Tuning Parallel Code on Solaris — Lessons Learned from HPC

Dani Flexer *dani@daniflexer.com*

Presentation to the London OpenSolaris User Group

Based on a Sun White Paper of the same name published 09/09

23/9/2009

Agenda

- Background
- Performance analysis on Solaris
- Examples of using DTrace for performance analysis
  - Thread scheduling
  - I/O performance
- Conclusion

Background

- Business processing increasingly requires parallel applications
    - Multicore CPUs dominant
    - Multi-server and multi-CPU applications prevalent
    - *Both models perform best with parallel code*
- Performance tuning of parallel code is required in most environments

Challenges

- Due to the complex interactions in parallel systems, tuning parallel code in test environments is often ineffective
- Conventional tools are not well suited to analysis of parallel code
- Tuning production environments with most conventional tools is risky

Some System Analysis Tools

- intrstat — gathers and displays run-time interrupt statistics
- busstat — reports memory bus related performance statistics
- cputrack, cpustat — monitor system and/or application performance using CPU hardware counters
- trapstat — reports trap statistics
- prstat — reports active process statistics
- vmstat — reports memory statistics

Studio Performance Analyzer

- Collector — collects performance related data for an application
- Analyzer — analyzes and displays data
- Can run directly on unmodified production code
- Supports
  - Clock-counter and hardware-counter memory allocation tracing
  - Other hardware counters
  - MPI tracing

dani@daniflexer.com

DTrace

- A framework that allows the dynamic instrumentation of both kernel and user level code
- Permits users to trace system data safely without affecting performance
- Programmable in D
  - No control statements — flow depends on state of specific data through *predicates*

dani@daniflexer.com

Observability — a key Solaris design goal

- *Observability* is a measure for how well internal states of a system can be inferred by knowledge of its external outputs. *Wikipedia*
- DTrace is arguably the best observability tool available

A few questions suitable for a quick, initial diagnosis

- Are there a lot 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   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?

Analyzing results of `prstat`

Col	Meaning	If the value seems high …
USR	% user mode	Profile user mode with DTrace using either pid or profile providers
SYS	% system mode	Profile the kernel
LCK	% waiting for locks	Use plockstat DTrace provider to see which user locks are used extensively
SLP	% sleeping	Use sched DTrace provider and view call stacks with DTrace to see why threads are sleeping
TFL/ DFL	% processing page faults	Use the vminfo DTrace provider to identify the source of the page faults

Two practical examples

- Thread Scheduling Analysis
- I/O Performance Problems
- *See the White Paper for more!*

dani@daniflexer.com

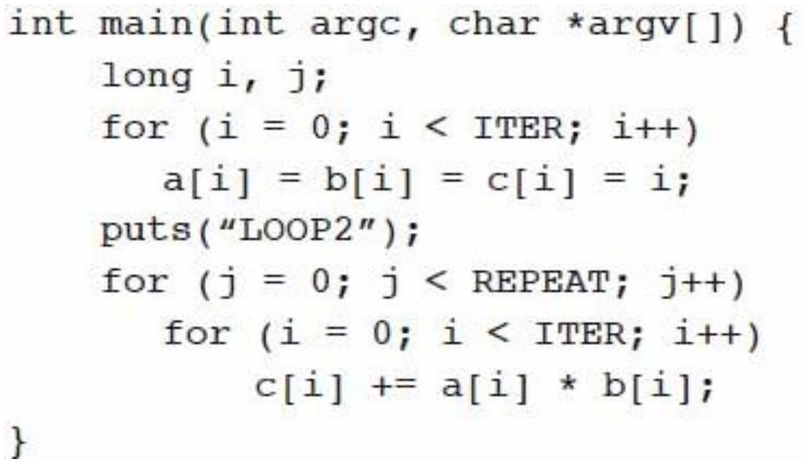
Thread Scheduling Analysis (1)

- Performance of a multithreaded application requires balanced allocation of cores to threads
- Analyzing thread scheduling on the different cores can help tune multithreaded applications

dani@daniflexer.com

Thread Scheduling Analysis (2)

- Use -xautopar to compile
- Compiler automatically generates multithreaded code that uses OpenMP
- Program is CPU bound



```
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];
}
```

Thread Scheduling Analysis (3)

```
#!/usr/sbin/dtrace -s
#pragma D option quiet
BEGIN
{
  baseline = walltimestamp;
  scale = 1000000;
}
sched:::on-cpu
/ pid == $target && !self->stamp /
{
  self->stamp = walltimestamp;
  self->lastcpu = curcpu->cpu_id;
  self->lastlgrp = curcpu->cpu_lgrp;
  self->stamp = (walltimestamp – baseline) / scale;
  printf(“%9d:%-9d TID %3d CPU %3d(%d) created\n”,
  self->stamp, 0, tid, curcpu->cpu_id, curcpu->cpu_lgrp);
}
```

BEGIN fires when the script starts and initializes the baseline timestamp from walltimestamp DTrace timestamps are in nanos so measurement is scaled down to milliseconds (scale)

sched:::on-cpu *fires when a thread is scheduled to run*

pid == $target ensures that probe fires for processes that are controlled by this script

Thread Scheduling Analysis (4)

- Thread switches from one CPU to another

```
sched:::on-cpu
/ pid == $target && self->stamp && self->lastcpu != \
                                          curcpu->cpu_id /
```

- Thread is rescheduled to run on the same CPU it ran on the previous time it was scheduled to run

```
sched:::on-cpu
/ pid == $target && self->stamp && self->lastcpu == \
                                          curcpu->cpu_id /
```

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

```
sched:::off-cpu
/ pid == $target && self->stamp /
```

Thread Scheduling Analysis (5)

```
sched:::sleep
/ pid == $target /
{
  self->sobj = (curlwpsinfo->pr_stype == SOBJ_MUTEX ?
  “kernel mutex” : curlwpsinfo->pr_stype == SOBJ_RWLOCK ?
  “kernel RW lock” : curlwpsinfo->pr_stype == SOBJ_CV ?
  “cond var” : curlwpsinfo->pr_stype == SOBJ_SEMA ?
  “kernel semaphore” : curlwpsinfo->pr_stype == SOBJ_USER ?
  “user-level lock” : curlwpsinfo->pr_stype == SOBJ_USER_PI ?
  “user-level PI lock” : curlwpsinfo->pr_stype ==
  SOBJ_SHUTTLE ? “shuttle” : “unknown”);
  self->delta = (walltimestamp - self->stamp) /scale;
  self->stamp = walltimestamp;
  self->stamp = (walltimestamp - baseline) / scale;
  printf(“%9d:%-9d TID %3d sleeping on ‘%s’\n”,
  self->stamp, self->delta, tid, self->sobj);
}
```

This code runs when sched:::sleep probe fires before the thread sleeps on a synchronization object and the type of synchronization object is printed

Thread Scheduling Analysis (6)

```
sched:::sleep
/ pid == $target && ( curlwpsinfo->pr_stype == SOBJ_CV ||
curlwpsinfo->pr_stype == SOBJ_USER ||
curlwpsinfo->pr_stype == SOBJ_USER_PI) /
{
  ustack();
}
```

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.

Thread Scheduling Analysis (7)

- The psrset command is used to set up a processor set with two CPUs (0, 4) to simulate CPU over-commitment:

```
host# psrset -c 0 4
```

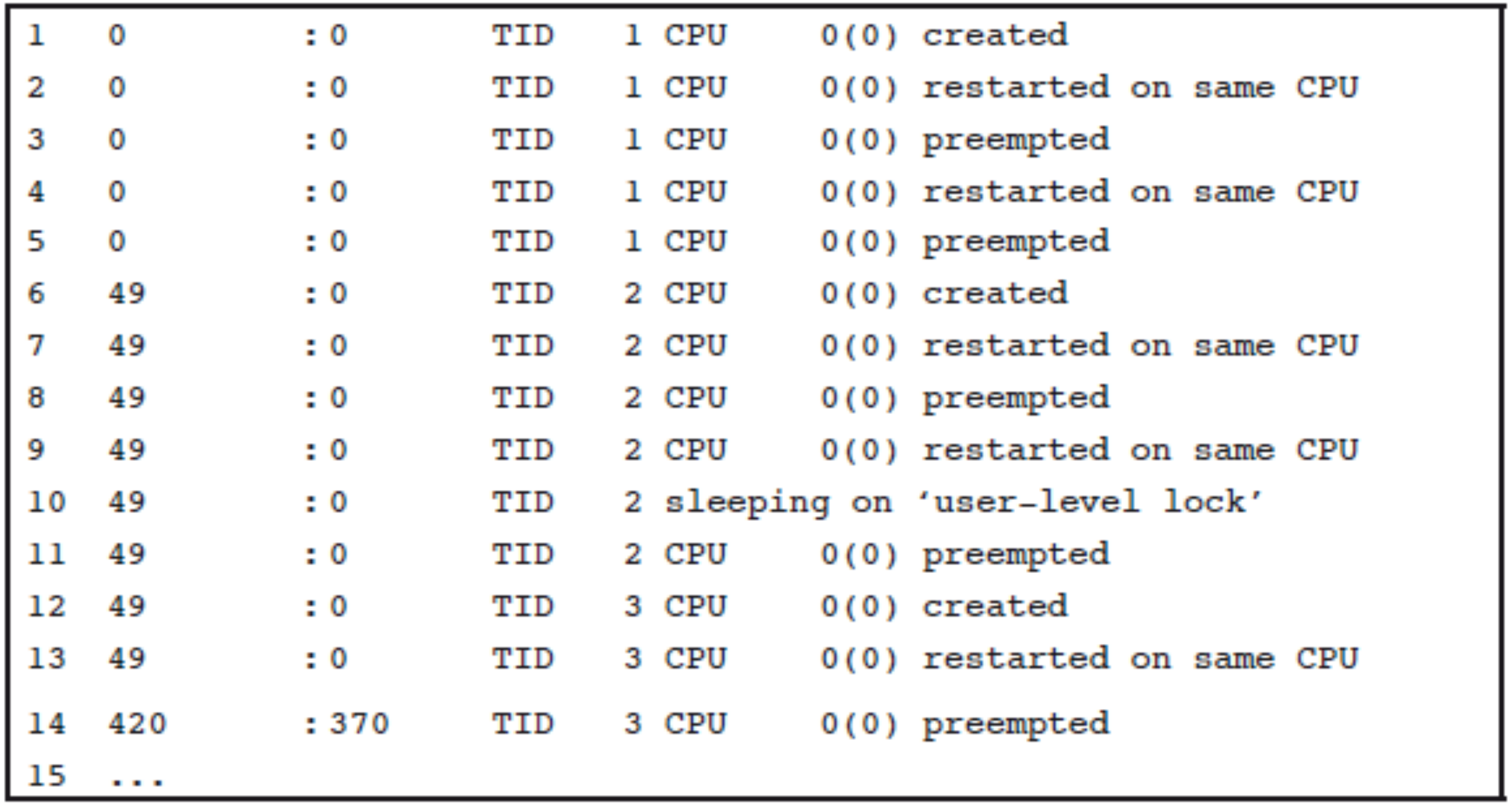
- The number of threads is set to three with the `OMP_NUM_THREADS` environment variable and threadsched.d is executed with partest:

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

dani@daniflexer.com

Thread Scheduling Analysis (8)

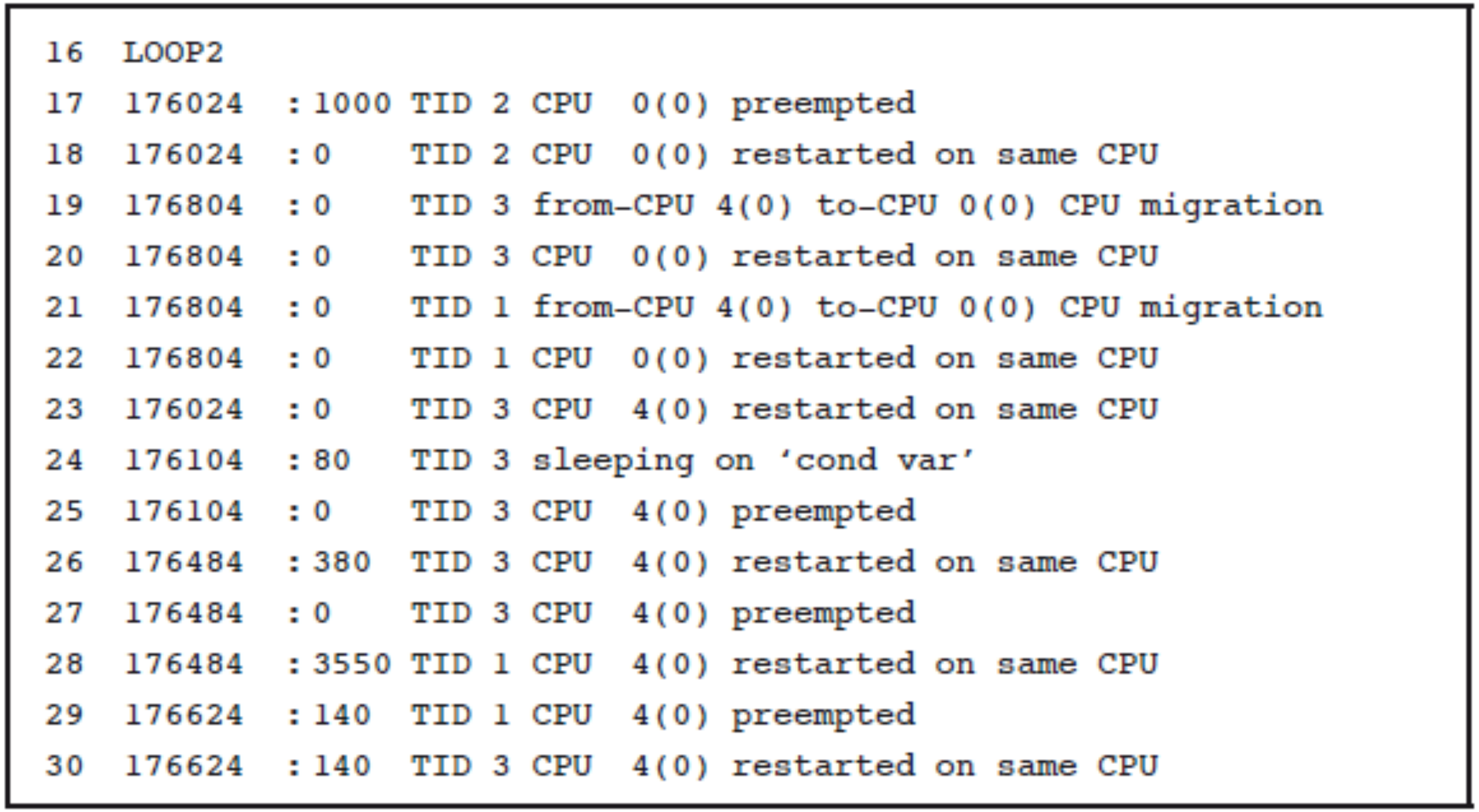
The output first shows the startup of the main thread (lines 1 to 5). The second thread first runs at line 6 and the third at line 12:



```
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   ...
```

Thread Scheduling Analysis (9)

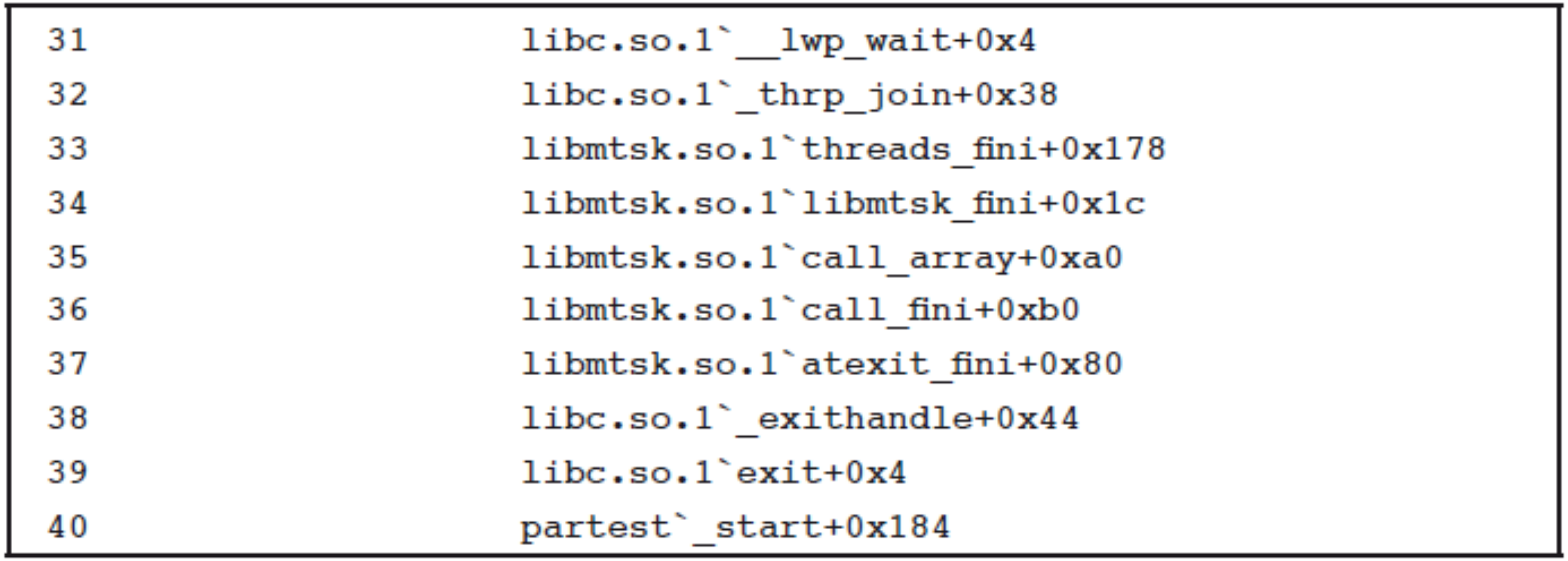
As the number of available CPUs is set to two, only two of the three threads can run simultaneously resulting in many thread migrations between CPUs. On line 24, thread 3 goes to sleep:



```
16  LOOP2
17  176024  :1000 TID 2 CPU  0(0) preempted
18  176024  :0    TID 2 CPU  0(0) restarted on same CPU
19  176804  :0    TID 3 from-CPU 4(0) to-CPU 0(0) CPU migration
20  176804  :0    TID 3 CPU  0(0) restarted on same CPU
21  176804  :0    TID 1 from-CPU 4(0) to-CPU 0(0) CPU migration
22  176804  :0    TID 1 CPU  0(0) restarted on same CPU
23  176024  :0    TID 3 CPU  4(0) restarted on same CPU
24  176104  :80   TID 3 sleeping on 'cond var'
25  176104  :0    TID 3 CPU  4(0) preempted
26  176484  :380  TID 3 CPU  4(0) restarted on same CPU
27  176484  :0    TID 3 CPU  4(0) preempted
28  176484  :3550 TID 1 CPU  4(0) restarted on same CPU
29  176624  :140  TID 1 CPU  4(0) preempted
30  176624  :140  TID 3 CPU  4(0) restarted on same CPU
```

Thread Scheduling Analysis (10)

From line 31, 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:



```
31                      libc.so.1`__lwp_wait+0x4
32                      libc.so.1`_thrp_join+0x38
33                      libmtsk.so.1`threads_fini+0x178
34                      libmtsk.so.1`libmtsk_fini+0x1c
35                      libmtsk.so.1`call_array+0xa0
36                      libmtsk.so.1`call_fini+0xb0
37                      libmtsk.so.1`atexit_fini+0x80
38                      libc.so.1`_exithandle+0x44
39                      libc.so.1`exit+0x4
40                      partest`_start+0x184
```

I/O Performance Problems (1)

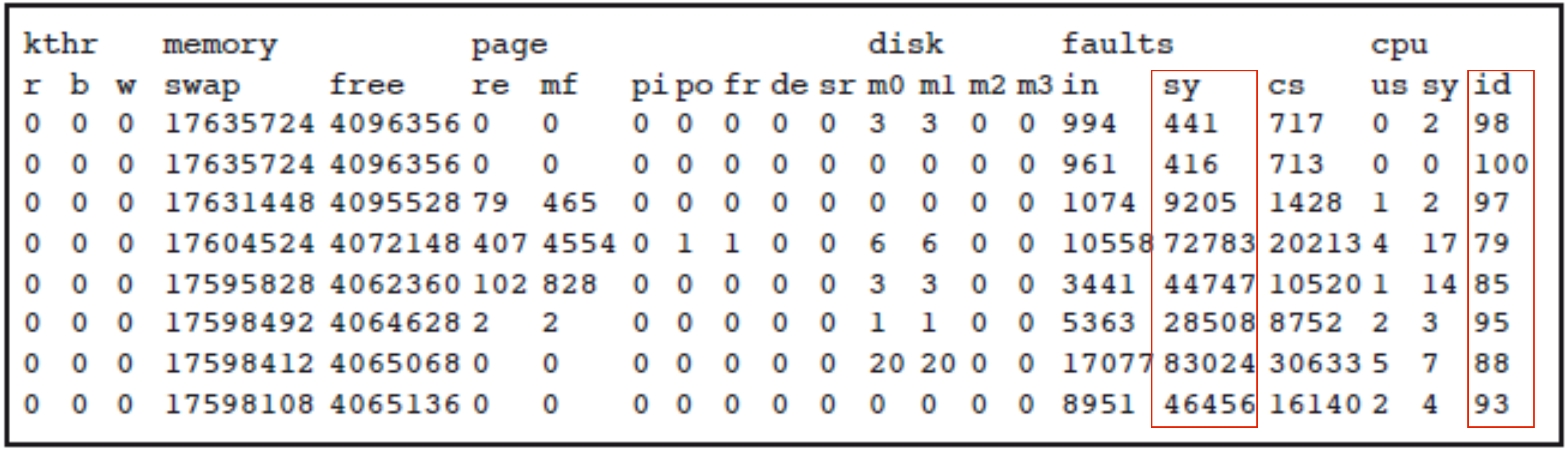
- Sluggishness due to a high rate of I/O system calls is a common problem
- To identify the cause it is necessary to determine:
  - Which system calls are called
  - What frequency
  - By which process
  - Why?
- Good tools for initial analysis: vmstat, prstat

I/O Performance Problems (2)

- In this example:
  - A Windows 2008 server is virtualized on OpenSolaris 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

I/O Performance Problems (3)

- vmstat results:

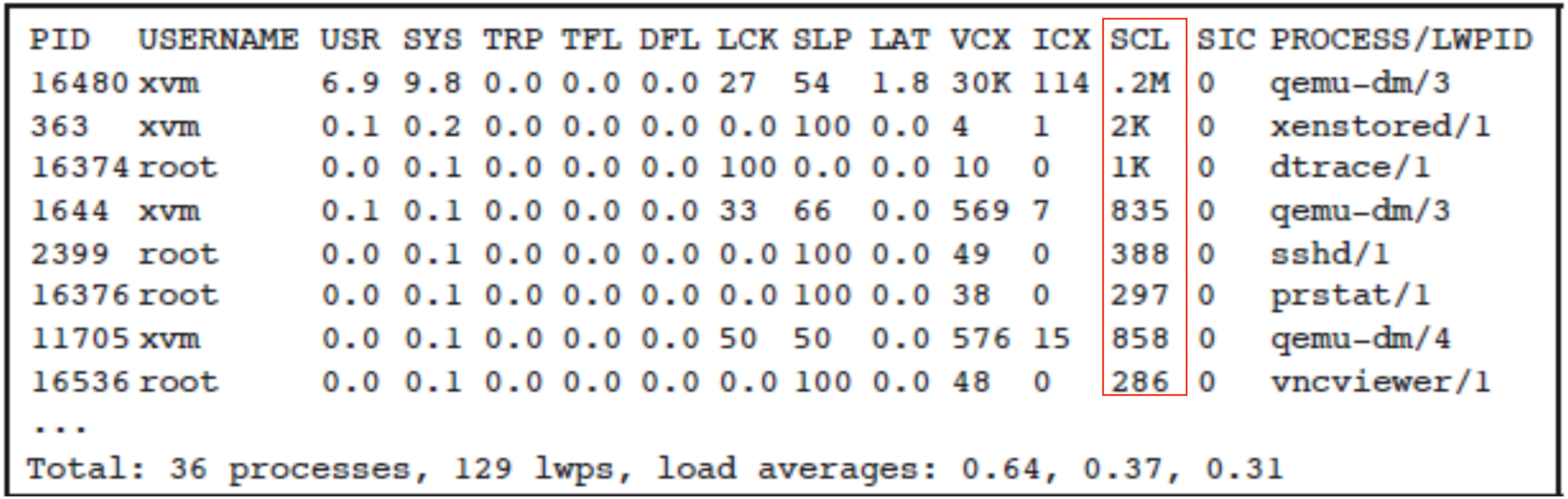


```
kthr    memory                  page                    disk          faults                  cpu
r  b  w swap     free    re  mf   pi po fr de sr m0 m1 m2 m3 in    sy    cs    us sy id
0  0  0 17635724 4096356 0   0    0  0  0  0  0  3  3  0  0  994   441   717   0  2  98
0  0  0 17635724 4096356 0   0    0  0  0  0  0  0  0  0  0  961   416   713   0  0  100
0  0  0 17631448 4095528 79  465  0  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  0  10558 72783 20213 4  17 79
0  0  0 17595828 4062360 102 828  0  0  0  0  0  3  3  0  0  3441  44747 10520 1  14 85
0  0  0 17598492 4064628 2   2    0  0  0  0  0  1  1  0  0  5363  28508 8752  2  3  95
0  0  0 17598412 4065068 0   0    0  0  0  0  0  20 20 0  0  17077 83024 30633 5  7  88
0  0  0 17598108 4065136 0   0    0  0  0  0  0  0  0  0  0  8951  46456 16140 2  4  93
```

- # system calls (sy) grows and stays high while CPU is more than 79% idle (id)
- A CPU-bound workload on this system normally generates <5,000 calls per interval, here it is >9,000 up to 83,000

I/O Performance Problems (4)

- `prstat -Lm` results:



```
PID   USERNAME USR SYS TRP TFL DFL LCK SLP LAT VCX ICX SCL SIC PROCESS/LWPID
16480 xvm      6.9 9.8 0.0 0.0 0.0 27  54  1.8 30K 114 .2M 0   qemu-dm/3
363   xvm      0.1 0.2 0.0 0.0 0.0 0.0 100 0.0 4   1   2K  0   xenstored/1
16374 root     0.0 0.1 0.0 0.0 0.0 100 0.0 0.0 10  0   1K  0   dtrace/1
1644  xvm      0.1 0.1 0.0 0.0 0.0 33  66  0.0 569 7   835 0   qemu-dm/3
2399  root     0.0 0.1 0.0 0.0 0.0 0.0 100 0.0 49  0   388 0   sshd/1
16376 root     0.0 0.1 0.0 0.0 0.0 0.0 100 0.0 38  0   297 0   prstat/1
11705 xvm      0.0 0.1 0.0 0.0 0.0 50  50  0.0 576 15  858 0   qemu-dm/4
16536 root     0.0 0.1 0.0 0.0 0.0 0.0 100 0.0 48  0   286 0   vncviewer/1
...
Total: 36 processes, 129 lwps, load averages: 0.64, 0.37, 0.31
```

- qemu-dm executes a very large number of system calls (200K) SCL
- 100X more than xenstored in 2nd place
- Need to drill down to find out which system call and why

I/O Performance Problems (5)

- count_syscalls.d, prints call rates for the top-ten processes/system calls every 5 seconds:

```
 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(@c, 10);
11   normalize(@call_count, (timestamp-timer) / 1000000000);
12   printa(?%5d %-20s %6@d %s\n?, @call_count);
13   clear(@call_count);
14   printf(?\n?);
15   timer = timestamp;
16 }
```

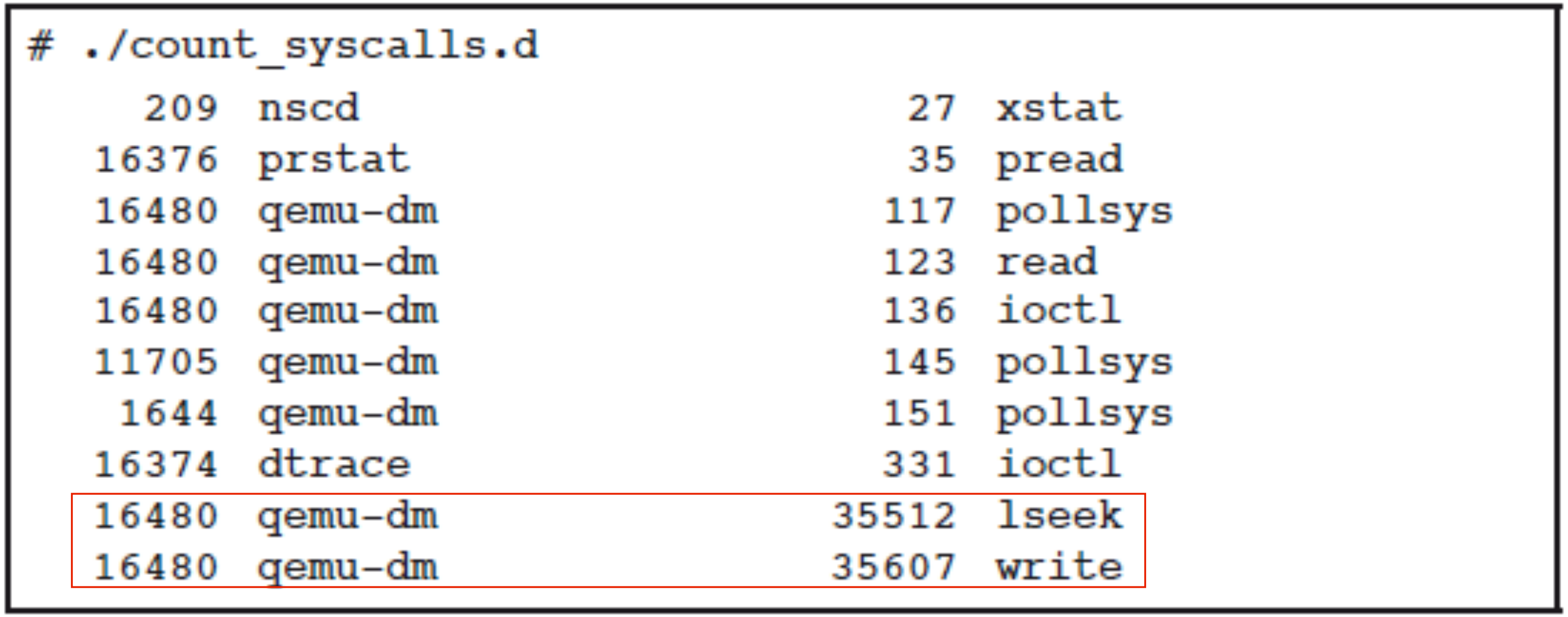
The syscall:::entry probe fires for each system call.

The system call name, executable, and PID are saved in the call_count aggregation

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 aggregation.

I/O Performance Problems (6)

- When count_syscalls.d is run, qemu-dm is clearly creating the load, primarily through calls to `write` and `lseek`:



```
# ./count_syscalls.d
    209  nscd                              27  xstat
  16376  prstat                            35  pread
  16480  qemu-dm                          117  pollsys
  16480  qemu-dm                          123  read
  16480  qemu-dm                          136  ioctl
  11705  qemu-dm                          145  pollsys
   1644  qemu-dm                          151  pollsys
  16374  dtrace                           331  ioctl
  16480  qemu-dm                        35512  lseek
  16480  qemu-dm                        35607  write
```

I/O Performance Problems (7)

- To see why qemu-dm is making these calls, qemu-stat.d is implemented to collect statistics of the I/O calls, focusing on `write` (not shown) and `lseek`:

```
 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 }
```

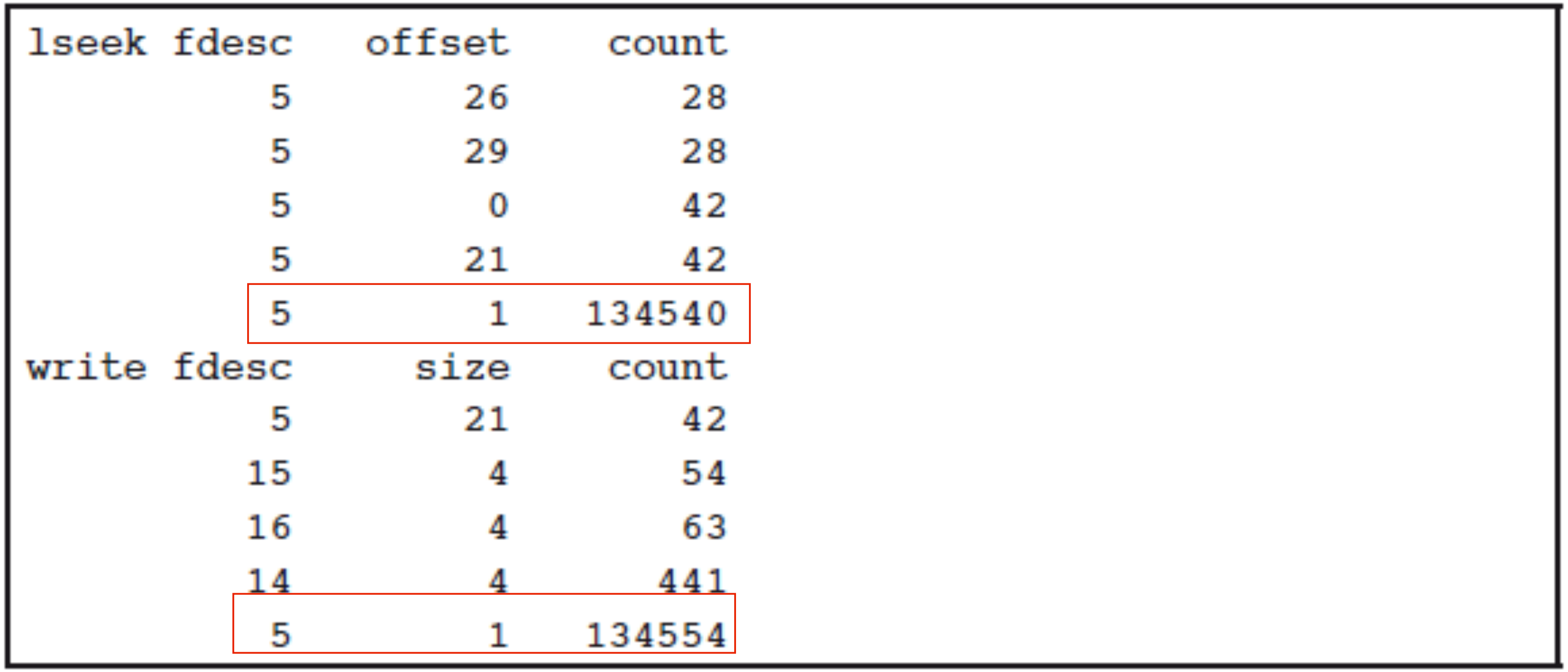
Probes called only if the triggering call to lseek sets the file pointer to an absolute value, (arg2 - whence - SEEK_SET)

The difference between the current and previous position of the file pointer is used as the second index of the aggregation in line 9

To determine the I/O pattern, the script saves the absolute position of the file pointer passed to lseek() in the variable seek in line 10

I/O Performance Problems (8)

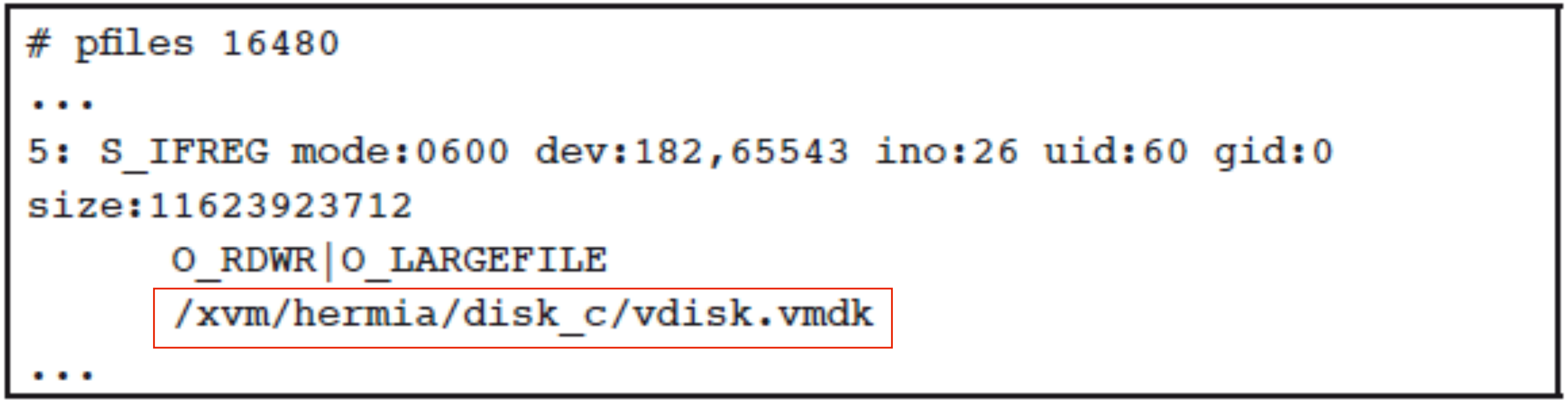
- Results show massive number of calls to file descriptor 5, moving the descriptor by offset 1, and writing a single byte
- In other words, qemu-dm writes a data stream as single bytes, without any buffering



```
lseek fdesc    offset     count
          5        26        28
          5        29        28
          5         0        42
          5        21        42
          5         1    134540
write fdesc      size     count
          5        21        42
         15         4        54
         16         4        63
         14         4       441
          5         1    134554
```

I/O Performance Problems (9)

- The `pfiles` command identifies 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
...
```

I/O Performance Problems (10)

- Next qemu-callstack.d is implemented to see where the calls to lseek originate by viewing the call stack
- Script prints the three most common call stacks for the lseek and write system calls every five seconds

```
 1 #!/usr/sbin/dtrace -s
 2 #pragma D option quiet
 3 syscall::lseek:entry, syscall::write:entry
 4 / execname == “qemu-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.

I/O Performance Problems (11)

- Looking at the most common stack trace:

```
write
      libc.so.1`__write+0xa
      qemu-dm`RTFileWrite+0x37
      qemu-dm`RTFileWriteAt+0x48
      qemu-dm`vmdkWriteDescriptor+0x1d5
      qemu-dm`vmdkFlushImage+0x23
      qemu-dm`vmdkFlush+0x9
      qemu-dm`VDFlush+0x91
      qemu-dm`vdisk_flush+0x1c
      qemu-dm`bdrv_flush+0x2e
      qemu-dm`ide_write_dma_cb+0x187
      qemu-dm`bdrv_aio_bh_cb+0x16
      qemu-dm`qemu_bh_poll+0x2d
      qemu-dm`main_loop_wait+0x22c
      qemu-dm`main_loop+0x7a
      qemu-dm`main+0x1886
      qemu-dm`_start+0x6c
      28758
```

- The stack trace shows that the virtual machine is flushing the disk cache for every byte indicating a disabled disk cache
- Later it was discovered that when an MS server is an Active Directory domain controller, the directory service writes unbuffered and disables the disk write cache on certain volumes

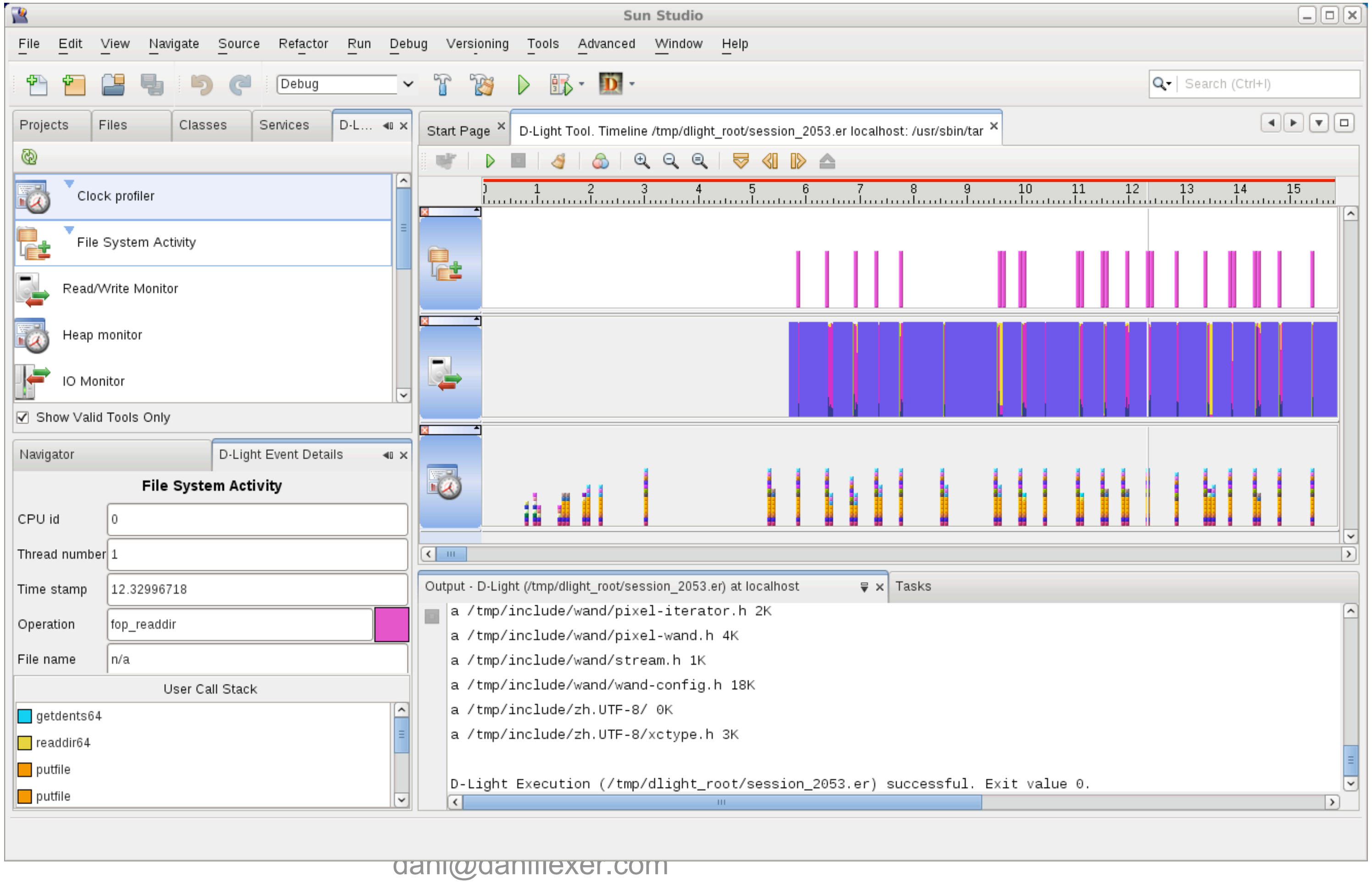
Other Examples in the Doc

- Additional detailed examples:
  - Improving performance by reducing data cache misses
  - Improving scalability by avoiding memory stalls
  - Memory placement optimization with OpenMP
  - Using DTrace with MPI
- These use a wider range of tools, including:
  - Sun Studio Performance Analyzer
  - busstat, cpustat, cputrack
  - gnuplot

dani@daniflexer.com

DTrace — not just text

DLight
(SS12)



File System Activity

CPU id: 0
Thread number: 1
Time stamp: 12.32996718
Operation: fop_readdir
File name: n/a

User Call Stack

- getdents64
- readdir64
- putfile
- putfile

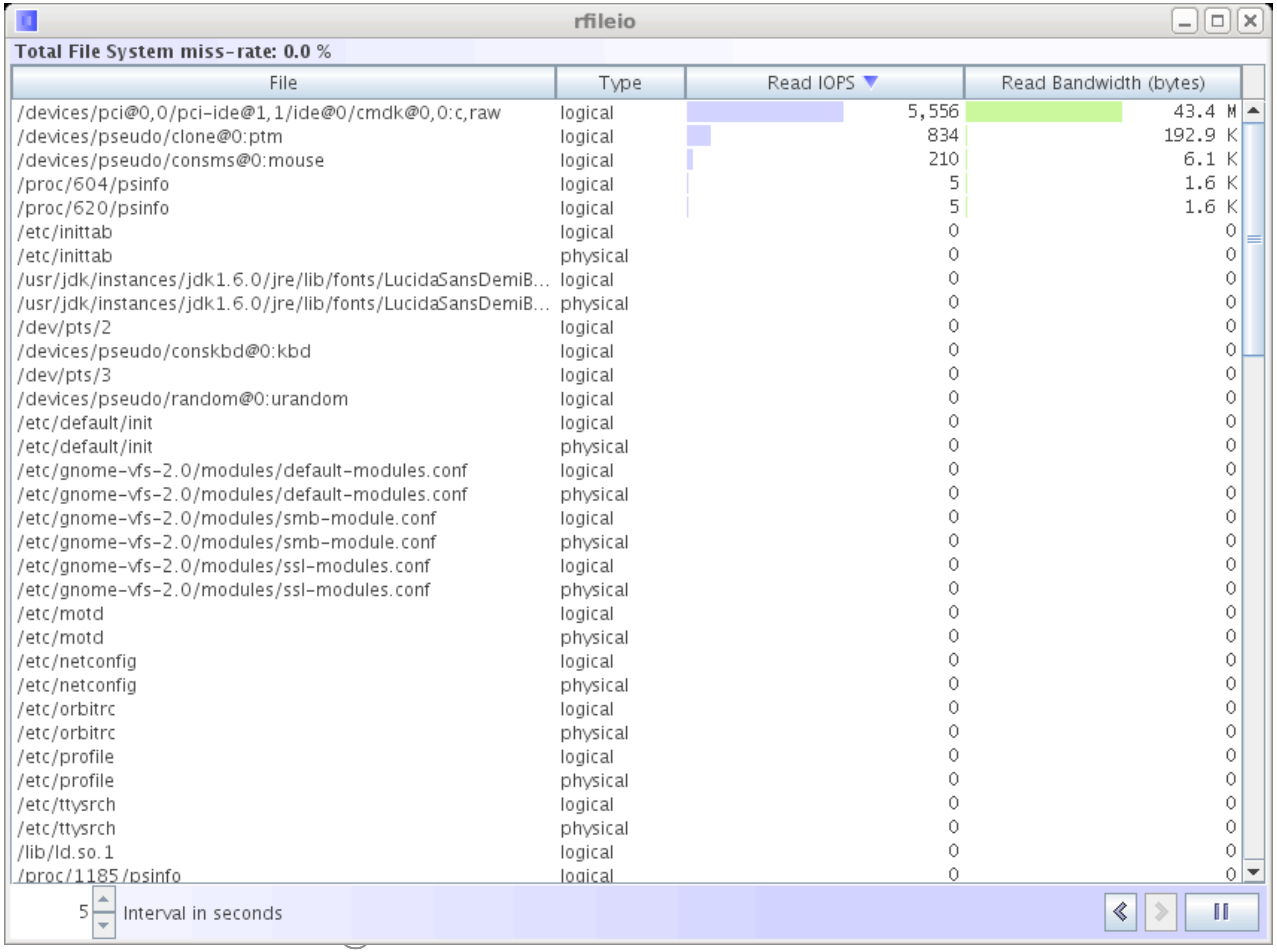
```
a /tmp/include/wand/pixel-iterator.h 2K
a /tmp/include/wand/pixel-wand.h 4K
a /tmp/include/wand/stream.h 1K
a /tmp/include/wand/wand-config.h 18K
a /tmp/include/zh.UTF-8/ 0K
a /tmp/include/zh.UTF-8/xctype.h 3K

D-Light Execution (/tmp/dlight_root/session_2053.er) successful. Exit value 0.
```

dani@danilexer.com

DTrace — not just text

Chime
(NetBeans)



Total File System miss-rate: 0.0 %

File	Type	Read IOPS ▼	Read Bandwidth (bytes)
/devices/pci@0,0/pci-ide@1,1/ide@0/cmdk@0,0:c,raw	logical	5,556	43.4 M
/devices/pseudo/clone@0:ptm	logical	834	192.9 K
/devices/pseudo/consms@0:mouse	logical	210	6.1 K
/proc/604/psinfo	logical	5	1.6 K
/proc/620/psinfo	logical	5	1.6 K
/etc/inittab	logical	0	0
/etc/inittab	physical	0	0
/usr/jdk/instances/jdk1.6.0/jre/lib/fonts/LucidaSansDemiB...	logical	0	0
/usr/jdk/instances/jdk1.6.0/jre/lib/fonts/LucidaSansDemiB...	physical	0	0
/dev/pts/2	logical	0	0
/devices/pseudo/conskbd@0:kbd	logical	0	0
/dev/pts/3	logical	0	0
/devices/pseudo/random@0:urandom	logical	0	0
/etc/default/init	logical	0	0
/etc/default/init	physical	0	0
/etc/gnome-vfs-2.0/modules/default-modules.conf	logical	0	0
/etc/gnome-vfs-2.0/modules/default-modules.conf	physical	0	0
/etc/gnome-vfs-2.0/modules/smb-module.conf	logical	0	0
/etc/gnome-vfs-2.0/modules/smb-module.conf	physical	0	0
/etc/gnome-vfs-2.0/modules/ssl-modules.conf	logical	0	0
/etc/gnome-vfs-2.0/modules/ssl-modules.conf	physical	0	0
/etc/motd	logical	0	0
/etc/motd	physical	0	0
/etc/netconfig	logical	0	0
/etc/netconfig	physical	0	0
/etc/orbitrc	logical	0	0
/etc/orbitrc	physical	0	0
/etc/profile	logical	0	0
/etc/profile	physical	0	0
/etc/ttysrch	logical	0	0
/etc/ttysrch	physical	0	0
/lib/ld.so.1	logical	0	0
/proc/1185/psinfo	logical	0	0

5 Interval in seconds

Conclusions

- Computing is getting more complex
  - Multiple CPUs, cores, threads, virtualized operating systems, networking, and storage devices
- Serious challenges to architects, administrators, developers, and users
  - Need high availability and reliability
  - Increasing pressure on datacenter infrastructure, budgets, and resources
- Need to maintain systems at a high level of performance — without adding resources
- Demand control through optimization is the most cost efficient way to grow DC capacity

dani@daniflexer.com

Conclusions

- To achieve these objectives, OpenSolaris has a comprehensive set of tools with DTrace at their core
  - Enable unprecedented levels of observability and insight into the workings of the operating system and the applications running on it
  - Tools allow you to quickly analyze and diagnose issues without increasing risk
- Observability is a primary driver of consistent system performance and stability

Thanks!

- Technical content and experience provided by Thomas Nau of the Infrastructure Department, Ulm University, Germany
  - Except section on MPI
- Paper recently published by Sun see:
  - http://sun.com/solutions/hpc/resources.jsp (under White Papers)
  - http://sun.com/solutions/hpc/development.jsp (under Sun Tools and Services)
- Dani Flexer — dani@daniflexer.com

Q&A

dani@daniflexer.com