is significantly higher
than in the row-wise matrix addition, resulting in a much lower instruction count per second and a much longer execution time for the column-wise addition.

Figure 3. Comparison of data-cache misses in row-wise and column-wise matrix addition

Using busstat to Monitor Memory Bus Use

The busstat command is only able to trace the system as a whole, so it is necessary to invoke it in the background, run the program that is to be measured, wait for a period of time, and terminate it. This must be done on a system that is not running a significant workload at the same time, since it is not possible to isolate the data generated by the test program from the other processes. The following short shell-script, run_busstat.sh, is used to run the test:

```
1 #!/bin/sh
2 busstat -n -w dram0,pic0=mem_reads 1 &
3 pid=$!
4 eval $@
5 sleep 1
6 kill $pid
```

The command to run — in this case add_row or add_col — is passed to the script as a command line parameter and the script performs the following steps:

- Runs busstat in the background in line 2
- Saves busstat’s process ID in line 3
- Runs the command in line 4
- After the command exits, the script waits for one second in line 5
- Terminates busstat in line 6

4. The gnuplot program used to generate the graph in Figure 3 is listed in the section “Listings for Matrix Addition cputrack Example” on page 40, followed by the listings of cputrack_add_col.data, and cputrack_add_row.data.
The \texttt{busstat} option is used to prevent the output of column headers, \texttt{-w} specifies the device to be monitored — in this case the memory bus (dram0) and \texttt{pic0=mem\_reads} defines the counter to count the number of memory reads performed by DRAM controller \#0 over the bus. The number 1 at the end of the command instructs \texttt{busstat} to print out the accumulated information every second.

The \texttt{run\_busstat.sh} script is used to run \texttt{add\_col} and \texttt{add\_row}, directing the output to \texttt{busstat\_add\_col.data} and \texttt{busstat\_add\_row.data}, as appropriate:

\begin{verbatim}
# ./run\_busstat\_sh add\_col > busstat\_add\_col.data
# ./run\_busstat\_sh add\_row > busstat\_add\_row.data
\end{verbatim}

Plotting the results of the two invocations with \texttt{gnuplot} (Figure 4) shows that the number of cache-misses in the column-wise addition seen by \texttt{cputrack} is reflected in the results reported by \texttt{busstat}, that shows a much larger number of reads over the memory bus\footnote{The \texttt{gnuplot} program used to generate the graph in Figure 4, is listed in the section "Listings for Matrix Addition busstat Example" on page 42, followed by the listings of \texttt{busstat\_add\_col.data}, and \texttt{busstat\_add\_row.data}.}

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure4.png}
\caption{Comparison of memory bus performance in row-wise and column-wise matrix addition}
\end{figure}

\section*{Improving Scalability by Avoiding Memory Stalls Using the Sun Studio Analyzer}

In this example, a Sun Fire™ T2000 server with a 1 GHz UltraSPARC T1 processor, with eight CPU cores and four threads per core is compared with a Sun Fire V100 server with a 550 MHz UltraSPARC-Ile single core processor. The Sun Fire V100 server was chosen since its core design is similar to the Sun Fire T2000 server. The software used
for the comparison is *John the Ripper* — a password evaluation tool that includes a
data encryption standard (DES) benchmark capability, has a small memory footprint,
and performs no floating point operations. Multiple instances of *John the Ripper* are
run in parallel to achieve any level of parallelism required.

The benchmark is compiled with the Sun Studio C compiler using the `-fast` and
`-arch=native64` options, and a single instance is executed on each of the servers.
Due to its higher clock-rate, it may seem reasonable to expect the Sun Fire T2000
server to perform twice as fast as the Sun Fire V100 server for a single invocation,
however, the UltraSPARC T1 processor is designed for maximum multithreaded
throughput so this expectation might not be valid. To demonstrate the reason for
this result it is necessary to first quantify it in further detail. With the cputrack
command it is possible to find out the number of instructions per cycle:

```
t2000> cputrack -T 5 -t -c pic1=Instr_cnt ./john -test -format:DES
```

The cputrack command is used to monitor the instruction count by defining
`pic1=Instr_cnt`, and the `-t` option sets the sampling interval to five seconds. The
output generated by this invocation indicates an average ratio of 0.58 instructions
per CPU cycle, calculated by dividing the total instruction count from the bottom cell
of the `pic1` column by the number of cycles in the bottom cell of the `%tick` column:

<table>
<thead>
<tr>
<th>time</th>
<th>lwp</th>
<th>event</th>
<th>%tick</th>
<th>pic1</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.019</td>
<td>1</td>
<td>tick</td>
<td>5013226812</td>
<td>3401744911</td>
</tr>
<tr>
<td>10.029</td>
<td>1</td>
<td>tick</td>
<td>6412314672</td>
<td>3182470158</td>
</tr>
<tr>
<td>10.032</td>
<td>1</td>
<td>exit</td>
<td>11426217732</td>
<td>6584425110</td>
</tr>
</tbody>
</table>

When the same command is run on the Sun Fire V100 server, the average
performance per core of the UltraSPARC-IIe architecture is higher, resulting in a ratio
of 2.05 instructions per CPU cycle.

The lower single threaded performance of the UltraSPARC T1 processor can be
attributed to the differing architectures — the single in-order issue pipeline of
each core of the UltraSPARC T1 processor is designed for lower single threaded
performance than the four-way superscalar architecture of the UltraSPARC IIe
processor. In this case, it is the different CPU designs that explain the results seen.

Next, the benchmark is run on the Sun Fire T2000 server in multiple process
invocations to check its scalability. While one to eight parallel processes scale
linearly as expected, at 12 processes performance is flat, with 16 processes, the
throughput goes down to a level comparable with that of four processes, and at 32
processes, the total throughput is similar to that of a dual process invocation.
Due to the fact that the UltraSPARC T1 processor has a 12-way L2 cache, it seems reasonable to suspect that the scaling problem, where throughput drops rapidly for more than 12 parallel invocations, is a result of L2 cache misses. To validate this assumption, an analysis of the memory access patterns is needed to see if there is a significant increase in memory access over the memory bus as the application scales.

This analysis can take advantage of the superior user interface of the Sun Studio Performance Analyzer. To use it, John the Ripper is first recompiled, adding the –g and –xhwcprowf options to each compilation and link. These options instruct the collector to trace hardware counters and the analyzer to use them in analysis. In addition, the analyzer requires UltraSPARC T1 processor specific hints to be added to one of the .er.c files, that instruct the collector to sample additional information, including max memory stalls on a specific function. See the section “Sun Studio Analyzer Hints to Collect Cache Misses on an UltraSPARC T1® Processor” on page 44 in the Appendix for a list of these hints.

Next John the Ripper is run with the collector:

```
t2000> collect -p +on -A copy -F on ./john --test --format:DES
```

The –p +on option triggers the collection of clock-based profiling data with the default profiling interval of approximately 10 milliseconds, the –A copy option requests that the load objects used by the target process are copied into the recorded experiment, and the –F on option instructs the collector to record the data on descendant processes that are created by fork and exec system call families.

Figure 5. Profile of John the Ripper on a Sun Fire T2000 server

Once the performance data is collected, the analyzer is invoked and the data collected for John the Ripper selected. The profile information (Figure 5) shows that

---

6. –g instructs the compiler to produce additional symbol table information needed by the analyzer, and –xhwcprowf, in conjunction with –g, instructs the compiler to generate information that helps tools associate profiled load and store instructions with data-types and structure members.
of the total of 9.897 seconds the process has run on the CPU, it stalled on memory access for 9.387 seconds, or nearly 95% of the time. Furthermore, a single function — DES_bs_crypt_25 — is responsible for almost 99% of the stalls.

When the hardware counter data related to the memory/cache subsystem is examined (Figure 6) it is seen that of the four UST1_Bank Memory Objects, stalls were similar on objects 0, 1 and 2 at 1.511, 1.561, and 1.321 seconds respectively while on object 3 it was 4.993 seconds, or 53% of the total.

Further analysis (Figure 7) shows that 85% of the memory stalls are on a single page, with half of all the stalls associated with a single memory bank. Note that the Solaris OS uses larger pages for the sun4v architecture to help small TLB caches. This can create memory hot spots, as illustrated here, since it limits cache mapping flexibility and when the default page coloring is retained, it can cause unbalanced L2 cache use. In the case of the Sun Fire T2000 server, this behavior is changed by adding a `set consistent_coloring=2` in `/etc/system` or by applying patch 118833-03 or later. Note that this patch can be installed as part of a recommended patch set or a Solaris update.

---

7. To improve performance, the Solaris OS uses a page coloring algorithm which allocates pages to a virtual address space with a specific predetermined relationship between the virtual address to which they are mapped and their underlying physical address.
Note: While the change in the consistent_coloring parameter in /etc/system improves the performance of *John the Ripper*, it can adversely affect the performance of other applications.

The `busstat(1M)` command can be used to demonstrate the effect of the change in the consistent_coloring parameter by measuring the memory stalls before and after modifying the page coloring, when running *John the Ripper* with 1 to 32 invocations:

```
# busstat -w dram0, pic0=bank_busy_stalls, pic1=mem_read_write 2
```

The `-w` option is used to instruct `busstat` to monitor the memory bus — `dram0`, while `pic0=bank_busy_stalls` assigns the first counter to count the number of memory stalls, and `pic1=mem_read_write` assigns the second counter to count the number of memory operations over the bus. The number 2 at the end of the command instructs `busstat` to print out the accumulated information every two seconds. The total number of memory accesses across the memory bus seen in the results is 337 million, with 904 million memory stall events, indicating a very low L2 cache hit-rate.

After the consistent_coloring setting is modified to 2, *John the Ripper* executed, and memory access measured again with the `busstat` command, the total number of memory accesses across the memory bus goes down to 10.6 million, with 17.7 million memory stall events. This reduction of 97% in the number of memory accesses over the memory bus indicates a huge improvement in the L2 cache hit-rate. Running the multiple invocations of *John the Ripper*, scalability
improves drastically, with 32 parallel invocations resulting in 1.7 million DES encryptions/second, compared to less than 400,000 DES encryptions/second before the consistent_coloring setting was changed.

**Memory Placement Optimization with OpenMP**

The Solaris OS optimizes memory allocation for NUMA architectures. This is currently relevant to UltraSPARC IV+, UltraSPARC T2+, SPARC64-VII®, AMD Opteron™, and Intel® Xeon® processor 5500 series multiprocessor systems. Each CPU has its own memory controller and through it — its own memory.

On the systems with NUMA CPUs running the Solaris OS, by default, the first time uninitialized memory is accessed or when memory is allocated due to a page fault, it is allocated on the memory local to the CPU that runs the thread accessing the memory. This mechanism is called memory placement optimization. However, if the first thread accessing the memory is moved to another CPU, or if the memory is accessed by a thread running on another CPU, this can result in higher memory latency.

The program used to test this behavior implements a matrix by vector multiplication that scales almost linearly with the number of CPUs, for large working sets. The benefits of first-touch optimization will show in the working routines, where the data is accessed repeatedly. However, the optimization itself must be applied when the data is initialized, since that is when the memory is allocated. By making use of the Solaris MPO the code can ensure that data gets initialized by the thread that processes the data later. This results in reduced memory latency and improved memory bandwidth. Following is the initialization code that scales almost linearly for large matrix and vector sizes:

```c
1  int init_dat (int m, int n, double *v1, double **ml, double *v2)
2  {
3      int i, j;
4      #pragma omp parallel default(none) \  
5          shared(m,n,v1,ml,v2) private(i,j)
6      {
7          #pragma omp for
8              for (j=0; j<n; j++)
9                  v2[j] = 1.0;
10         #pragma omp for
11             for (i=0; i<m; i++) {
12                 v1[i] = -1957.0;
13                 for (j=0; j<n; j++)
14                     ml[i][j] = i;
15             } /*-- End of omp for --*/
16         } /*-- End of parallel region --*/
17     }
```
This code implements two top level loops:

- Lines 8 and 9 write sequentially to floating point vector v2
- Lines 11 to 15 implement an outer loop, in which the floating point elements of vector v1 are set to -1957.0 (line 12)
- In the inner loop, for every iteration of the external loop, a sequence of elements of the matrix m1 is set to the value of the outer loop’s index (lines 13 and 14)

The three `#pragma omp` compile time directives instruct the compiler to use the OpenMP API to generate code that executes the top level loops in parallel. OpenMP generates code that spreads the loop workload over several threads. In this case, the initialization is the first time the memory written to in these loops is accessed, and first-touch attaches the memory the threads use to the CPU they run on. Here the code is executed with one or two threads on a four-way dual-core AMD Opteron system, with two cores — 1 and 2 — on different processors defined as a processor set using the `setSid(1M)` command. This setup helps ensure almost undisturbed execution and makes the `cpustat(1M)` command output easy to parse, while optimizing memory allocation. The `cpustat` command is invoked as follows:

```bash
cpustat -c 'pic0=NB_mem_ctrlr_page_access,umask0=0x01,\pic1=NB_mem_ctrlr_page_access,umask1=0x02,\pic2=NB_mem_ctrlr_page_access,umask2=0x04,sys’ 2
```

The command line parameters instruct `cpustat` to read the AMD Opteron CPU’s hardware performance counters to monitor memory accesses broken down by internal latency. This is based on the `umask` value, which associates each performance instrumentation counter (PIC) with the L1 cache, L2 cache, and main memory, respectively. The `sys` token directs `cpustat` to execute in system mode, and the sampling interval is set to two seconds.

Note: Many PICs are available for the different processors that run the Solaris OS and are documented in the processor developers’ guides. See “References” on page 32 for a link to the developer’s guide for the AMD Opteron processor relevant to this section.

To illustrate the affect of NUMA and memory placement on processing, the code is executed in the following configurations:

- One thread without first-touch optimization — achieved by setting the NCPUS environment variable to 1
- Two threads without first-touch optimization — achieved by removing all of the `#pragma omp` directives from the code, resulting in code where a single thread initializes and allocates all memory but both threads work on the working set
- Two threads, with first-touch optimization — achieved by setting the NCPUS environment variable to 2 and retaining all three `#pragma omp` directives

---

8. The OpenMP API supports multiplatform shared-memory parallel programming in C/C++ and Fortran. OpenMP is a portable, scalable model that gives shared-memory parallel programmers a simple and flexible interface to develop parallel applications.
Following are sample results of the invocations with one thread without first-touch optimization, that show that the vast majority of memory accesses is handled by CPU #2:

<table>
<thead>
<tr>
<th>time</th>
<th>cpu</th>
<th>event</th>
<th>pic0</th>
<th>pic1</th>
<th>pic2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.008</td>
<td>1</td>
<td>tick</td>
<td>62984</td>
<td>13114</td>
<td>1038</td>
</tr>
<tr>
<td>2.008</td>
<td>2</td>
<td>tick</td>
<td>13535400</td>
<td>19020927</td>
<td>3687046</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10.008</td>
<td>1</td>
<td>tick</td>
<td>96818</td>
<td>19440</td>
<td>1233</td>
</tr>
<tr>
<td>10.008</td>
<td>2</td>
<td>tick</td>
<td>13286825</td>
<td>19064293</td>
<td>3696902</td>
</tr>
</tbody>
</table>

Following are sample results of the invocation with two threads without first-touch optimization. The results show that even though two threads are used, only one of these threads had allocated the memory, so it is local to a single chip resulting in a memory access pattern similar to the single thread example above:

<table>
<thead>
<tr>
<th>time</th>
<th>cpu</th>
<th>event</th>
<th>pic0</th>
<th>pic1</th>
<th>pic2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.003</td>
<td>1</td>
<td>tick</td>
<td>20081817</td>
<td>7807881</td>
<td>2320821</td>
</tr>
<tr>
<td>2.003</td>
<td>2</td>
<td>tick</td>
<td>49905</td>
<td>5499</td>
<td>380</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10.003</td>
<td>1</td>
<td>tick</td>
<td>100035827</td>
<td>2957592</td>
<td>1807225</td>
</tr>
<tr>
<td>10.003</td>
<td>2</td>
<td>tick</td>
<td>75211</td>
<td>120158</td>
<td>7217</td>
</tr>
<tr>
<td>12.003</td>
<td>1</td>
<td>tick</td>
<td>51727</td>
<td>4009</td>
<td>299</td>
</tr>
<tr>
<td>12.003</td>
<td>2</td>
<td>tick</td>
<td>50065</td>
<td>3814</td>
<td>371</td>
</tr>
</tbody>
</table>

Finally, the sample results of the invocations with two threads with first-touch optimization show that memory access is balanced over both controllers as each thread has initialized it’s own working set:

<table>
<thead>
<tr>
<th>time</th>
<th>cpu</th>
<th>event</th>
<th>pic0</th>
<th>pic1</th>
<th>pic2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.009</td>
<td>1</td>
<td>tick</td>
<td>53061003</td>
<td>13418449</td>
<td>10235667</td>
</tr>
<tr>
<td>2.009</td>
<td>2</td>
<td>tick</td>
<td>53202416</td>
<td>13240606</td>
<td>10168401</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td></td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>10.009</td>
<td>1</td>
<td>tick</td>
<td>46410162</td>
<td>980230</td>
<td>189663</td>
</tr>
<tr>
<td>10.009</td>
<td>2</td>
<td>tick</td>
<td>46396368</td>
<td>974314</td>
<td>196322</td>
</tr>
<tr>
<td>12.009</td>
<td>1</td>
<td>tick</td>
<td>51087</td>
<td>3391</td>
<td>214</td>
</tr>
<tr>
<td>12.009</td>
<td>2</td>
<td>tick</td>
<td>49682</td>
<td>3167</td>
<td>399</td>
</tr>
</tbody>
</table>

When the memory is initialized from one core and accessed from another performance degrades, since access is through the hyperlink, which is slower than direct access by the on-chip memory controller. The first-touch capability enables each of the threads to access its data directly.
For a large memory footprint the performance is about 800 MFLOPS for a single thread, about 1050 MFLOPS for two threads without first-touch optimization, and close to 1600 MFLOPS — which represents perfect scaling — for two threads with first-touch optimization.

Analyzing I/O Performance Problems in a Virtualized Environment

A common performance problem is system sluggishness due to a high rate of I/O system calls. To identify the cause it is necessary to determine which I/O system calls are called and in what frequency, by which process, and to determine why. The first step in the analysis is to narrow down the number of suspected processes, based on the ratio of the time each process runs in system context versus user context since processes that spend a high proportion of their running time in system context, can be assumed to be requesting a lot of I/O.

Good tools to initiate the analysis of this type of problem are the `vmstat` and `prstat` commands, which examine all active processes on a system and report statistics based on the selected output mode and sort order for specific processes, users, zones, and processor-sets.

In the example described here and in further detail in the Appendix (“Analyzing I/O Performance Problems in a Virtualized Environment — a Complete Description” on page 45), a Windows 2008 server is virtualized on the OpenSolaris™ OS using the Sun™ xVM hypervisor for x86 and runs fine. When the system is activated as an Active Directory domain controller, it becomes extremely sluggish.

As a first step in diagnosing this problem, the system is examined using the `vmstat 5` command, which prints out a high-level summary of system activity every five seconds, with the following results:

| kthr r b w | memory swap free page r e mf p i o f d e s r m0 m1 m2 in sy cs us sy id |
|------------|---------------------------|-------------------------|-----------|-----------------------------------|---------------------|-------------------|-----------------|-------------------|
| 0 0 0      | 17635724 4096356 0 0     | 0 0 0 0 0 0 3 3 0 0 994 | 441 717   | 0 2 98                           |
| 0 0 0      | 17635724 4096356 0 0     | 0 0 0 0 0 0 0 0 0 0 961 | 416 713   | 0 0 100                          |
| 0 0 0      | 17631448 4095528 79 465  | 0 0 0 0 0 0 0 0 0 0 1074| 9205 1428 | 1 2 97                           |
| 0 0 0      | 17604524 4072148 407 4554| 1 1 0 6 6 0 10558 72783 | 20213 4 17 79 |
| 0 0 0      | 17595828 4062360 102 828 | 0 0 0 0 0 3 3 0 3441 44747 10520 1 14 85 |
| 0 0 0      | 17598492 4064628 2 2     | 0 0 0 0 0 1 1 0 5363 28508 8752 2 3 95 |
| 0 0 0      | 17598412 4065068 0 0     | 0 0 0 0 0 20 20 0 17077 83024 30633 5 7 88 |
| 0 0 0      | 17598108 4065136 0 0     | 0 0 0 0 0 0 0 0 8951 46456 16140 2 4 93 |

Examining these results shows that the number of system calls reported in the sy column increases rapidly as soon as the affected virtual machine is booted, and remains quite high even though the CPU is constantly more than 79% idle, as reported in the id column. While it is known from past experience that a CPU-
bound workload on this system normally generates less than 5 thousand calls per five second interval, here the number of calls is constantly more than 9 thousand from the third interval and on, ranging as high as 83 thousand. Clearly something is creating a very high system call load.

In the next step in the analysis, the processes are examined with `prstat -L -m`. The `-L` option instructs the `prstat` command to report statistics for each thread separately, and the thread ID is appended to the executable name. The `-m` option instructs the `prstat` command to report information such as the percentage of time the process has spent processing system traps, page faults, latency time, waiting for user locks, and waiting for CPU, while the virtual machine is booted:

<table>
<thead>
<tr>
<th>PID</th>
<th>USERNAME</th>
<th>USR</th>
<th>SYS</th>
<th>TRP</th>
<th>TFL</th>
<th>DFL</th>
<th>LCK</th>
<th>SLP</th>
<th>LAT</th>
<th>VCX</th>
<th>ICX</th>
<th>SCL</th>
<th>SIC</th>
<th>PROCESS/LWPID</th>
</tr>
</thead>
<tbody>
<tr>
<td>16480</td>
<td>xvm</td>
<td>6.9</td>
<td>9.8</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>27</td>
<td>54</td>
<td>1.8</td>
<td>30K</td>
<td>114</td>
<td>.2M</td>
<td>0</td>
<td>qemu-dm/3</td>
</tr>
<tr>
<td>363</td>
<td>xvm</td>
<td>0.1</td>
<td>0.2</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>100</td>
<td>0.0</td>
<td>0.0</td>
<td>1</td>
<td>2X</td>
<td>0</td>
<td>xenstored/1</td>
<td></td>
</tr>
<tr>
<td>16374</td>
<td>root</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>0.0</td>
<td>100</td>
<td>0.0</td>
<td>0.0</td>
<td>10</td>
<td>0</td>
<td>1X</td>
<td>0</td>
<td>dtrace/1</td>
<td></td>
</tr>
<tr>
<td>1644</td>
<td>xvm</td>
<td>0.1</td>
<td>0.1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>33</td>
<td>66</td>
<td>0.0</td>
<td>569</td>
<td>7</td>
<td>835</td>
<td>0</td>
<td>qemu-dm/3</td>
</tr>
<tr>
<td>2399</td>
<td>root</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>100</td>
<td>0.0</td>
<td>0.0</td>
<td>49</td>
<td>0</td>
<td>0</td>
<td>sshd/1</td>
<td></td>
</tr>
<tr>
<td>16376</td>
<td>root</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>100</td>
<td>0.0</td>
<td>0.0</td>
<td>38</td>
<td>0</td>
<td>297</td>
<td>0</td>
<td>prstat/1</td>
</tr>
<tr>
<td>11705</td>
<td>xvm</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>50</td>
<td>50</td>
<td>0.0</td>
<td>576</td>
<td>15</td>
<td>850</td>
<td>0</td>
<td>qemu-dm/4</td>
</tr>
<tr>
<td>16536</td>
<td>root</td>
<td>0.0</td>
<td>0.1</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>100</td>
<td>0.0</td>
<td>0.0</td>
<td>48</td>
<td>0</td>
<td>286</td>
<td>0</td>
<td>vncviewer/1</td>
</tr>
</tbody>
</table>

Total: 36 processes, 129 lwp's, load averages: 0.64, 0.37, 0.31

The first line of the output indicates that the `qemu-dm` process executes a very large number of system calls — 200,000 — as shown in the `SCL` column. A factor of 100 more than `xenstored`, the process with the second largest number of system calls. At this stage, while it is known that the `qemu-dm` process is at fault, to solve the problem it is necessary to identify which system calls are called. This is achieved with the help of a D-script, `count_syscalls.d`, which prints system call rates for the top-ten processes/system call combinations every five seconds (for the listing and results of invocation see the Appendix).

When `count_syscalls.d` is run, of the top 10 process/system call pairs that are printed, seven are executed by `qemu-dm`, with the number of executions of `lseek` and `write` the most prevalent by far.

The next stage of the investigation is to identify why `qemu-dm` calls these I/O system calls. To answer this, an analysis of the I/O is implemented with another D-script — `qemu-stat.d`, which collects the statistics of the I/O calls performed by `qemu-dm`, while focusing on `lseek` and `write` (see listing and results of invocation in the Appendix).

The results of this invocation show that all calls to `lseek` with an absolute position and most of the calls to `write` target a single file. The vast majority of the calls to `lseek` move the file pointer of this file to an absolute position that is one byte away...
from the previous position, and the vast majority of the calls to the `write` system call write a single byte. In other words, `qemu-dm` writes a data stream as single bytes, without any buffering — an access pattern that is inherently slow.

Next, the `pfiles` command is used to identify the file accessed by `qemu-dm` through file descriptor 5 as the virtual Windows system disk:

```
# pfiles 16480
...
5: S_IFREG mode:0600 dev:182,65543 ino:26 uid:60 gid:0
size:11623923712
  O_RDWR|O_LARGEFILE
 /xvm/hermia/disk_c/vdisk.vmdk
...
```

To further analyze this problem, it must be seen where the calls to `lseek` originate by viewing the call stack. This is performed with the `qemu-callback` script, which prints the three most common call stacks for the `lseek` and `write` system calls every five seconds (see listing and invocation in the Appendix).

By examining the `write` stack trace, it seems that the virtual machine is flushing the disk cache very often, apparently for every byte. This could be the result of a disabled disk cache. Further investigation uncovers the fact that if a Microsoft Windows server acts as an Active Directory domain controller, the Active Directory directory service performs unbuffered writes and tries to disable the disk write cache on volumes hosting the Active Directory database and log files. Active Directory also works in this manner when it runs in a virtual hosting environment. Clearly this issue can only be solved by modifying the behavior of the Microsoft Windows 2008 virtual server.

**Using DTrace to Analyze Thread Scheduling with OpenMP**

To achieve the best performance on a multithreaded application the workload needs to be well balanced for all threads. When tuning a multithreaded application, the analysis of the scheduling of the threads on the different processing elements — whether CPUs, cores, or hardware threads — can provide significant insight into the performance characteristics of the application.

Note: While this example focuses on the use of DTrace, many other issues important to profiling applications that rely on OpenMP for parallelization are addressed by the Sun Studio Performance Analyzer. For details see the Sun Studio Technical Articles Web site linked from “References” on page 38.

The following program — `partest.c` — is includes a matrix initialization loop, followed by a loop that multiplies two matrices. The program is compiled with the `-xautopar` compiler and linker option, which instructs the compiler to generate
code that uses the OpenMP API to enable the program to run in several threads in parallel on different cores, accelerating it on multicore hardware:

```c
int main(int argc, char *argv[]) {
    long i, j;
    for (i = 0; i < ITER; i++)
        a[i] = b[i] = c[i] = i;
    puts("LOOP2");
    for (j = 0; j < REPEAT; j++)
        for (i = 0; i < ITER; i++)
            c[i] += a[i] * b[i];
}
```

This code lacks calls to library functions or system calls and is thus fully CPU bound. As a result, it is only affected by CPU scheduling considerations.

The `threadsched.d` DTrace script that uses the schedule provider `sched` is implemented to analyze the execution of `partest`. This script can be used to analyze the scheduling behavior of any multithreaded program.

`threadsched.d` uses the following variables:

<table>
<thead>
<tr>
<th>Variable</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>baseline</td>
<td>Used to save the initial time-stamp when the script starts</td>
</tr>
<tr>
<td>curcpu-&gt;cpu_id</td>
<td>Current CPU ID</td>
</tr>
<tr>
<td>curcpu-&gt;cpu_lgrp</td>
<td>Current CPU locality group</td>
</tr>
<tr>
<td>self-&gt;delta</td>
<td>Thread local variable that measures the time elapsed between two events</td>
</tr>
<tr>
<td>pid</td>
<td>Process ID</td>
</tr>
<tr>
<td>scale</td>
<td>Scaling factor of time-stamps from nanoseconds to milliseconds</td>
</tr>
<tr>
<td>self-&gt;lastcpu</td>
<td>Thread local ID of previous CPU the thread ran on</td>
</tr>
<tr>
<td>self-&gt;lastlgrp</td>
<td>Thread local ID of previous locality group the thread ran on</td>
</tr>
<tr>
<td>self-&gt;stamp</td>
<td>Thread local time-stamp which is also used as an initialization flag</td>
</tr>
<tr>
<td>tid</td>
<td>Thread ID</td>
</tr>
</tbody>
</table>

1. #!/usr/sbin/dtrace -s
2. #pragma D option quiet
3. BEGIN
4. {
5.    baseline = walltimestamp;
6.    scale = 1000000;
7. }

- The BEGIN probe fires when the script starts and is used to initialize the baseline timestamp employed to measure all other times from the `walltimestamp` built-in DTrace variable. `walltimestamp` and all other DTrace timestamps are measured in nanoseconds, which is a much higher resolution than needed, so the measurement is scaled down by a factor of 1 million to milliseconds. The scaling factor is saved in the variable `scale`.

10. If the program is executed on a single core system, it is likely to run slower, since it runs the code for multithreaded execution without benefiting from the performance improvement of running on multiple cores.
The `sched::on-cpu` probe fires whenever a thread is scheduled for running.

The first part of the predicate on line 9 — `pid == $target` — is common to all of the probes in this example and ensures that the probe only fires for processes that are controlled by this script.

`DTrace` initializes variables to zero, so the fact that the `self->stamp` thread-local variable is still zero indicates that the thread is running for the first time and `self->stamp` and the other thread-local variables are initialized in lines 11-14.

This code runs when a thread switches from one CPU to another.

The timestamp, the time the thread spent on the previous CPU, the last CPU this thread ran on, and the CPU it is switching to are printed in lines 25-28.

This code runs when the thread is rescheduled to run on the same CPU it ran on the previous time it was scheduled to run.
• The timestamp is updated in line 37 and a message that the thread is rescheduled to run on the same CPU is printed together with the timestamp, in lines 39-41.

```c
43 sched::off-cpu
44 / pid == $target && self->stamp /
45 {
46    self->delta = (walltimestamp - self->stamp) / scale;
47    self->stamp = walltimestamp;
48    self->stamp = (walltimestamp - baseline) / scale;
49    printf("%d:%d TID %d CPU %d(%d) preempted\n",
50       self->stamp, self->delta, tid, curcpu->cpu_id,
51       curcpu->cpu_lgrp);
52 }
```

• The sched::off-cpu probe fires whenever a thread is about to be stopped by the scheduler.

```c
53 sched::sleep
54 / pid == $target /
55 {
56    self->obj = (curlwpsinfo->pr_stype == SOBJ_MUTEX ?
57       "kernel mutex": curlwpsinfo->pr_stype == SOBJ_RWLOCK ?
58       "kernel RW lock": curlwpsinfo->pr_stype == SOBJ_Mutex ?
59       "kernel semaphore": curlwpsinfo->pr_stype == SOBJ_USER ?
60       "user-level lock": curlwpsinfo->pr_stype == SOBJ_USER_PI ?
61       "user-level PI lock": curlwpsinfo->pr_stype ==
62           SOBJ_SHUTDOWN ? "shut off": "unknown");
63    self->delta = (walltimestamp - self->stamp) / scale;
64    self->stamp = walltimestamp;
65    self->stamp = (walltimestamp - baseline) / scale;
66    printf("%d:%d TID %d sleeping on '%s'\n",
67        self->stamp, self->delta, tid, self->obj);
68 }
```

• The sched::sleep probe fires before the thread sleeps on a synchronization object.

• The type of synchronization object is printed in lines 67-68.

```c
70 sched::sleep
71 / pid == $target && ( curlwpsinfo->pr_stype == SOBJ.CV ||
72       curlwpsinfo->pr_stype == SOBJ_USER ||
73       curlwpsinfo->pr_stype == SOBJ_USER_PI ) /
74 {
75    ustack();
76 }
```

• The second sched::sleep probe fires when a thread is put to sleep on a condition variable or user-level lock, which are typically caused by the application itself, and prints the call-stack.
The `pars`et command is used to set up a processor set with two CPUs (0, 4) to simulate CPU over-commitment:

```
host# pars -c 0 4
```

Note: The numbering of the cores for the `pars`et command is system dependent.

The number of threads is set to three with the `OMP_NUM_THREADS` environment variable and `threadsched.d` is executed with `partest`:

```
host# OMP_NUM_THREADS=3 ./threadsched.d -c ./partest
```

The output first shows the startup of the main thread (lines 1 to 5). The second thread first runs at line 6 and the third at line 12:

```
1 0 : 0  TID 1 CPU 0(0) created
2 0 : 0  TID 1 CPU 0(0) restarted on same CPU
3 0 : 0  TID 1 CPU 0(0) preempted
4 0 : 0  TID 1 CPU 0(0) restarted on same CPU
5 0 : 0  TID 1 CPU 0(0) preempted
6 49 : 0  TID 2 CPU 0(0) created
7 49 : 0  TID 2 CPU 0(0) restarted on same CPU
8 49 : 0  TID 2 CPU 0(0) preempted
9 49 : 0  TID 2 CPU 0(0) restarted on same CPU
10 49 : 0  TID 2 sleeping on ‘user-level lock’
11 49 : 0  TID 2 CPU 0(0) preempted
12 49 : 0  TID 3 CPU 0(0) created
13 49 : 0  TID 3 CPU 0(0) restarted on same CPU
14 420 : 370 TID 3 CPU 0(0) preempted
15 ...
```

As the number of available CPUs is set to two, only two of the three threads can run simultaneously resulting in many thread migrations between CPUs, as seen in lines 19, 21, 36, 39, 43, and 46. At lines 24, 33, and 53 respectively, each of the three threads goes to sleep on a condition variable:

Note: OpenMP generates code to synchronize threads which cause them to sleep. This is not related to threads migrating between CPUs resulting from a shortage of available cores, as demonstrated here.

```
16  LOOP2
17  176024 : 1000 TID 2 CPU 0(0) preempted
18  176024 : 0  TID 2 CPU 0(0) restarted on same CPU
19  176004 : 0  TID 3 from-CPU 4(0) to-CPU 0(0) CPU migration
20  176004 : 0  TID 3 CPU 0(0) restarted on same CPU
21  176004 : 0  TID 1 from-CPU 4(0) to-CPU 0(0) CPU migration
22  176004 : 0  TID 1 CPU 0(0) restarted on same CPU
23  176024 : 0  TID 3 CPU 4(0) restarted on same CPU
```

11. For compatibility with legacy programs, setting the `PARALLEL` environment variable has the same effect as setting `OMP_NUM_THREADS`. However, if they are both set to different values, the runtime library issues an error message.
From line 54, the call stack dump shows that the last function called is `thrp_join`, which indicates the end of a parallelized section of the program with all threads concluding their processing and only the main thread of the process remaining:

```
54       libc.so.1`__lwp_wait+0x4
55       libc.so.1`__thrp_join+0x38
56       libmtsk.so.1`threads_fini+0x178
57       libmtsk.so.1`libmtsk_fini+0x1c
58       libmtsk.so.1`call_array+0xa0
59       libmtsk.so.1`call_fini+0xb0
60       libmtsk.so.1`atexit_fini+0x80
61       libc.so.1`_exithandle+0x44
62       libc.so.1`exit+0x4
63       partest`__start+0x184
```
Using DTrace with MPI

The Message Passing Interface (MPI) *de facto* standard is a specification for an API that allows many computers to communicate with one another. Sun HPC ClusterTools™ software is based directly on the Open MPI open-source implementation of the MPI standard, and reaps the benefits of that community initiative, which has deployments and demonstrated scalability on very large scale systems. Sun HPC ClusterTools software is the default MPI distribution in the Sun HPC Software. It is fully tested and supported by Sun on the wide spectrum of Sun x86 and SPARC processor-based systems. Sun HPC ClusterTools provides the libraries and run-time environment needed for creating and running MPI applications and includes DTrace providers, enabling additional profiling capabilities.

Note: While this example focuses on the use of DTrace with MPI, alternative, complementary, and supplementary ways to profile MPI based applications are provided by the Sun Studio Performance Analyzer. The Analyzer directly supports the profiling of MPI, and provides features that help understand message transmission issues and MPI stalls. For details see the Sun Studio Technical Articles Web site linked from “References” on page 38.

The MPI standard states that MPI profiling should be implemented with wrapper functions. The wrapper function performs the required profiling tasks, and the real MPI function is called inside the wrapper through a profiling MPI (PMPI) interface. However, using DTrace for profiling has a number of advantages over the standard approach:

- The PMPI interface is compiled code that requires restarting a job every time a library is changed. DTrace is dynamic and profiling code can be implemented in D and attached to a running process.
- MPI profiling changes the target system, resulting in differences between the behavior of profiling and non-profiling code. DTrace enables the testing of an actual production system with, at worst, a negligible affect on the system under test.
- DTrace allows the user to define probes that capture MPI tracing information with a very powerful and concise syntax.
- The profiling interface itself is implemented in D, which includes built-in mechanisms that allow for safe, dynamic, and flexible tracing on production systems.
- When DTrace is used in conjunction with MPI, DTrace provides an easy way to identify the potentially problematic function and the desired job, process, and rank that require further scrutiny.
- D has several built-in functions that help in analyzing and trouble shooting problematic programs.
Setting the Correct `mpirun` Privileges

The `mpirun` command controls several aspects of program execution and uses the Open Run-Time Environment (ORTE) to launch jobs. `dtrace_proc` and `dtrace_user` privileges are needed to run a script with the `mpirun` command, otherwise DTrace fails and reports an insufficient privileges error. To determine whether the correct privileges are assigned, the following shell script, `mppriv.sh`, must be executed by `mpirun`:

```bash
#!/bin/sh
# mppriv.sh - run ppriv(1) under a shell to see the # privileges of the process that mpirun creates
ppriv $$
```

Execute the following command to determine whether the correct privileges are assigned for a cluster consisting of `host1` and `host2`:

```
% mpirun --np 2 --host host1,host2 mppriv.sh
```

The privileges can be set by the system administrator. Once they are set, `dtrace_user` and `dtrace_proc` privileges are reported by `mppriv.sh` as follows:

```
4085: -csh
  flags = <none>
  E:basic,dtrace_proc,dtrace_user
  I:basic,dtrace_proc,dtrace_user
  P:basic,dtrace_proc,dtrace_user
  L: all
```

A Simple Script for MPI Tracing

The following script, `mpitrace.d`, is used to trace the entry into and exit from all of the MPI API calls:

```d
pid$target:libmpi:MPI_*:entry
{
  printf(“Entered %s...”, probefunc);
}

pid$target:libmpi:MPI_*:return
{
  printf(“exiting, return value = %d\n”, arg1);
}
```

To trace the entry and exit of MPI calls in four instances of an MPI application, `mpiapp`, running under dtrace and using the `mpitrace.d` script, it seems you should execute the following command:

```
% mpirun --np 4 dtrace -o mpiapp.trace -s mptrace.d -c mpiapp
```
Where:

- `np 4` instructs the `mpirun` command to invoke four copies of `dtrace`
- `-s mpitrace.d` instructs `dtrace` to run the `mpitrace.d` D-script
- `-c mpiapp` instructs each instance of the `dtrace` command to run `mpiapp`

In this invocation, the output for all processes is directed to the standard-output and the different output from different processes is mixed up in the single output file `mpiapp.trace`. To direct the output of each invocation of the `dtrace` command to a separate file, invoke it using the following `partrace.sh` script:

```bash
#!/bin/sh
dtrace -s $1 -c $2 -o $2.$OMPI_COMM_WORLD_RANK.trace
```

Where:

- `-s $1` instructs `dtrace` to run the D-script passed to it as the first argument
- `-c $2` instructs `dtrace` to run the application named in the second argument
- `-o $2.$OMPI_COMM_WORLD_RANK.trace` directs the trace-output to the file named from the concatenation of the second argument, the MPI-rank of the shell and the string `trace`. If this file exists, the output is appended to it.

Now invoke four copies of the `partrace.sh` script that invokes `mpiapp`:

```bash
% mpirun -np 4 partrace.sh mpitrace.d mpiapp
```

Note: The status of the `OMPI_COMM_WORLD_RANK` shell variable is unstable and subject to change. The user may need to change it to its replacement.

To attach the `dtrace` command with the same `mpitrace.d` D-script to a running instance of the MPI program `mpiapp` with process id 24768 run the following command:

```bash
% dtrace -p 24768 -s mpitrace.d
```

When DTrace running the `mpitrace.d` script is attached to a job that performs `send` and `recv` operations, the output looks similar to the following:

```bash
% dtrace -q -p 24770 -s mpitrace.d
Entered MPI_Send...exiting, return value = 0
Entered MPI_Recv...exiting, return value = 0
Entered MPI_Send...exiting, return value = 0
Entered MPI_Recv...exiting, return value = 0
Entered MPI_Send...exiting, return value = 0
```
The mpitrace.d script can be easily modified to include an argument list. The resulting output resembles output of the truss command:

```c
1  pid$target:libmpi:MPI_Send:entry,
2  pid$target:libmpi:MPI_*send:entry,
3  pid$target:libmpi:MPI_Recv:entry,
4  pid$target:libmpi:MPI_*recv:entry
5 { 
6   printf("%s(0x%x, %d, 0x%x, %d, %d, 0x%x)", probedest, 
7      arg0, arg1, arg2, arg3, arg4, arg5);
8 } 
9  pid$target:libmpi:MPI_Send:return,
10 pid$target:libmpi:MPI_*send:return,
11 pid$target:libmpi:MPI_Recv:return,
12 pid$target:libmpi:MPI_*recv:return
13 { 
14   printf("\ttt = %d\n", arg1); 
15 }
```

The mpitruss.d script demonstrates the use of wildcard names to match all send and receive type function calls in the MPI library. The first probe shows the usage of the built-in &%V variables to print out the argument list of the traced function. The following example shows a sample invocation of the mpitruss.d script:

```
% dtrace -q -p 24770 -s mpitruss.d
MPI_Send(0x80470b0, 1, 0x8060f48, 0, 1, 0x8060d48) = 0
MPI_Recv(0x80470a8, 1, 0x8060f48, 0, 0, 0x8060d48) = 0
MPI_Send(0x80470b0, 1, 0x8060f48, 0, 1, 0x8060d48) = 0
MPI_Recv(0x80470a8, 1, 0x8060f48, 0, 0, 0x8060d48) = 0
```

**Tracking Down MPI Memory Leaks**

MPI communicators are allocated by the MPI middleware at the request of application code, making it difficult to identify memory leaks. The following D-script — mpicommcheck.d — demonstrates how DTrace can help overcome this challenge.

mpicommcheck.d probes for MPI functions that allocate and deallocate communicators, and keeps track of the call-stack for each function. Every 10 seconds, the script dumps out the count of MPI communicator calls and the total number of instances of communicator that were allocated and freed. At the end of the DTrace session, the script prints the total count and the different stack traces, as well as the number of times those stack traces were seen.

```
1 BEGIN
2 { 
3   allocs = 0; deallocs = 0; seconds = 0;
4 } 
```

- The BEGIN probe fires when the script starts and initializes the count of allocations, deallocations, and seconds counter.

12. In MPI, a communicator is an opaque object with a number of attributes together with simple rules that govern its creation, use, and destruction. Communicators are the basic data types of MPI.
This code section implements the communicator constructor probes —

\( \text{MPI\_Comm\_create}, \text{MPI\_Comm\_dup}, \text{or MPI\_Comm\_split} \).

Line 9 increments the allocation counter, line 10 saves the count of calls to the specific function, and line 11 saves the count of the stack frames.

This code section implements the communicator destructor probe —

\( \text{MPI\_Comm\_free} \).

Line 15 increments the deallocation counter, line 16 saves the count of calls to \( \text{MPI\_Comm\_free} \), and line 17 saves the count of the stack frames.

The code section runs every 10 seconds and prints the counters.

The END probe fires when the DTrace session exits.

The count of the number of times each call stack was seen is printed in line 29.

DTrace prints aggregations on exit by default, so the three communicator constructors and the communicator destructor call stacks are printed.

When this script is run, a growing difference between the total number of allocations and deallocations can indicate the existence of a memory leak. Using the stack traces dumped, it is possible to determine where in the application code the objects
are allocated and deallocated to help diagnose the issue. The following example demonstrates such a diagnosis.

The mpicommleak MPI application performs three MPI_Comm_dup operations and two MPI_Comm_free operations. The program thus allocates one communicator more than it frees with each iteration of a specific loop. Note that this in itself does not necessarily mean that the extra allocation is in fact a memory leak, since the extra object might be referred to by code that is external to the loop in question.

When DTrace is attached to mpicommleak using mpicommcheck.d, the count of allocations and deallocations is printed every 10 seconds. In this example, this output shows that the count of the allocated communicators is growing faster than the count of deallocations.

When the DTrace session exits, the END code outputs a total of five stack traces — three for the constructor and two for the destructor call stacks, as well as the number of times each call stack was encountered.

The following command attaches the mpicommcheck.d D-script to process 24952:

```
% dtrace -q -p 24952 -s mpicommcheck.d
```

In the following output, the difference between the number of communicators allocated and freed grows continuously, indicating a potential memory leak:

```
==============================================
MPI_Comm_free          4
MPI_Comm_dup           6
Communicator Allocations = 6
Communicator Deallocations = 4
==============================================
MPI_Comm_free          8
MPI_Comm_dup           12
Communicator Allocations = 12
Communicator Deallocations = 8
==============================================
MPI_Comm_free          12
MPI_Comm_dup           18
Communicator Allocations = 18
Communicator Deallocations = 12
```
After the D-script is halted (by the user interrupting DTrace), the final count of allocations and deallocations, together with the table of stack traces, is printed:

<table>
<thead>
<tr>
<th>Communicator Allocations = 21, Communicator Deallocations = 14</th>
</tr>
</thead>
<tbody>
<tr>
<td>libmpi.so.0.0.0’MPI_Comm_free</td>
</tr>
<tr>
<td>mpicommleak’deallocate_comms+0x19</td>
</tr>
<tr>
<td>mpicommleak’main+0x6d</td>
</tr>
<tr>
<td>mpicommleak’0x805081a</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>libmpi.so.0.0.0’MPI_Comm_free</td>
</tr>
<tr>
<td>mpicommleak’deallocate_comms+0x26</td>
</tr>
<tr>
<td>mpicommleak’main+0x6d</td>
</tr>
<tr>
<td>mpicommleak’0x805081a</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>libmpi.so.0.0.0’MPI_Comm_dup</td>
</tr>
<tr>
<td>mpicommleak’allocate_comms+0x1e</td>
</tr>
<tr>
<td>mpicommleak’main+0x5b</td>
</tr>
<tr>
<td>mpicommleak’0x805081a</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>libmpi.so.0.0.0’MPI_Comm_dup</td>
</tr>
<tr>
<td>mpicommleak’allocate_comms+0x30</td>
</tr>
<tr>
<td>mpicommleak’main+0x5b</td>
</tr>
<tr>
<td>mpicommleak’0x805081a</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>libmpi.so.0.0.0’MPI_Comm_dup</td>
</tr>
<tr>
<td>mpicommleak’allocate_comms+0x42</td>
</tr>
<tr>
<td>mpicommleak’main+0x5b</td>
</tr>
<tr>
<td>mpicommleak’0x805081a</td>
</tr>
<tr>
<td>7</td>
</tr>
</tbody>
</table>

In analyzing the output of the script, it is clear that 50% more communicators are allocated than are deallocated, indicating a potential memory leak. The five stack traces, each occurring seven times, point to the code that called the communicator constructors and destructors. By analyzing the handling of the communicator objects by the code executed following the calls to their constructors, it is possible to identify whether there is in fact a memory leak, and its cause.

**Using the DTrace mpiperuse Provider**

PERUSE is an extension interface to MPI that exposes performance related processes and interactions between application software, system software, and MPI message-passing middleware. PERUSE provides a more detailed view of MPI performance.
compared to the standard MPI profiling interface (PMPI). Sun HPC ClusterTools includes a DTrace provider named \texttt{mpiperuse} that supports DTrace probes into the MPI library.

The DTrace \texttt{mpiperuse} provider exposes the events defined in the MPI PERUSE specification that track the MPI requests. To use an \texttt{mpiperuse} probe, add the string \texttt{mpiperuse$target:::} before the probe-name, remove the PERUSE\_ prefix, convert the letters into lower-case, and replace the underscore characters with dashes. For example, to specify a probe to capture a \texttt{PERUSE\_COMM\_REQ\_ACTIVATE} event, use \texttt{mpiperuse$target:::comm-req-activate}. If the optional object and function fields in the probe description are omitted, all of the \texttt{comm-req-activate} probes in the MPI library and all of its plug-ins will fire.

All of the \texttt{mpiperuse} probes recognize the following four arguments:

<table>
<thead>
<tr>
<th>Argument</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{mpicomminfo_t *i}</td>
<td>Provides a basic source and destination for the request and which protocol is expected to be used for the transfer.</td>
</tr>
<tr>
<td>\texttt{uintptr_t uid}</td>
<td>The PERUSE unique ID for the request that fired the probe (as defined by the PERUSE specifications). For OMPI this is the address of the actual request.</td>
</tr>
<tr>
<td>\texttt{uint_t op}</td>
<td>Indicates whether the probe is for a send or receive request.</td>
</tr>
<tr>
<td>\texttt{mpicomm_spec_t *cs}</td>
<td>Mimics the \texttt{spec} structure, as defined in the PERUSE specification.</td>
</tr>
</tbody>
</table>

To run D-scripts with \texttt{mpiperuse} probes, use the \texttt{-Z} (upper case) switch with the \texttt{dtrace} command since the \texttt{mpiperuse} probes do not exist at initial load time:

```bash
#!/bin/sh
# partrace.sh - a helper script to dtrace MPI jobs from
# the start of the job.
dtrace -Z -s $1 -c $2 -o $2.$OMPI_COMM_WORLD_RANK.trace
```

### Tracking MPI Message Queues with DTrace

Sun HPC ClusterTools™ software includes an example of using \texttt{mpiperuse} probes on the Solaris OS that allows users to see the activities of the MPI message queues. To run the script, \texttt{mpistat.d}, a compiled MPI program that is executed with the \texttt{mpirun} command is needed to create two MPI processes. In this example, the process ID of the traced process is 13371.

Run the \texttt{mpistat.d} D-script:

```
% dtrace -q -p 13371 -s mpistat.d
```
The `mpistat.d` D-script prints out the following columns of information on the MPI message queues every second:

<table>
<thead>
<tr>
<th>Col</th>
<th>PERUSE event probe</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><code>REQ_XFER_CONTINUE</code></td>
<td>Count of bytes received</td>
</tr>
<tr>
<td>2</td>
<td><code>REQ_ACTIVATE</code></td>
<td>Number of message transfers activated</td>
</tr>
<tr>
<td>3</td>
<td><code>REQ_INSERT_IN_POSTED_Q</code></td>
<td>Number of messages inserted in the posted request queue</td>
</tr>
<tr>
<td>4</td>
<td><code>MSG_INSERT_IN_UNEX_Q</code></td>
<td>Number of messages inserted into the unexpected queue</td>
</tr>
<tr>
<td>5</td>
<td><code>MSG_MATCH_POSTED_REQ</code></td>
<td>Number of incoming messages matched to a posted request</td>
</tr>
<tr>
<td>6</td>
<td><code>REQ_MATCH_UNEX</code></td>
<td>Number of user requests matched to an unexpected message</td>
</tr>
<tr>
<td>7</td>
<td><code>REQ_XFER_END</code></td>
<td>Count of bytes that have been scheduled for transfer</td>
</tr>
<tr>
<td>8</td>
<td><code>REQ_ACTIVATE</code></td>
<td>Number of times MPI has initiated the transfer of a message</td>
</tr>
</tbody>
</table>

**A Few Simple `mpiperuse` Usage Examples**

The examples in this section show how to perform the described DTrace operations from the command line. In each example, the process ID shown (254) must be substituted with the process ID of the process to be monitored.

Count the number of messages to or from a host:

```bash
% dtrace -p 254 -n 'mpiperuse$target:::comm-req-xfer-end { \\ @[args[0]->ci_remote] = count();}'
dtrace: description 'mpiperuse$target:::comm-req-xfer-end' matched 17 probes
^C
count
host2 recv 3
host2 send 3
```

Count the number of messages to or from specific MPI byte transfer layers (BTLs):

```bash
% dtrace -p 254 -n 'mpiperuse$target:::comm-req-xfer-end { \\ @[args[0]->ci_protocol] = count();}'
dtrace: description 'mpiperuse$target:::comm-req-xfer-end' matched 17 probes
^C
sm 60
```
Obtain distribution plots of message sizes sent or received from a host:

```
% dtrace -p 254 -n 'mpiperuse$target::comm-req-xfer-end {'
  @[args[0]->ci_remote] = quantize(args[3]->mcs_count);}'
dtrace: description 'mpiperuse$target::comm-req-xfer-end'
mached 17 probes
`C
myhost
value ------------ Distribution ------------ count
 2 | 0
 4 |................................................................. 4
 8 | 0

Create distribution plots of message sizes by communicator, rank, and send/receive:

```
% dtrace -p 254 -n 'mpiperuse$target::comm-req-xfer-end {'
  @[args[3]->mcs_comm, args[3]->mcs_peer, args[3]->mcs_op]  
  = quantize(args[3]->mcs_count);}'
dtrace: description 'mpiperuse$target::comm-req-xfer-end'
mached 19 probes
`C
134614864 1 recv
value ------------ Distribution ------------ count
 2 | 0
 4 |................................................................. 9
 8 | 0
134614864 1 send
value ------------ Distribution ------------ count
 2 | 0
 4 |................................................................. 9
 8 | 0
```
Chapter 4

Conclusion

The increased complexity of modern computing and storage architectures with multiple CPUs, cores, threads, virtualized operating systems, networking, and storage devices present serious challenges to system architects, administrators, developers, and users. The requirements of high availability and reliability, together with the increasing pressure on datacenter budgets and resources, create an environment where it is necessary to maintain systems at a high level of performance — without the ability to simply add resources to resolve performance issues.

To achieve these seemingly conflicting objectives, Sun Microsystems provides a comprehensive set of tools on the Solaris 10 OS with DTrace at their core, that enable unprecedented levels of observability and insight into the workings of the operating system and the applications running on it. These tools allow datacenter professionals to quickly analyze and diagnose issues in the systems they maintain without increasing operational risk, making the promise of observability as a driver of consistent system performance and stability a reality.

Acknowledgments

Thomas Nau of the Infrastructure Department, Ulm University, provided the content for the chapters on DTrace and its Visualization Tools and Performance Analysis Examples, except for the section on Using DTrace with MPI.

The section on Using DTrace with MPI is based on the article Using the Solaris DTrace Utility With Open MPI Applications, by Terry Dontje, Karen Norteman, and Rolf vande Vaart of Sun Microsystems.
### Table 3. Web sites for more information

<table>
<thead>
<tr>
<th>Description</th>
<th>URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>DLight tutorial</td>
<td><a href="http://blogs.sun.com/solarisdev/entry/project_d_light_tutorial">http://blogs.sun.com/solarisdev/entry/project_d_light_tutorial</a></td>
</tr>
<tr>
<td>Yet another DLight tutorial</td>
<td><a href="http://developers.sun.com/sunstudio/documentation/tutorials/dlight/">http://developers.sun.com/sunstudio/documentation/tutorials/dlight/</a></td>
</tr>
<tr>
<td>Chime Visualization Tool for DTrace</td>
<td><a href="http://opensolaris.org/os/project/dtrace-chime/">http://opensolaris.org/os/project/dtrace-chime/</a></td>
</tr>
<tr>
<td>The NetBeans DTrace GUI Plugin Tutorial (includes Chime)</td>
<td><a href="http://netbeans.org/kb/docs/ide/NetBeans_DTrace.GUI_Plugin_0.4.html">http://netbeans.org/kb/docs/ide/NetBeans_DTrace.GUI_Plugin_0.4.html</a></td>
</tr>
<tr>
<td>DTraceToolkit</td>
<td><a href="http://opensolaris.org/os/community/dtrace/dtrace">http://opensolaris.org/os/community/dtrace/dtrace</a> toolkit/</td>
</tr>
<tr>
<td>Proposed cpc DTrace provider</td>
<td><a href="http://wikis.sun.com/display/DTrace/cpc+Provider">http://wikis.sun.com/display/DTrace/cpc+Provider</a></td>
</tr>
<tr>
<td>Sun Studio documentation</td>
<td><a href="http://developers.sun.com/sunstudio/documentation/index.jsp">http://developers.sun.com/sunstudio/documentation/index.jsp</a></td>
</tr>
<tr>
<td>Sun Studio Technical Articles</td>
<td><a href="http://developers.sun.com/sunstudio/documentation/techart/">http://developers.sun.com/sunstudio/documentation/techart/</a></td>
</tr>
<tr>
<td>Sun HPC ClusterTools</td>
<td><a href="http://sun.com/clustertools">http://sun.com/clustertools</a></td>
</tr>
<tr>
<td>Sun HPC ClusterTools 8 Documentation</td>
<td><a href="http://docs.sun.com/app/docs/prod/hpc.cluster8%22hpc-clustertools8?l=en#hic">http://docs.sun.com/app/docs/prod/hpc.cluster8&quot;hpc-clustertools8?l=en#hic</a></td>
</tr>
<tr>
<td>MPO/lgroup observability tools</td>
<td><a href="http://opensolaris.org/os/community/performance/numa/observability/tools/">http://opensolaris.org/os/community/performance/numa/observability/tools/</a></td>
</tr>
<tr>
<td>John the Ripper password evaluation tool</td>
<td><a href="http://openwall.com/john/">http://openwall.com/john/</a></td>
</tr>
<tr>
<td>BIOS and Kernel Developer's Guide for the AMD Athlon 64 and AMD Opteron Processors AMD publication #26094</td>
<td><a href="http://support.amd.com/us/Processor_TechDocs/26094.PDF">http://support.amd.com/us/Processor_TechDocs/26094.PDF</a></td>
</tr>
<tr>
<td>Using the Solaris DTrace Utility With Open MPI Applications</td>
<td><a href="http://developers.sun.com/solaris/articles/dtrace_article.html">http://developers.sun.com/solaris/articles/dtrace_article.html</a></td>
</tr>
</tbody>
</table>
Chapter 6
Appendix

Chime Screenshots

Figure 8. Processes sorted by system call load

Figure 9. Plotting of reads over time with the rfilio DTraceToolkit script
Listings for Matrix Addition cputrack Example

gnuplot Program to Graph the Data Generated by cputrack

```plaintext
set terminal post eps color enh
set output "cputrack_add_reads.eps"
set xlabel "time [s]"
set ylabel "L1 D-cache misses/s"
set y2label "Instruction count/s"
set format y "%.12g"
set format y2 "%.3g"
set key right center
plot "cputrack_add_col.data" using 5 title "D-cache misses col"
with lines lt 1, "cputrack_add_row.data" using 5 title "D-cache misses row"
with lines lt 2,"cputrack_add_col.data"
using 6 title "Instruction count col" axes x1y2 with lines,"cputrack_add_row.data" using 6 title "Instruction count row"
axes x1y2 with lines
```

cputrack_add_row.data

```
1.139 20407 1 tick 543876 87150751
2.139 20407 1 tick 1080515 172856204
3.139 20407 1 tick 751448 120190633
4.189 20407 1 tick 800640 128044765
5.436 20407 1 tick 1016923 162701934
6.029 20407 1 tick 1320444 105641221
7.321 20407 1 tick 2177030 174171702
8.119 20407 1 tick 1777592 142214947
9.129 20407 1 tick 1549835 123993388
10.148 20407 1 tick 1563710 125103730
11.149 20407 1 tick 2227908 178242225
12.036 20407 1 tick 1269566 101570698
13.029 20407 1 tick 2213748 177109392
14.049 20407 1 tick 1564170 125140106
15.059 20407 1 tick 1543978 123525043
16.069 20407 1 tick 2251644 180141162
17.149 20407 1 tick 1693036 135450053
18.159 20407 1 tick 2251025 180091840
19.169 20407 1 tick 2251214 180106870
```

cputrack_add_col.data

```
1.077 20435 1 tick 1837 181610
2.205 20435 1 tick 3963 55920
3.155 20435 1 tick 5592 55920
4.111 20435 1 tick 5594 55940
5.118 20435 1 tick 5639902 56404794
6.068 20435 1 tick 5343405 53439438
7.078 20435 1 tick 5670176 56707868
8.128 20435 1 tick 5866023 58666242
9.088 20435 1 tick 5392723 53932755
```
10.038 20435 1 tick 5341397 53419619
11.048 20435 1 tick 5669263 56698666
12.088 20435 1 tick 5847069 58476774
13.048 20435 1 tick 5393087 53936563
14.058 20435 1 tick 5669046 56696528
15.098 20435 1 tick 5850128 58507291
16.058 20435 1 tick 5382361 53829433
17.068 20435 1 tick 5679960 56805633
18.108 20435 1 tick 5838186 58387889
19.068 20435 1 tick 5394299 53948583
20.078 20435 1 tick 5675952 56765368
21.088 20435 1 tick 5704645 57050473
22.098 20435 1 tick 5732652 57329627
23.108 20435 1 tick 5731854 57321618
24.118 20435 1 tick 5729861 57301764
25.128 20435 1 tick 5731798 57321087
26.138 20435 1 tick 5725666 57259767
27.148 20435 1 tick 5731746 57320554
28.028 20435 1 tick 4990441 49907147
29.198 20435 1 tick 6640872 66412320
30.208 20435 1 tick 5731060 57313707
31.218 20435 1 tick 5732886 57331967
32.028 20435 1 tick 4596166 45964143
33.038 20435 1 tick 5730656 57309707
34.048 20435 1 tick 5733241 57335525
35.058 20435 1 tick 5732103 57324146
36.068 20435 1 tick 5734208 57345187
37.078 20435 1 tick 5734788 57350987
38.088 20435 1 tick 5730253 57305615
39.098 20435 1 tick 5732309 57326189
40.108 20435 1 tick 5732599 57329105
41.118 20435 1 tick 5731675 57319849
42.128 20435 1 tick 5731438 57317487
43.138 20435 1 tick 5733813 57341246
44.148 20435 1 tick 5732757 57330658
45.158 20435 1 tick 5732348 57326601
46.168 20435 1 tick 5731334 57316430
47.178 20435 1 tick 5733060 57333714
48.188 20435 1 tick 5731966 57322764
49.198 20435 1 tick 5733201 57335121
50.208 20435 1 tick 5732679 57329906
51.218 20435 1 tick 5720798 57211057
52.028 20435 1 tick 4594868 45951163
53.038 20435 1 tick 5732078 57323887
54.048 20435 1 tick 5732950 57332608
Listings for Matrix Addition busstat Example

gnuplot Program to Graph the Data Generated by busstat

```plaintext
set terminal post eps color enh
set output “busstat_add_reads.eps”
set xlabel “time [s]”
set ylabel “reads/s”
set format y “%.10g”
set key right center
plot “busstat_add_col.
data” using 4 title “reads col” with lines,
“busstat_add_row.data” using 4 title “reads row” with lines
```

```
busstat_add_row.data
1 dram0 mem_reads 533963 mem_reads 533961
2 dram0 mem_reads 799399 mem_reads 799401
3 dram0 mem_reads 592914 mem_reads 592913
4 dram0 mem_reads 596631 mem_reads 596631
5 dram0 mem_reads 646201 mem_reads 646202
6 dram0 mem_reads 760912 mem_reads 760918
7 dram0 mem_reads 914140 mem_reads 914121
8 dram0 mem_reads 793090 mem_reads 793110
9 dram0 mem_reads 808819 mem_reads 808821
10 dram0 mem_reads 806084 mem_reads 806062
11 dram0 mem_reads 952585 mem_reads 952590
12 dram0 mem_reads 786520 mem_reads 786534
13 dram0 mem_reads 908651 mem_reads 908651
14 dram0 mem_reads 787943 mem_reads 787946
15 dram0 mem_reads 792237 mem_reads 792234
16 dram0 mem_reads 909442 mem_reads 909442
17 dram0 mem_reads 788911 mem_reads 788914
18 dram0 mem_reads 909884 mem_reads 909883
19 dram0 mem_reads 909077 mem_reads 909075
20 dram0 mem_reads 893899 mem_reads 893887
21 dram0 mem_reads 712354 mem_reads 712360
22 dram0 mem_reads 706965 mem_reads 706939

busstat_add_col.data
1 dram0 mem_reads 433670 mem_reads 433679
2 dram0 mem_reads 473979 mem_reads 473979
3 dram0 mem_reads 499469 mem_reads 499470
4 dram0 mem_reads 515084 mem_reads 515079
5 dram0 mem_reads 2187521 mem_reads 2187515
6 dram0 mem_reads 2133002 mem_reads 2133002
7 dram0 mem_reads 2338061 mem_reads 2338384
8 dram0 mem_reads 2222522 mem_reads 2222200
9 dram0 mem_reads 219717 mem_reads 2129715
10 dram0 mem_reads 2299319 mem_reads 2299648
11 dram0 mem_reads 2257854 mem_reads 2257526
```
Tuning Parallel Code on the Solaris OS — Lessons Learned from HPC

Sun Microsystems, Inc.

<table>
<thead>
<tr>
<th>dram0</th>
<th>mem_reads</th>
<th>mem_reads</th>
<th>mem_reads</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2130926</td>
<td>2130925</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2310048</td>
<td>2310379</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2251190</td>
<td>2250859</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2139761</td>
<td>2139762</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2318954</td>
<td>2319165</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2267098</td>
<td>2266887</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2133796</td>
<td>2133796</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2316716</td>
<td>2317043</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2247743</td>
<td>2247415</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2126646</td>
<td>2126646</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2422330</td>
<td>2422331</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2126140</td>
<td>2126139</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2257797</td>
<td>2258125</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2290093</td>
<td>2289766</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2285157</td>
<td>2285157</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2421488</td>
<td>2421488</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2127068</td>
<td>2127067</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2306129</td>
<td>2306451</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2241317</td>
<td>2240996</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2126109</td>
<td>2126109</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2423166</td>
<td>2423234</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2126926</td>
<td>2126861</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2372355</td>
<td>2372674</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2175058</td>
<td>2174736</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2164405</td>
<td>2164613</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2393185</td>
<td>2392976</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2127068</td>
<td>2127068</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2374448</td>
<td>2374656</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2203958</td>
<td>2203751</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2134857</td>
<td>2135187</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2418531</td>
<td>2418201</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2129696</td>
<td>2129695</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2415138</td>
<td>2415472</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2132350</td>
<td>2132016</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2208236</td>
<td>2208451</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2341949</td>
<td>2341734</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2125826</td>
<td>2125826</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2420923</td>
<td>2420924</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2147163</td>
<td>2147162</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2288892</td>
<td>2289109</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2273169</td>
<td>2272953</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2131584</td>
<td>2131583</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2421206</td>
<td>2421207</td>
<td></td>
</tr>
<tr>
<td></td>
<td>1889551</td>
<td>1889558</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9474402</td>
<td>9474415</td>
<td></td>
</tr>
<tr>
<td></td>
<td>620107</td>
<td>620083</td>
<td></td>
</tr>
</tbody>
</table>
Sun Studio Analyzer Hints to Collect Cache Misses on an UltraSPARC T1® Processor

```c
en_desc on
ignore_no_xhwcpprof
mobj_define Vaddr VADDR
mobj_define Paddr PADDR
mobj_define Process PID
mobj_define Thread (PID*1000)+THRID
mobj_define ThreadID THRID
mobj_define Seconds (TSTAMP/1000000000)
mobj_define Minutes (TSTAMP/60000000000)
prev_obj UST1_Bank (PADDR&0xc0)>>6
mobj_define UST1_L2CacheLine (PADDR&0x2ff0)>>6
mobj_define UST1_L1DataCacheLine (PADDR&0x7f0)>>4
mobj_define UST1_Strand (CPUID)
prev_obj UST1_Core (CPUID&0x1c)>>2
mobj_define VA_L2 VADDR>>6
mobj_define VA_L1 VADDR>>4
mobj_define PA_L2 PADDR>>6
mobj_define PA_L1 PADDR>>4
Vpage_256M VADDR>>28
Ppage_256M PADDR>>28
```
Analyzing I/O Performance Problems in a Virtualized Environment — a Complete Description

As described in the summary of this example in the section “Analyzing I/O Performance Problems in a Virtualized Environment” on page 19, the following is a full description of resolving system sluggishness due to a high rate of I/O system calls. To identify the cause it is necessary to determine which I/O system calls are called and in what frequency, by which process, and to determine why. The first step in the analysis is to narrow down the number of suspected processes, based on the ratio of the time each process runs in system context versus user context, with the `vmstat` and `pstat` commands.

In the example described here in detail, a Windows 2008 server is virtualized on the OpenSolaris™ OS using the Sun™ xVM hypervisor for x86 and runs fine. When the system is activated as an Active Directory domain controller, it becomes extremely sluggish.

As a first step in diagnosing this problem, the system is examined using the `vmstat` command, which prints out a high-level summary of system activity every five seconds, with the following results:

```
    kthr  memory    page  disk  faults  cpu
    r  b  w  swap  free  re  mf  pi  po  fr  de  sr  m0  m1  m2  m3  in  sy  cs  us  sy  id
0  0  0  17635724 4096356  0  0  0  0  0  0  3  3  0  0  994  441  717  0  2  98
0  0  0  17635724 4096356  0  0  0  0  0  0  0  0  0  961  416  713  0  0  100
0  0  0  17631448 4095528  79  465  0  0  0  0  0  0  0  0  1074  9205  1428  1  2  97
0  0  0  17604524 4072148  407  4554  0  1  1  0  0  6  6  0  10558  72783  202134  17  79
0  0  0  17595828 4062360 102  828  0  0  0  0  0  3  3  0  3441  44747  105201  14  85
0  0  0  17598492 4064628  2  2  0  0  0  0  0  1  1  0  5363  28508  8752  2  3  95
0  0  0  17598412 4065068  0  0  0  0  0  0  0  0  0  0  0  0  8951  46456  161402  4  93
```

The results show that the number of system calls reported in the `sy` column increases rapidly as soon as the affected virtual machine is booted, and remains quite high even though the CPU is constantly more than 79% idle, as reported in the `id` column. The number of calls is constantly more than 9 thousand from the third interval and on, ranging as high as 83 thousand. Clearly something is creating a very high system call load.

In the next step in the analysis, the processes are examined with `pstat –L –m`. The `–L` option instructs the `pstat` command to report statistics for each thread separately, and the thread ID is the appended to the executable name. The `–m` option instructs the `pstat` command to report information such as the percentage of time the process has spent processing system traps, page faults, latency time, waiting for user locks, and waiting for CPU, while the virtual machine is booted:
The first line of the output indicates that the `qemu-dm` process executes a very large number of system calls — 200,000 — as shown in the SCI column. A factor of 100 more than `xenstored`, the process with the second largest number of system calls.

At this stage, while it is known that the `qemu-dm` process is at fault, to solve the problem it is necessary to identify which system calls are called. This is achieved with the help of the following D-script, `count_syscalls.d`, which prints system call rates for the top-ten processes/system call combinations every five seconds:

```d
1 #!/usr/sbin/dtrace -s
2 #pragma D option quiet
3 BEGIN {
4   timer = timestamp; /* nanosecond timestamp */
5 }
6 syscall:::entry {
7     @call_count[pid, execname, probefunc] = count();
8 }
9 tick-5s {
10    trunc(t, 10);
11    normalize(@call_count, (timestamp-timer) / 1000000000);
12    printf("%5d %20s %6d %s
", @call_count);
13    clear(@call_count);
14    printf("\n");
15    timer = timestamp;
16 }
```

- The `BEGIN` probe fires when the script starts and is used to initialize the timer.
- The `syscall:::entry` probe fires whenever a system call is called and the system call name, the name of executable that called it, and the process ID are saved in the `call_count` aggregation.
- The `tick-5s` prints the information collected — line 10 truncates the aggregation to its top 10 entries, line 12 prints the system call count, and line 13 clears the contents of the aggregation.

When `count_syscalls.d` is run, of the top 10 process/system call pairs that are printed, seven are executed by `qemu-dm`, with the number of executions of `1seek` and `write` the most prevalent by far:
The next stage of the investigation is to identify why `qemu-dm` calls these I/O system calls. To answer this, an analysis of the I/O is implemented with another D-script — `qemu-stat.d`, which collects the statistics of the I/O calls performed by `qemu-dm`, while focusing on `lseek` and `write`:

```plaintext
1 #!/usr/sbin/dtrace -s
2 #pragma D option quiet
3 BEGIN {
4   seek = 0L;
5 }
6 syscall::lseek::entry
7 / execname == "qemu-dm" && !arg2 && seek /
8 {
9   @lseek[arg0, arg1-seek] = count();
10   seek = arg1;
11 }
12 syscall::lseek::entry
13 / execname == "qemu-dm" && !arg2 && !seek /
14 {
15   seek = arg1;
16 }
```

- The two instances of the `syscall::lseek::entry` probe have predicates (lines 7 and 13) that ensure they are called only if the triggering call to `lseek` sets the file pointer to an absolute value, by testing that `arg2` (whence) is zero, or `SEEK_SET`.
- The probes identify the file according to the file descriptor (`arg0`).
- To determine the I/O pattern, the D-script saves the last absolute position of the file pointer passed to `lseek()` in the variable `seek` in line 10. The difference between the current and previous position of the file pointer is used as the second index of the aggregation in line 9.

```plaintext
17 syscall::write::entry
18 / execname == "qemu-dm" /
19 {
20   @write[arg0, arg2] = count();
21 }
```

- The `syscall::write::entry` probe counts the number of times the `write` system call is called with a specific number of bytes written.
Lines 23 and 24 truncate the saved counts to the top five values.

Lines 25 and 26 print out the count of calls to `lseek` and lines 27 and 28 print out the count of calls to `write`.

Following are results of a single invocation of `qemu-stat-d`:

```
<table>
<thead>
<tr>
<th>lseek fdesc offset count</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 26 28</td>
</tr>
<tr>
<td>5 29 28</td>
</tr>
<tr>
<td>5 0 42</td>
</tr>
<tr>
<td>5 21 42</td>
</tr>
<tr>
<td>5 1 134540</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>write fdesc size count</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 21 42</td>
</tr>
<tr>
<td>15 4 54</td>
</tr>
<tr>
<td>16 4 63</td>
</tr>
<tr>
<td>14 4 441</td>
</tr>
<tr>
<td>5 1 134554</td>
</tr>
</tbody>
</table>
```

The results of this invocation show that all calls to `lseek` with an absolute position and most of the calls to `write` target a single file. The vast majority of the calls to `lseek` move the file pointer of this file to an absolute position that is one byte away from the previous position, and the vast majority of the calls to the `write` system call write a single byte. In other words, `qemu-dm` writes a data stream as single bytes, without any buffering — an access pattern that is inherently slow.

Next, the `pfiles` command is used to identify the file accessed by `qemu-dm` through file descriptor 5 as the virtual Windows system disk:

```
# pfiles 16480
... 5: S_IFREG mode:0600 dev:182,65543 ino:26 uid:60 gid:0 size:11623923712
    O_RDWR|O_LARGEFILE
    /xvm/hermia/disk_c/vdisk.vmdk
...```
To drill down further into this problem, it is necessary to investigate where the calls to `lseek` originate by viewing the call stack. This is performed with the `gemu-callstack.d` script, which prints the three most common call stacks for the `lseek` and `write` system calls every five seconds:

```bash
1  #!/usr/sbin/dtrace -s
2  #pragma D option quiet
3  syscall::lseek::entry, syscall::write::entry
4  / execname == "gemu-dm" /
5  {
6    @c[probefunc, ustack()] = count();
7  }
8  tick-5s {
9    trunc(@c, 3);
10   printa(@c);
11   clear(@c);
12 }
```

- Line 6 saves the call stack of `lseek` and `write`.
- Line 10 prints the three most common stacks.

Following is the most common stack trace of the calls to `write`, extracted from a single invocation of `gemu-callstack.d`:

```
write
tlibc.so.1\_write+0xa
gemu-dm\_RTFileWrite+0x37
gemu-dm\_RTFileWriteAt+0x48
gemu-dm\_vmdkWriteDescriptor+0x1d5
gemu-dm\_vmdkFlushImage+0x23
gemu-dm\_vmdkFlush+0x9
gemu-dm\_VDFlush+0x91
gemu-dm\_vdisk\_flush+0x1c
gemu-dm\_bdrv\_flush+0x2e
gemu-dm\_ide\_write\_dma\_cb+0x187
gemu-dm\_bdrv\_aio\_bh\_cb+0x16
gemu-dm\_qemu\_bh\_poll+0x2d
gemu-dm\_main\_loop\_wait+0x22c
gemu-dm\_main\_loop+0x7a
gemu-dm\_main+0x1886
gemu-dm\_start+0x6c
```

By examining the `write` stack trace, it seems that the virtual machine is flushing the disk cache very often, apparently for every byte. This could be the result of a disabled disk cache. Further investigation uncovers the fact that if a Microsoft Windows server acts as an Active Directory domain controller, the Active Directory directory service performs unbuffered writes and tries to disable the disk write cache on volumes hosting the Active Directory database and log files. Active Directory also works in this manner when it runs in a virtual hosting environment. Clearly this issue can only be solved by modifying the behavior of the Microsoft Windows 2008 virtual server.
Once the problem is resolved on the Microsoft Windows 2008 virtual server, the `gemu-stat-quant.d` D-script is implemented to print the patterns of calls to write:

```bash
1  #!/usr/sbin/dtrace -s
2  #pragma D option quiet
3  syscall::write:entry
4  / execname == "gemu-dm" && arg0 == $1 /
5  {
6    @writeq["write"] = quantize(arg2);
7  }
8  tick-5s {
9    printa(@writeq);
10   clear(@writeq);
11 }
```

Invoking it shows a much improved pattern of calls, in term of the size of each write:

```
# ./gemu-stat-quant.d 5
write value          Distribution count
256 |            0
512 | @@@@@@@@@ 14
1024 |           0
2048 | @@@@@@@@@ 10
4096 | @@@@@@@@@@@@@@@@@@@ 39
8192 | @@@@@@@@@@@@@@@@@@@@@ 36
16384 |            0
```
Tracking MPI Requests with DTrace — mpistat.d

```
BEGIN
{
  recvs_bytes=0; recvs_act=0; recvs_posted_size=0;
  recvs_unexp_size=0; recvs_posted_matches=0;
  recvs_unexp_matches=0;
  sends_act=0; sends_bytes=0; output_cnt = 0;
  printf("input(Total) Q-sizes Q-Matches\n");
  printf(" bytes active posted unexp posted unexp bytes\n");
  printf("active\n\n", recvs_bytes, recvs_act, recvs_posted_size,
          recvs_unexp_size, recvs_posted_matches,
          recvs_unexp_matches, sends_bytes, sends_act);
}
/* Print Statistics every 1 sec */
profile::tick-1sec
{
  printf("%d %d %d %d %d %d %d %d %d %d %d \n",
          recvs_bytes, recvs_act, recvs_posted_size,
          recvs_unexp_size, recvs_posted_matches,
          recvs_unexp_matches, sends_bytes, sends_act);
  ++output_cnt;
}
/* Collect Active Send Requests */
mpiperuse$target::comm-req-activate
/args[3]->mcs_op="send"
{
  ++sends_act;
}
/* Collect Removal of Send Requests */
mpiperuse$target::comm-req-notify
/args[3]->mcs_op="send"
{
  --sends_act;
}
/* Collect bytes Sent */
mpiperuse$target::comm-req-xfer-end
/args[3]->mcs_op="send"
{
  sends_bytes += args[3]->mcs_count;
}
/* Collect Active Recv Request */
mpiperuse$target::comm-req-activate
/args[3]->mcs_op="recv"
{
  ++recvs_act;
}
/* Collect Removal of Recv Request */
mpiperuse$target::comm-req-notify
/args[3]->mcs_op="recv"&recvs_act>0
{
  --recvs_act;
}
/* Collect Request Placed on Posted Q */
mpiperuse$target::comm-req-insert-in-posted-q
/args[3]->mcs_op="recv"
{
  ++recvs_posted_size;
}
```
/* Collect Msg matched Posted Q */
mpiperuse$target:::comm-msg-match-posted-req
/args[3]->mcs_op=="recv"
{
    ++recvs_posted_matches;
}
mpiperuse$target:::comm-msg-match-posted-req
/args[3]->mcs_op=="recv"&&recvs_posted_size>0
{
    --recvs_posted_size;
}
/* Collect messages in unexp Q */
mpiperuse$target:::comm-msg-insert-in-unexp-q
/args[3]->mcs_op=="recv"
{
    ++recvs_unexp_size;
}
/* Collect messages removed from unexp Q */
mpiperuse$target:::comm-msg-remove-from-unexp-q
/args[3]->mcs_op=="recv"&&recvs_unexp_size>0
{
    --recvs_unexp_size;
}
/* Collect messages removed from unexp Q */
mpiperuse$target:::comm-req-match-unex
/args[3]->mcs_op=="recv"
{
    ++recvs_unexp_matches;
}
/* Collect count of bytes recieved */
mpiperuse$target:::comm-req-xfer-continue
/Args[3]->mcs_op=="recv"
{
    recvs_bytes += args[3]->mcs_count;
}