Solaris Performance Metrics

–

Disk Utilisation by Process

10th December 2005

Brendan Gregg

[Sydney, Australia]

Abstract

This paper presents new metrics for monitoring disk utilisation by process for the Solaris 10™ operating environment by Sun Microsystems™. How to measure and understand disk utilisation by process is discussed, as well as the origin of the measurements. Facilities used to monitor disk utilisation include procfs, TNF tracing and DTrace. [1]

1 This is one part of a series of papers I'm planning that cover performance monitoring metrics.

Table of Contents

1. Introduction

Checking CPU usage by process is a routine task for Solaris system administrators, tools such as `prstat` or `ps` provide a number of CPU metrics. However checking *disk usage by process* has been difficult to monitor on Solaris 9 and earlier, with no Solaris command providing such statistics.[2]

Question

On Solaris 9 or earlier, how do you measure disk I/O by process?

No, iostat does not have a switch for that...

These days disk I/O is often the bottleneck in a system, spurring the use of volume managers, disk arrays and storage area networks. For an administrator to identify disk I/O as the bottleneck, the `iostat` command can be used. An artful system administrator can interpret extra details from iostat's output, such as if the disk activity is likely to be random or sequential. But there is no way to determine disk I/O details by process, nor does iostat have a switch for that[3].

Now, there *are* some Neanderthal-like ways to bash this data from the system. The most entertaining are,

- Freeze every process on the system in turn while watching `iostat`. If the disk load vanishes you have found your culprit.
- Create a separate mount point for every process in the system, from which the applications are run. Now `iostat -xnmp` has separate details per process.

This paper will focus on *sensible* ways to monitor disk usage by process.

A variety of monitoring solutions will be presented with their strengths and weaknesses discussed. Since we will be introducing new techniques, the origin of the data used and the algorithms applied will be covered carefully. A short summary for various background topics such as procfs and DTrace will be provided. Apologies to experienced readers who may encounter a few pages of recap.

This paper is one in a series of papers covering performance monitoring metrics in Solaris. The topics covered in this paper are highlighted in the following matrix[4],

Resource	Qualifier	Scope
CPU	**Utilisation**	by System
Memory	Saturation	**by Process**
Disk	Errors	
Network		

Table 1. Resource Monitoring Matrix

Rather than use the term "usage", the terms "utilisation" and "saturation" will be used along with the following descriptions. This paper in particular covers: *Disk Utilisation by Process.*

Utilisation[5] can be measured as a percentage of the resource that was in use. This is usually presented as an average measured over a time interval.

Saturation is a measure of the work that has queued waiting for the resource. This is usually presented as an average over time, however it can also be measured at a particular point in time.

2 If you are using VxFS, you can use vxstat for per process disk statistics.
3 At least, not as of Solaris 10.
4 Other combinations will be covered in future papers.
5 This is the Australian spelling, other countries may spell this as "Utilization".

2. Strategies

Lets start with a summary of techniques available to measure disk I/O utilisation by process.

2.1. Existing Tools

There are no tools in Solaris that achieve this directly, for example a switch on `prstat` or `iostat`.

2.2. Additional Tools[6]

- **pea.se** from the SE Toolkit[7] reports on the size of disk activity.
- **prusage** is a Perl/Kstat tool[8] to report on the size of disk activity.
- **psio** is a Perl/TNF tracing tool[9] that can report the size and time consumed by disk activity.
- **iosnoop** from the DTraceToolkit[10] is a shell/DTrace tool to print disk events, size and time.
- **iotop** from the DTraceToolkit is a shell/DTrace tool to report a summary of disk size or time.

2.3. Available Strategies

- **procfs** – the process filesystem contains information on disk size totals.
- **TNF tracing** – kernel tracing lets us monitor disk events, including size and time.
- **DTrace** – dynamic tracing lets us safely monitor disk events, including size and time.

2.4. Monitoring Goals

- **Disk Utilisation by process** – a value to represent disk resource consumption.

2.4. Not Covered

- **Performance *Tuning*** – After identifying a problem, how to fix it. This would be a good topic for a separate paper, however it would need to cover many application specific strategies.[11]
- **Advanced Storage Devices** – such as volume managers, storage area networks and disk arrays. Many of the algorithms presented still apply to these devices, however specific eccentricities are not covered.

6 The tools we focus on are both freeware and opensource. So there are no license fees, and the source can be inspected before use.
7 http://www.sunfreeware.com is the current location of the SE Toolkit.
8 http://www.brendangregg.com/psio.html
9 also at http://www.brendangregg.com/psio.html
10 http://www.opensolaris.org/os/community/dtrace/dtracetoolkit, or http://www.brendangregg.com/dtrace.html
11 And there are already some good books on this. See section 5. References.

3. Details

A close look at existing tools, additional tools, strategies and solutions.

3.1. Utilisation

How much is each process utilising the disks. We could either examine the size of the disk events, or the service times of the disk events. We are after a value that can be used for comparisons, such as a percentage.

3.1.1. iostat

Since we are talking disk usage, we'll start with our old friend **iostat**,

```
$ iostat -xnmpz 5
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
    0.0    0.0    0.2    0.1  0.0  0.0   12.4    6.3   0   0 c0t0d0
    0.0    0.0    0.2    0.1  0.0  0.0   12.4    6.2   0   0 c0t0d0s0 (/)
    0.0    0.0    0.0    0.0  0.0  0.0   39.0   20.4   0   0 c0t0d0s1
    0.0    0.0    0.0    0.0  0.0  0.0    2.2    9.6   0   0 c0t0d0s3 (/var)
    0.0    0.0    0.0    0.0  0.0  0.0    0.0    3.0   0   0 c0t2d0
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
  167.9    0.2  217.1    0.2  0.0  0.6    0.0    3.5   0  59 c0t0d0
  166.3    0.2  204.2    0.2  0.0  0.6    0.0    3.4   0  57 c0t0d0s0 (/)
    0.4    0.0    3.2    0.0  0.0  0.0    0.0   19.9   0   1 c0t0d0s3 (/var)
[...]
```

Figure 1   Output of iostat

This gives us disk I/O utilisation and saturation systemwide, and is useful to first identify that a problem exists.[12] Just a quick rundown while we are here: utilisation is best determined from the percent busy column "%b", and saturation from the wait queue length column "wait". The first output is a summary since boot, followed by samples per interval.

Tip

"iostat -xnmp" is a nice combination, but don't forget to check for errors;

"iostat -xnmpe" will also print error counts, and "iostat -E" prints an extended summary.

We are interested in disk I/O by process, however `iostat` does not have a "by process" switch. `iostat` fetches its info from Kstat[13], the Kernel statistics framework. Kstat is a great resource, and is used by tools such as `vmstat`, `mpstat` and `sar`. While Kstat does track statistics by disk and by CPU, it does *not* track statistics by process – that's what **procfs** is for.

12 See my forthcoming paper titled "Solaris Performance Monitoring – Disk by System".
13 For info on Kstat, try a "man -l kstat" on Solaris.

3.1.2. procfs

procfs is responsible for tracking statistics by process, it would be the first place to look for "by process" information. In this section we try and then *fail* to find suitable details in procfs, however you may find the journey interesting. If you'd like to cut to the chase, please skip to the next section on TNF tracing.

The ps command

If you haven't encountered procfs before, the following demonstrates that procfs is used by **ps**,

```
$ truss -ftopen ps
[...]
15153:  open("/proc/0/psinfo", O_RDONLY)                = 4
15153:  open("/proc/1/psinfo", O_RDONLY)                = 4
15153:  open("/proc/2/psinfo", O_RDONLY)                = 4
15153:  open("/proc/3/psinfo", O_RDONLY)                = 4
[...]
$
$ df -k /proc
Filesystem            kbytes    used   avail capacity  Mounted on
proc                       0       0       0     0%    /proc
```

Figure 2   Examining how ps works

A "`truss -ftopen`" often reveals how tools work, here we see many files were read from /proc. /proc is the process filesystem[14] "procfs", a pseudo in-memory filesystem that contains per-process information. This exists so that user level commands can read process info via an easy[15] and well defined interface, rather than fishing this info from the depths of kernel memory. Both the `ps` and the `prstat` commands read /proc.

Neither `ps` nor the traditional "`ps -ef`" prints disk I/O by process. The "`-o`" option to `ps` does lets us customise what is printed[16], for example "`ps -eo pid,pcpu,pmem,args`". However disk I/O statistics is not currently in the list of fields that `ps` can provide, listed in Figure 3.

```
$ ps -o
ps: option requires an argument -- o
usage: ps [ -aAdeflcjLPyZ ] [ -o format ] [ -t termlist ]
        [ -u userlist ] [ -U userlist ] [ -G grouplist ]
        [ -p proclist ] [ -g pgrplist ] [ -s sidlist ] [ -z zonelist ]
  'format' is one or more of:
        user ruser group rgroup uid ruid gid rgid pid ppid pgid sid taskid ctid
        pri opri pcpu pmem vsz rss osz nice class time etime stime zone zoneid
        f s c lwp nlwp psr tty addr wchan fname comm args projid project pset
```

Figure 3   conventional ps doesn't print disk I/O

14 The man page is proc(4).
15 Since it is a filesystem, a programmer immediately knows how to do a read, write, open and close.
16 The fields are documented in the man page for ps.

The prusage struct

There *are* disk statistics in procfs somewhere. Lets take a look at the procfs header file,

```
$ cat /usr/include/sys/procfs.h
[...]
/*
 * Resource usage.  /proc/<pid>/usage /proc/<pid>/lwp/<lwpid>/lwpusage
 */
typedef struct prusage {
[...]
        ulong_t         pr_nswap;       /* swaps */
        ulong_t         pr_inblk;       /* input blocks */
        ulong_t         pr_oublk;       /* output blocks */
        ulong_t         pr_msnd;        /* messages sent */
        ulong_t         pr_mrcv;        /* messages received */
        ulong_t         pr_sigs;        /* signals received */
        ulong_t         pr_vctx;        /* voluntary context switches */
        ulong_t         pr_ictx;        /* involuntary context switches */
        ulong_t         pr_sysc;        /* system calls */
        ulong_t         pr_ioch;        /* chars read and written */
        ulong_t         filler[10];     /* filler for future expansion */
} prusage_t;
[...]
```

Figure 4 The prusage structure from procfs

The procfs header file lists many structs of related process information. Above is a portion of the "prusage" struct, which is accessible from procfs as "/proc/<pid>/usage".

I've highlighted the values **pr_inblk** and **pr_oublk**, which provide us with input and output "blocks" by process; and **pr_ioch** for characters read and written. As they are by process statistics that cover disk activity they are worth investigating here, if at the very least to explain why they *can't* be used.

Foresight

The prusage structure, like many structures in procfs, is *future safe*.

As of Solaris 10 there are six timestruc_t for "future expansion" and ten ulong_t.

If you are reading this many years since Solaris 10, check what is in prusage now (assuming it hasn't been superseded by then). More disk statistics may have been added.

A while ago I wrote a freeware tool called **`prusage`**[17] to print out this prusage struct data and other statistics. The default output of `prusage` looks like this,

```
$ prusage
   PID  MINF  MAJF     INBLK     OUBLK     CHAR-kb COMM
     3     0     0       852     57487           0 fsflush
  2143     0    11        13      4316       44258 setiathome
   407     0   897      1034       356        2833 poold
     9     0    76       155      1052      117082 svc.configd
   491     0   653       771         7      282161 Xorg
   611     0   360       407         1        2852 afterstep
   593     0   190       268         0        1062 snmpd
     7     0    73       128        24        7248 svc.startd
     1     0    91       122         0        6338 init
```

Figure 5   The prusage tool prints the prusage data

The highlighted columns are taken from the pr_inblk, pr_oublk and pr_ioch values.

Another tool to view these statistics is from the the SE Toolkit, **`pea.se`**,[18]

```
$ se -DWIDE pea.se 1 | cut -c1-36,91-118
13:05:41 name set lwmx   pid  ppid   inblk outblk chario    sysc
bash             -1   1 19768 19760    0.00   0.00      0       0
ssh              -1   1  9068 16378    0.00   0.00      6       0
ttsession        -1   2 19746     1    0.00   0.00      0       0
bash             -1   1 26061  2408    0.00   0.00      0       0
bash             -1   1 19732 19730    0.00   0.00      0       0
se.sparcv9.5.9   -1   1 10283  2533    0.00   0.00 376516    3550
dsdm             -1   1 19729     1    0.00   0.00      0       0
bash             -1   1 16378  2408    0.00   0.00      0       0
tail             -1   1  3695  3694    0.00   0.00      3      10
bash             -1   1  5412  2408    0.00   0.00      0       0
sshd             -1   1  8149  2324    0.00   0.00    437       9
[...]
```

Figure 6   The pea.se program prints prusage data

The highlighted columns also print pr_inblk, pr_oublk and pr_ioch. If needed, it would be easy to write a tool to just print these columns plus PID and process name, in the SE Toolkit's own language – SymbEL.

Myth

A common belief is that the SE Toolkit is just a collection of tools. It is much more than that – it contains a powerful interpreter for writing your own tools that gathers kernel statistics together in a meaningful way.

17 http://www.brendangregg.com/Solaris/prusage
18 the output of "se -DWIDE pea.se" is very wide, > 80 chars. I've used a cut command to keep things tidy.

Testing pr_inblk, pr_oublk

What do pr_inblk and pr_oublk *really* measure?[19] Their meaning appears lost in the mists of time, however it would seem sensible to assume that they once measured blocks, and that 1 block == 8 Kb. To quickly test what they are now I wrote two programs in C to do 100 well spaced reads[20], then used the `prusage` tool to see their value. The first program did reads of 50 bytes in size, the second 50 Kilobytes,

```
$ prusage -Cp `pgrep read1` 1 1
   PID  MINF  MAJF    INBLK    OUBLK   CHAR-kb COMM
 16119     0   101      106        0         5 read1
$
$ prusage -Cp `pgrep read2` 1 1
   PID  MINF  MAJF    INBLK    OUBLK   CHAR-kb COMM
 16128     0     2      125        0      5000 read2
$
```

Figure 7 Examining values from the prusage data with known activity

Tip

Writing test programs is a great way to confirm what statistics really are.

For disk I/O, remember to take caching effects into account. You could mount remount filesystems first to clear the cache.

In Figure 7 the pr_ioch values of 5 Kb and 5000 Kb were expected, however the pr_inblk values of 106 and 125 blocks don't relate. This means there is no correlation between block count and size, such as 1 block equalling 8 Kb.

The pr_inblk and pr_oublk counters are best understood by reading the **OpenSolaris source**, much of which is the same as the Solaris 10 source[21]. For some paths to a disk event we increment pr_inblk and pr_oublk[22] correctly, but for others this doesn't seem to occur[23].

Both pr_inblk and pr_oublk are still useful as *indicators* of disk I/O, perhaps you just wanted to know which processes were using the disks. So long as they aren't used as an accurate measurement of I/O size.

It is worth noting that no bundled Solaris tool actually uses these. Any statistic used by a Solaris tool is carefully tested and maintained, and bugs are submitted if the statistic breaks. pr_inblk and pr_oublk simply aren't used.

Testing pr_ioch

What about pr_ioch? That can be seen in the "CHAR-kb" column of `prusage`, and for those simple tests the values look accurate. pr_ioch is tracking the total size of read and write system calls.

In Figure 7 pr_ioch for the first test was 5 Kb, which is correct at that level: 100 x 50 byte reads =~ 5 Kb. However if we are tracking disk usage by process this isn't quite correct – 100 different disk events must at least be a disk sector in size, such that the total should be 100 x 512 bytes == 50 Kb. Other tools indicate the disk actually read 8 Kb per read – so what the disks really did should be 100 x 8 Kb == 800 Kb, not 5 Kb as reported.

19 Google didn't help – the first two hits were my own!
20 to avoid read ahead.
21 so long as the bits we are reading haven't changed between Solaris and OpenSolaris.
22 Via per thread counters: see bread_common in /usr/src/uts/common/os/bio.c for "lwp->lwp_ru.inblock++".
23 It's not obvious how UFS read aheads are measured correctly. Check for bugs related to this.

Another simpler reason why we can't use pr_ioch for tracking disk I/O size goes like this,

```
$ yes > /dev/null &
[1] 3211
$ prusage -Cp 3211 1 1
   PID  MINF  MAJF    INBLK    OUBLK   CHAR-kb COMM
  3211     0     1        2        0    107544 yes
```

Figure 8 The pr_ioch vaulue matches all read/write traffic

The `yes` command hasn't caused over 100 Mb of disk activity, pr_ioch is measuring reads and writes whether they are to disk or not. pr_ioch isn't going to help us much. It can be used as an estimate if we know the activity is mostly disk I/O.

Disk I/O size is the wrong track

So we didn't find accurate statistics for disk I/O size. pr_oublk and pr_inblk should probably have worked, so an outcome from this may be to dive into the kernel code and fix them.

However disk I/O size by process cannot be used to determine disk utilisation by process anyway, even if we had accurate size information it is at best an approximation[24]. The problem is that a byte count alone does not identify random or sequential behaviour. Many readers will be quite familiar with this principle, if not the following sections on sequential and random disk I/O should demonstrate this clearly.

Sequential Disk I/O

The `dd` command can be used to generate **sequential I/O**, as demonstrated in Figure 9,

```
# dd if=/dev/dsk/c0t0d0s0 of=/dev/null bs=128k &
[1] 3244
#
# iostat -xnmpz 5
[...]
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
  106.1    0.0 13578.4    0.0  0.0  1.7    0.0   15.9   0 100 c0t0d0
  106.1    0.0 13578.6    0.0  0.0  1.7    0.0   15.9   0 100 c0t0d0s0 (/)
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
  105.8    0.0 13543.1    0.0  0.0  1.7    0.0   16.0   0 100 c0t0d0
  105.8    0.0 13543.0    0.0  0.0  1.7    0.0   16.0   0 100 c0t0d0s0 (/)
[...]
```

Figure 9 An iostat output with sequential I/O

So when the disk is 100% busy it is pulling around **13.5 Mb/sec**.

24 Consider the following: your application does mostly sequential I/O. You know the maximum disk throughput, say 80 Mb/s. If it was currently 40 Mb/s, you could estimate a utilisation of 50%. This is our best case *approximation*.

Random Disk I/O

The `find` command generates **random I/O** by walking scattered directories, as shown in Figure 10,

```
# find / > /dev/null 2>&1 &
[1] 3237
#
# iostat -xnmpz 5
[...]
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
  223.4    0.0  342.7    0.0  0.0  0.8    0.0    3.7   0  82 c0t0d0
  223.4    0.0  342.7    0.0  0.0  0.8    0.0    3.7   0  82 c0t0d0s0 (/)
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
   15.6   82.4   71.8  172.5  5.0  2.0   51.3   20.2  51 100 c0t0d0
   15.6   82.4   71.8  172.5  5.0  2.0   51.3   20.2  51 100 c0t0d0s0 (/)
[...]
```

Figure 10 An iostat output with random I/O

When the disk is 100% busy, now we are pulling **244.3 Kb/sec** (71.8 + 172.5).

Both disks are equally busy[25], but the Kb/sec transferred is dramatically different. If we were using Kb/sec as our measure of utilisation, then we may well have grossly underestimated how busy the find command was causing the disks to be.

The problem with size information is that we don't know if it is 244 *random* Kb, or 244 *sequential* Kb. The difference between these disk access patterns in terms of utilisation can be great – 244 random Kb may equal 100% utilisation (Figure 10), but 244 sequential Kb may only equal 1.8% utilisation (calculated from Figure 9). So we must take random/sequential access into account.

How do we differentiate between random and sequential access patterns? One way would be to look at the block addresses for each disk event. Another would be to analyse the time of each event, as random access involves seeking the disk heads and rotating the disk, consuming time.

We can try the above techniques if we can trace timestamps and block addresses per disk *event*. The Solaris **TNF tracing** facility allows us to do this.

Myth

Myth: it's the size that counts!

Fact: The type of I/O (random or sequential) may be more important. Looking at I/O size ("kr/s" and "kw/s" in iostat) is a useful indicator of disk I/O, but you don't know if it is random or sequential.

(To be fair to iostat, there are techniques to estimate this behaviour based on other details iostat provides: "asvc_t" and the (r+w)/(kr+kw) ratio).

25 assuming that busy "%b" is a measurement that we can trust (yes, we pretty much can).

3.1.3. TNF tracing

The TNF tracing facility was added to the Solaris 2.5 release. It provided a standard format for adding debugging probes to programs, and tools such as `prex`, `tnfxtract` and `tnfdump` to activate and extract probe info. TNF is for Trace Normal Form, the binary output format for the probe data.

A developer could leave TNF probes in production code and only activate them on customer sites if needed, especially to analyse performance problems that only present themselves in production. An excellent overview and demonstration of TNF tracing is in Sun Performance and Tuning, 2nd edition, chapter 8. There is also an overview in the tracing(3TNF) man page.

The kernel can also be traced as around thirty TNF probes have been inserted in strategic locations. They included such probes as,

- System Call probes: syscall_start, syscall_end
- Page Fault probes: address_fault, major_fault, ...
- Local I/O probes: strategy, biodone

A full list can be found in tnf_kernel_probes(4).

strategy, biodone

Of interest here are the strategy and biodone probes from the block I/O driver.

strategy – probes the function used to initiate a block I/O event[26].

biodone – probes the function called when a block I/O event completes[27].

A disk event is requested with a strategy and then completes with a biodone.

These probes also provide details on the PID, device, block address, I/O size and time. This lets us not only examine size accurately by process, but also lets us identify random or sequential access based on the block address and device details, and to fetch timestamps for disk driver events.

Warning

The way `prex` works may seem a little *unusual*.

It allows us to communicate with the kernel and send various commands: create a buffer, enable these probes, begin tracing, stop tracing, return the buffer.

It some ways it is like sending commands to an interplanetary probe: Earth to TNF, enable these instruments! begin measurement. stop measurement. return data.

Once instructed to download, TNF will return a stream of raw data that requires post processing. You may then find you sent the wrong instructions, and need to repeat the process.

There are qualities about the `prex` and TNF tracing implementation to admire: it is really simple, and it does work. (unusual is subjective anyway).

26 There is a man page for this function, strategy(9E)
27 There is a man page for this function, biodone(9F)

prex, tnfxtract, tnfdump

The following demonstrates using TNF tracing to examine the strategy and biodone probes,

```
# prex -k
Type "help" for help ...
prex> buffer alloc 1m
Buffer of size 1048576 bytes allocated
prex> trace io
prex> enable io
prex> ktrace on
(tracing happens here)
prex> ktrace off
prex> quit
#
# tnfxtract io01.tnf
# ls -l io01.tnf
-rw-------   1 root     root     1048576 Sep 13 02:22 io01.tnf
#
```

Figure 11   Using TNF tracing

The prex command was used to create a 1 Mb buffer, enable the I/O probes and activate kernel tracing using "**ktrace on**". Trace details are stored in the ring buffer until "**ktrace off**" is issued. "**trace io**" indicates disk I/O probes (including strategy and biodone) should return probe data, and "**enable io**" activates them.

The `tnfxtract` command was used to fetch the kernel buffer and save it to a file on disk. It is in TNF format, so we use `tnfdump` to convert it into a text format for reading by eyeballs or other scripts[28].

```
# tnfdump io01.tnf
probe      tnf_name:  "pagein"  tnf_string:  "keys  vm  pageio  io;file  ../../
common/os/bio.c;line 1333;"
probe      tnf_name:  "strategy"  tnf_string:  "keys  io  blockio;file  ../../
common/os/driver.c;line 411;"
probe      tnf_name:  "biodone"  tnf_string:  "keys  io  blockio;file  ../../
common/os/bio.c;line 1222;"
probe      tnf_name:  "pageout"  tnf_string:  "keys  vm  pageio  io;file  ../../
common/vm/vm_pvn.c;line 558;"
----------------     ----------------     -----     -----     ----------     ---
------------------------- -------------------------
    Elapsed (ms)       Delta (ms)    PID LWPID    TID     CPU Probe Name
       Data / Description . . .
----------------     ----------------     -----     -----     ----------     ---
------------------------- -------------------------
[...continued...]
```

28 You can read TNF binary files directly if you like, see /usr/include/sys/tnf_com.h. There is also libtnf, which you can decipher by picking through /usr/src/lib/libtnf and /usr/src/cmd/tnf/tnfdump. Or just use tnfdump.

```
         0.000000         0.000000   917     1 0xd590de00   0 pagein
        vnode: 0xd66fd480 offset: 476545024 size: 4096
         0.011287         0.011287   917     1 0xd590de00   0 strategy
        device: 26738689 block: 205888 size: 4096 buf: 0xdb58c2c0 flags: 34078801
        55.355558        55.344271     0     0 0xd41edde0   0 biodone
        device: 26738689 block: 205888 buf: 0xdb58c2c0
        55.529550         0.173992   917     1 0xd590de00   0 pagein
        vnode: 0xd4eda480 offset: 3504209920 size: 4096
        55.532130         0.002580   917     1 0xd590de00   0 strategy
        device: 26738689 block: 206232 size: 4096 buf: 0xdb58c2c0 flags: 34078801
        66.961196        11.429066     0     0 0xd41edde0   0 biodone
        device: 26738689 block: 206232 buf: 0xdb58c2c0
        378.803659       311.842463     0     0 0xd41edde0   0 biodone
        device: 0 block: 0 buf: 0xd53f0330
[...]
```

Figure 12   TNF trace data

The output above has wrapped badly, but all the information is there. This includes,

- PID 917 caused the first pagein and strategy event (and in turn the biodone event)
- The size of this event is 4096
- The block address of this event is 205888 on device 26738689
- The elapsed or delta times can be used to determine the event took around 55 ms
- The flags provide details such as direction, read or write

PIDs for strategy look correct, but for biodone they are always 0. This is because the disk event completion is *asynchronous* to the process.

To clean up after tracing (or before tracing), run `prex -k` and issue a "buffer dealloc", "untrace $all", "disable $all". At any time run a "list probes $all" to check the trace/enable state of every probe.

Using block addresses

Block addresses are available such that random vs sequential patterns can be identified; but if we choose this approach – how do we provide a meaningful value which represents our goal of disk utilisation by process?

Just the count of how many events were or weren't random may not work – some events may be *very* random as compared to others. We could try to solve this by taking the byte size of the seek into account, however this proves difficult without information about the disk density; a high density disk may seek 1 mm to cover 10 Gb, where a low density disk may seek 5 mm to cover the same size, taking longer. There are also problems when a single disk has several active slices - each provides their own range of blocks to seek across. Calculating the total seek across slices adds to the complexity of this tactic.

So these block addresses may be useful for understanding the nature of the disk activity, but they don't easily equate into a utilisation value.

Using delta times

Since the TNF probes give us timestamps for the start and end of each disk event, delta times can be calculated. I wrote the freeware **psio**[29] tool to do this. psio runs prex, activates TNF tracing, runs tnfxtract, then processes the output of tnfdump. It associates the start event with the end event by using the device number and block address provided by the TNF probes as a key.

The default output of psio prints a disk %I/O value (I/O time based) by process,

```
# psio
     UID   PID  PPID %I/O    STIME TTY      TIME CMD
 brendan 13271 10093 65.4 23:20:16 pts/20   0:01 grep brendan contents
    root     0     0  0.0   Mar 16 ?        0:16 sched
    root     1     0  0.0   Mar 16 ?        0:10 /etc/init -
    root     2     0  0.0   Mar 16 ?        0:00 pageout
[...]
```

Figure 13 psio uses the TNF trace data for %I/O

Figure 13 shows a grep process consuming 65.4% of I/O time. Great. psio can print sizes and counts too.

Using sizes and counts

The TNF probes also provide the size of each I/O event, and by counting the number of probes seen we also know the count of I/O events. The following output of psio demonstrates using this information; the "-n" prints raw values for IOTIME (ms), IOSIZE (bytes), and IOCOUNT (number); the "-f" option prints details by filesystem, with the first line for each process the totals,

```
# psio -nf 10
     UID   PID IOTIME    IOSIZE IOCOUNT CMD
 brendan 25128   1886    347648     221 find /var
       "     "   1886    347648     221  /dev/dsk/c0t0d0s5, /var
    root     0    212     66560      27 sched
       "     "    112     45568      13  /dev/dsk/c0t0d0s5, /var
       "     "     68     11264      11  /dev/dsk/c0t0d0s6, /export/home
       "     "     33      9728       3  /dev/dsk/c0t0d0s4, /opt
 brendan 25125    189    303104      34 pine
       "     "    189    303104      34  /dev/dsk/c0t0d0s6, /export/home
[...]
```

Figure 14 psio can also print raw counts

pine has I/O totals of 189 ms for time, 303104 bytes for size, and a total count of 34 events.

To turn the bytes value into a percentage for easier comparisons, as psio did with I/O time, how do we determine the maximum bytes possible per second? This road leads back to the random vs sequential bytes problem. I/O size is an interesting statistic, but we will focus on I/O time instead

29 psio is at http://www.brendangregg.com/psio.html, and is useful for Solaris 9 and earlier.

3.1.4. I/O Time Algorithms

Disk I/O time is the most promising metric for disk utilisation, represented in previous figures as **%I/O** and **IOTIME**. I/O time takes into account seek time, rotation time, transfer time, controller and bus times, etc, and as such is an excellent metric for disk utilisation. It also has a known maximum: 1000 ms per second.

Recapping, TNF probes provide,

- strategy – the request for the disk event from the device driver.
- biodone – the disk event completion.

We can read timestamps plus other I/O details for each of these.

Simple Disk Event

We want the time the disk spends satisfying a disk request, which we'll call the "*Disk Response Time"*. Such a measurement is often called the "service time"[30]. Ideally we would be able to read event timestamps from the disk controller itself, so that we knew exactly when the heads seeked, sectors were read, etc. Instead, we have strategy and biodone events from the driver.

By measuring the time from the strategy to the biodone we have a "*Driver Response Time*". It is the closest information available to measure the disk response. In reality it includes a little extra time to arbitrate and send the request over the I/O bus, which in comparison to the disk time (which is usually measured in milliseconds) will often be negligible.

We want disk response time and we can measure driver response time. For a simple disk event they are close to being equal, so we'll start by assuming they are equal. This is illustrated in Figure 15.

Simple Disk Event

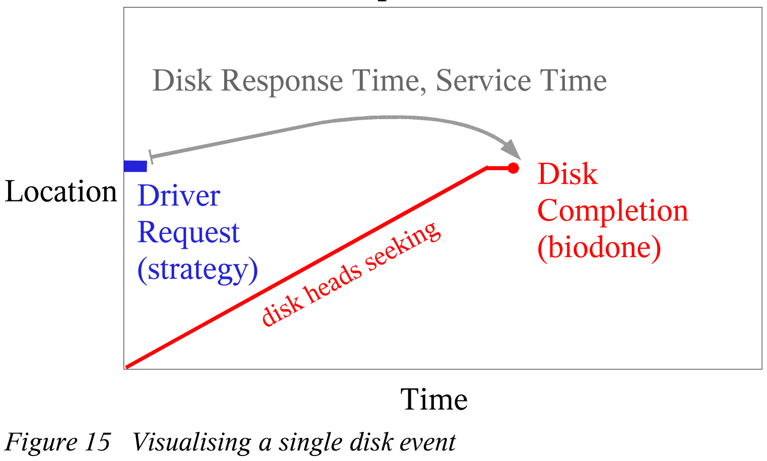
Disk Response Time, Service Time
Location
Driver Request (strategy)
disk heads seeking
Disk Completion (biodone)
Time

Figure 15 Visualising a single disk event

The strategy event is represented by a blue rectangle, the biodone by a red spot. The location of the disk head over time is traced in red[31]. The disk response time or the service time is drawn in grey.

30 The term "service time" makes more sense on older disks where it originated. For a detailed explanation and history lesson see "Clarifying disk measurements and terminology", SunWorld Online, September 1997.

31 Assuming this is an ordinary disk. Storage arrays use large front end caches, and many events will return quickly if they hit the cache. Even so, our algorithms are still of value as we concentrate on time consumed by the disk to service a request, whether that time was for mechanical events or cache activity.

The algorithm to measure disk response time would then be,

time(disk response) = time(biodone) – time(strategy)

A total by process would sum all disk response times.

Looks simple, doesn't work. Disks these days will allow multiple events to be sent to the disk where they will be queued. Modern disks will reorder the queue for optimisation, completing disk events with a minimal sweep of the heads – sometimes called “elevator seeking”. The following example will illustrate the multiple event problem.

Terminology

Disk Response Time will be used to describe the time consumed by the disk to service the event in question only.

This time starts when the disk begins to service *that* event, which may mean the heads begin to seek. The time ends when the disk completes the request.

The advantage of this measurement is it provides a known maximum for the disk – 1000 ms of disk response time per second. This helps calculate utilisation percentages.

Concurrent Disk Events 1

Lets consider five concurrent disk requests are sent at time = 0, they complete at times = 10, 20, 30, 40 and 50 ms. This is represented in Figure 16. The disk is busy processing these events from time = 0 to 50 ms, and so is busy for 50 ms.

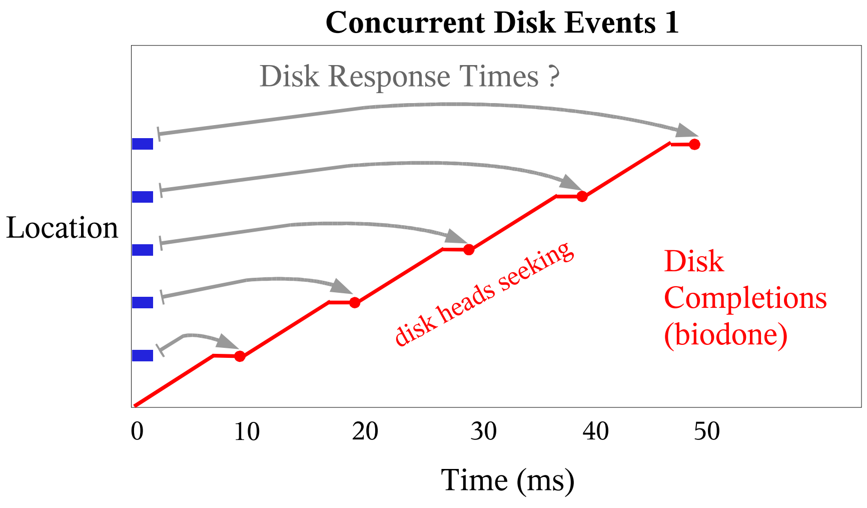

Figure 16 Measuring concurrent disk event times from strategy to biodone

The previous algorithm gives disk response times of 10, 20, 30, 40 and 50 ms. The total would then be 150 ms, implying the disk has delivered 150 ms of disk response time in only 50 ms. The problem is that we are *over counting* response times; just adding them together assumes the disk processes events one by one, which isn't always the case.

Concurrent Disk Events 2

An improved algorithm for concurrent disk requests may be to ignore the strategy events, and only measure time between the biodone events as shown in Figure 17.

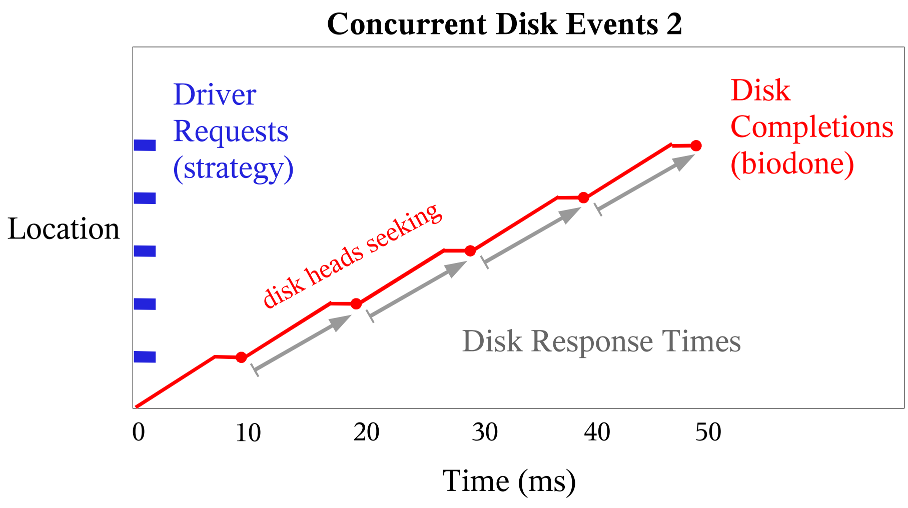

Figure 17 Measuring concurrent disk event times from consecutive biodones

(I've deliberately missed the first response time, from 0 to 10 ms. I'll get back to that)...

The following algorithm measures disk response as time between biodones,

time(disk response) = time(biodone) – time(previous biodone)

So the last four disk events would give disk response times of 10, 10, 10, 10 ms – and a total of 40 ms. That bit makes sense.

Now we must take into account if the previous biodone event was on a different disk entirely. The algorithm is better as,

time(disk response) = time(biodone) – time(previous biodone on the same disk device)

What about the first response time? This would be calculated correctly if the previous disk event conveniently finished at time = 0, but not if it finished some time before that. The next scenario will illustrate this clearly.

Sparse Disk Events

Consider two disk requests at time = 0 ms and time = 40 ms, each taking 10 ms to complete,

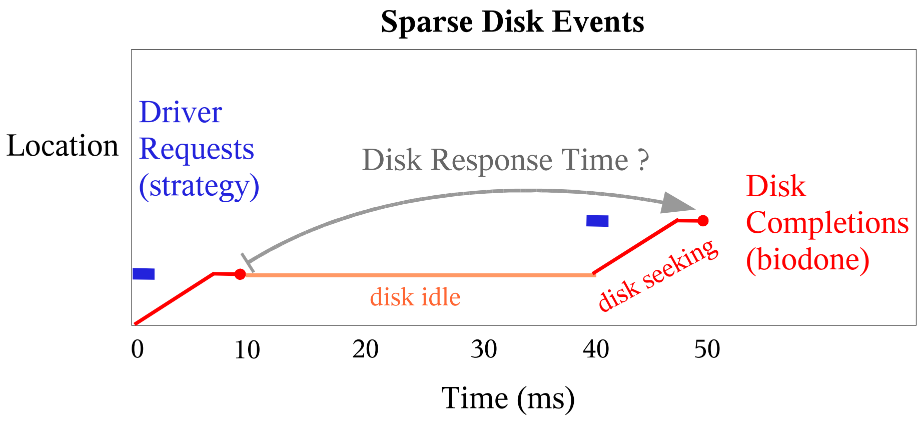

Figure 18 A problem with measuring times by consecutive biodones

The disk response time for the second request will be 50 ms – 10 ms = 40 ms. That's not right – the disk response time should be 10 ms, now we are counting idle time as disk response time.

A simple algorithm to take both concurrent and sparse events into account is,

```
time(disk response) = MIN(    time(biodone) - time(previous biodone, same disk device),
                              time(biodone) - time(previous strategy, same disk device)        )
```

Which, by deliberately choosing the minimum times from each approach, avoids over counting. This is depicted in Figure 19 below, and is the algorithm that the `psio` tool currently uses (version 0.68).

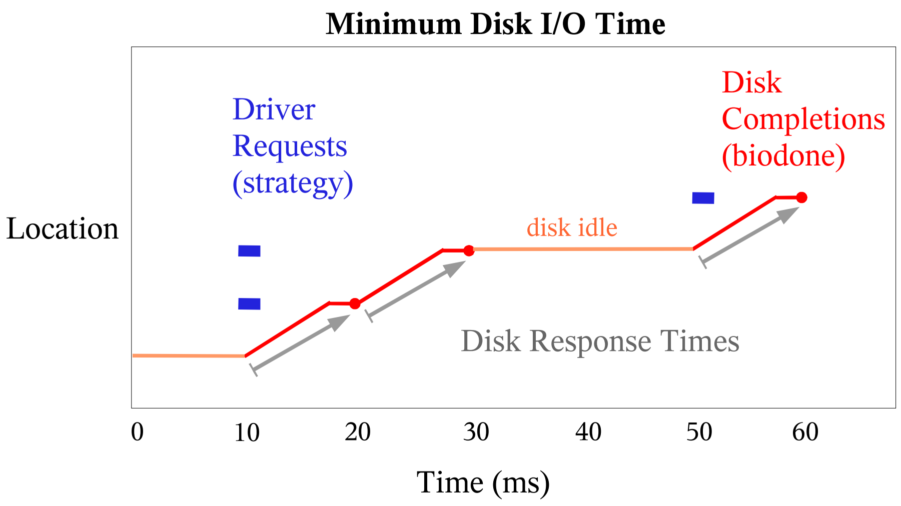

Figure 19 Measuring both concurrent and sparse events correctly

But not so fast! In Figure 19 two disk strategies cleanly occurred at time 10 ms, but what if they occurred at 10ms and 12ms, such that they were *mixed* and not in sync? By the minimum disk I/O time algorithm the first disk response time would be incorrectly reported as 8ms. While this minimum disk I/O time algorithm is simple to implement, it is possible that undercounts actual disk response time.

Before I get too carried away drawing more diagrams, let me cut to the chase with the final algorithm.

Adaptive Disk I/O Time Algorithm

To cover simple, concurrent, sparse and mixed events we'll need to be a bit more creative,

```
time(disk response) = MIN(   time(biodone) - time(previous biodone, same dev),
                             time(biodone) - time(previous idle -> strategy event, same dev) )
```

Tracking idle -> strategy events is achieved by counting pending events, and matching on a strategy event when pending == 0. Both “previous” times above refer to previous times on the same disk device.

This covers all scenarios, and is the algorithm currently used by the DTrace tools in the next section.

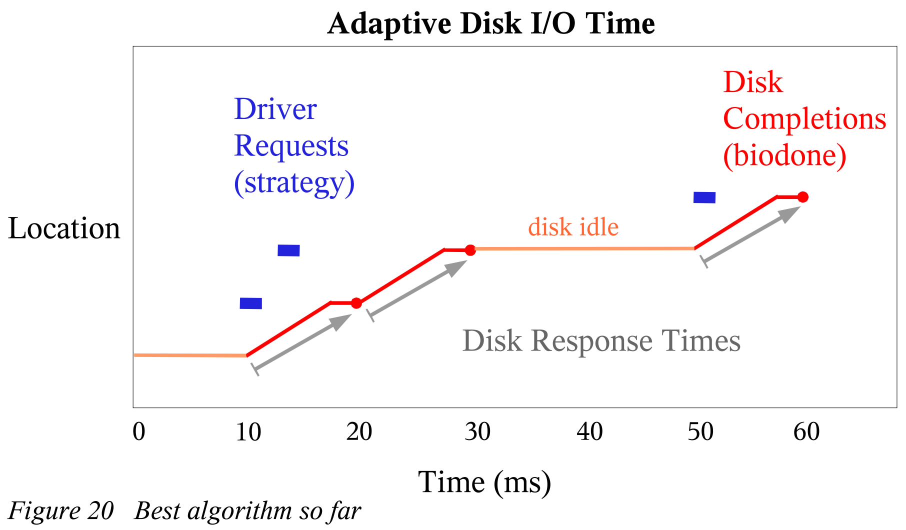

Figure 20 Best algorithm so far

In the above example, both concurrent and post-idle events are measured correctly.

Taking things too far

But ... what if, PIDs 101 and 102 both request nearby disk events at time = 0. They complete at 10 ms for PID 101 and 11 ms for PID 102. PID 101 will have 10 ms of I/O time, and PID 102 will have 1 ms. Is that fair? While the heads are seeking they are on their way to satisfy both requests – if this happens often in the exact same way you may be wondering why PID 102 is so fast, unable to realise that PID 101 is seeking the heads nearby. In this example, should both PIDs be actually be given a disk response time of 5.5 ms? Erm. Is this scenario even realistic? Or will such 'jitter' cancel out and be negligible?

If we keep throwing scenarios at our disk algorithm and are exceedingly lucky, we'll end up with an elegant algorithm to covers everything in an obvious way. However I think there is a greater chance that we'll end up with an overly complex beast-like algorithm, several contrived scenarios that still don't fit, and a pounding headache.

We'll consider the last algorithm presented as sufficient, so long as we remember that it is a close estimate.

Other Response Times

Thread Response Time is the time that the requesting thread experiences. This can be measured from the time that a read/write system call blocks to its completion, assuming the request made it to disk and wasn't cached[32]. This time includes other factors such as the time spent waiting on the run queue to be rescheduled, and the time spent checking the page cache – if used.

Application Response Time is the time for the application to respond to a client event, often transaction orientated. Such a response time helps us to understand why an application may respond slowly.

Time by Layer

The relationship between the response times is summarised in Figure 21, which serves as a visual reference for terminology used. This highlights different layers from which to consider response time.

Layers of Response Time

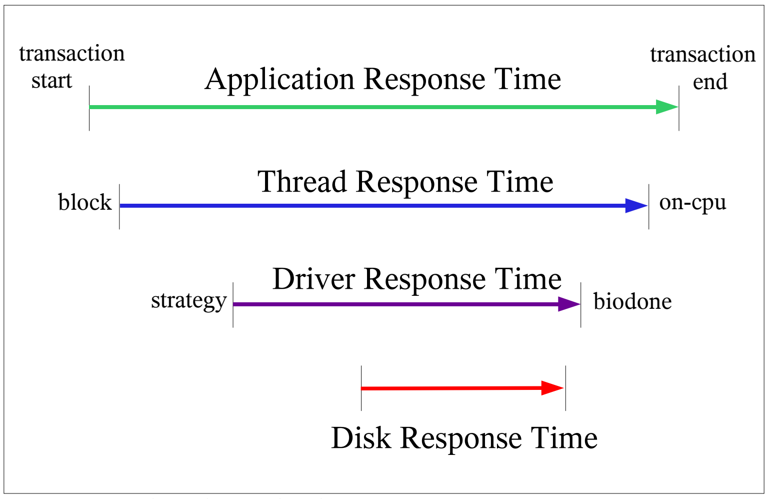

Figure 21 The relationship between response times

32 This is fairly difficult to determine from the system call details alone.

Completion is Asynchronous

From the previous algorithms discussed, the final disk I/O time by process may be calculated when the biodone event occurs. Lets look again at the TNF details available for this biodone event,

```
# tnfdump io01.tnf
[...]
         0.011287         0.011287   917     1 0xd590de00    0 strategy
        device: 26738689 block: 205888 size: 4096 buf: 0xdb58c2c0 flags: 34078801
        55.355558        55.344271     0     0 0xd41edde0    0 biodone
        device: 26738689 block: 205888 buf: 0xdb58c2c0
[...]
```
Figure 22 TNF details for the start and end events

The strategy event has a PID of 917, but the biodone event has a PID of 0. This is because the completion occurs asynchronously to the thread, or the process. Since we are interested in the process responsible, we need some way to associate the strategy with the biodone.

Traps

Don't trust the process on I/O completion.

When a thread executes a read system call to a file on disk, the thread is blocked and leaves the CPU until the read has completed. This gives other threads a chance to run during the 'slow' disk response time, and is a voluntary context switch.

When an I/O event completes, the thread that requested the I/O is placed back on the run queue where it can be scheduled to run.

But! At the moment the I/O event completes, the responsible thread is not on the CPU – it is on the sleep queue. This means we cannot measure the thread or process ID from the CPU on I/O completion.

Chapter 9 of Solaris Internals covers dispatcher operations such as context switching in detail.

Events are associated using the device number and the block address as a unique key. This assumes that we are unlikely to issue concurrent disk events to the same block number.

Analogy

Some people have become quite confused by the concept of completions being asynchronous to the process. Let me draw an analogy to make this point clear.

The scene is a popular coffee shop. The manager wonders why sometimes the staff seem saturated with requests, and a queue of customers is forming. Perhaps the problem is that one customer is ordering really weird coffees (containing numerous random ingredients) and they are taking a while to make. They could be identified if we knew *coffee response time by customer*.

Customers order a coffee, it is made, placed on the counter, then the customer picks it up. Measuring time between the customer ordering and picking it up may be unfair – they may have wandered away for some reason adding time that isn't coffee preparation related. We ought to measure the time from a coffee order to it being completed (strategy to coffeedone).

When an order is being placed, we know who the customer is – they are still standing there. When the coffee is completed, the customer may or may not still be standing there (they may be outside talking on the phone). To associate the two events *by customer*, we'd need to remember who ordered what. (Or we could just blame the half caf soy chai late with extra froth person!).

Understanding psio's values

IOTIME gives us a close estimate of the number of milliseconds of time the disk spent satisfying requests by process. That is quite a useful measurement, as I/O time does take into account both the size of the disk events and the seek time involved.

%I/O, the default output of `psio`, takes IOTIME and divides it by the period of the `psio` sample. By default this is one second, but can be changed on the command line – in Figure 14 this was made ten seconds. So with a period of one second, 65.4% I/O time means that process caused 654 ms of I/O time.

If a process accessed two different disks at 600 ms I/O time each, `psio` would add the I/O time values and report 120% I/O. Hmm. For a while I capped %I/O to 100 to avoid confusion, but I've now returned it to this uncapped value.

So %I/O represents a *single disk's utilisation by process.*

It is a useful measure of disk I/O utilisation by process, so long as we understand its origin and limitations.

food for thought

%I/O time, is a percentage of what?

I/O capacity across the entire server isn't as useful as it seems, not all disks may be useable by the application.

I/O for a single disk makes more sense. This means an application running at 200% is effectively using two disks at 100% each.

Asymmetric vs Symmetric Utilisation

If the vmstat tool reported 100% CPU utilisation, we know all CPUs are 100% utilised – which may well indicate a resource problem. If it reported the CPUs were 20% utilised we'd not worry about CPU usage. I'll describe this as a utilisation percentage for a *symmetric resource*, as CPU utilisation resembles a resource where work can be evenly distributed across each provider (CPU).[33]

Disk I/O utilisation is not a symmetric resource. An application may use a filesystem that contains two disks on a server which has ten disks. If those two disks become busy, work cannot be distributed to the other disks. This can be described as an *asymmetric resource*.

If disk I/O utilisation was reported in a same way as CPU utilisation, then two disks busy out of ten would report 20%. This may grossly understate the problem. The algorithm we do use would report 200%. Examples of these algorithms are depicted below,

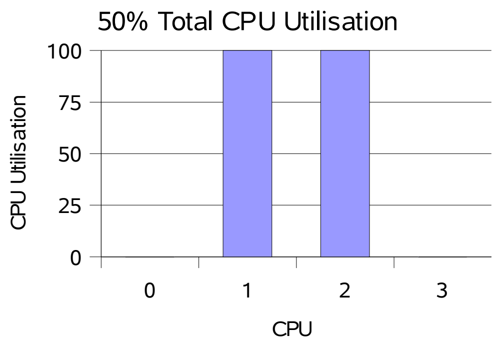

Figure 23 Symmetric resource utilisation %

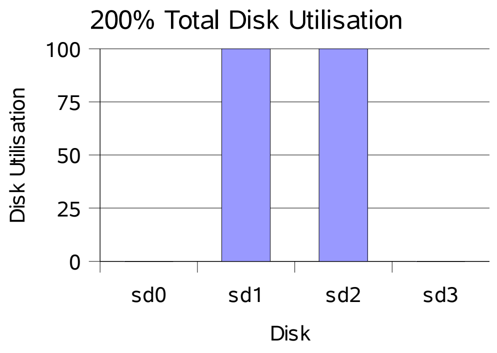

Figure 24 Asymmetric resource utilisation %

33 I'll discuss single-threaded/process issues in a paper on CPU utilisation by system.

A problem with our asymmetric resource utilisation percentage is that it may overestimate the problem rather than underestimate. It may be normal for a large database server to drive 16 disks at 50% utilisation each, which would report 800%. Although it may be argued that it is safer to overstate than understate.

Another problem when generating a %I/O value occurs when metadevices are used. We may overcount disk events if we see both block events to the metadevice and block events to the physical disks that it contains.

Is there a much better way to calculate resource utilisation percentages that is accurate for every scenario? Probably not. This is a trade-off incurred when simplifying a complex resource into a single value.

Bear in mind percentages over time

Another important point to mention is the duration for which we are measuring I/O utilisation. If we saw 10% utilisation it would seem to indicate light resource usage. If the sample was an hour it may have actually been 6 minutes of disk saturation (at 100% utilisation) and 54 minutes idle.

This problem applies to any percentage average over time, and is one of the compromises made when using a percentage rather than examining raw event details. This is all fine, so long as we bear it in mind.

Question

What sample duration should you use?

This depends on what collects the data and how it is used.

A generic system monitoring tool may use long intervals, such as every 5 minutes.

A system administrator troubleshooting a problem may use quick intervals such as every 5 seconds.

Duelling Banjos and Target Fixation

An interesting problem occurs if you focus on the disk utilisation value for your target process, while ignoring the utilisation values for other processes.

Consider you have an application where you believe it should be performing numerous sequential events, however the high disk utilisation value (which is time based) suggests that it is actually performing random I/O. To try and understand the utilisation value better, you dump the raw I/O event details, grep for your target process and eyeball the block address and size information for the strategy events. Still no answer! Your process appears to be calling a number of sequential events, but not enough to warrant the I/O time incurred that makes up the utilisation value.

Only when inspecting all events, not just those for your process, do you find an explanation: another process is also using the disk at the same time, and is seeking the heads *elsewhere* on disk before many of your target process events. The seek time to return the heads is counted in your target process's disk I/O time.

Tip

Don't grep your PID.

disk activity by other processes is important to consider too, they may be affecting the process you are examining.

So the worst case scenario is where two processes are taking turns to read from either end of the disk. This is called the “duelling banjos condition”[34]. Focusing on the I/O time of one process will be confusing without considering the other.

In summary, the disk utilisation value by process is influenced by the value of other processes.

34 it isn't really, I just made that up.

What %utilisation is bad?

We've previously used a value of 100% disk utilisation to indicate a resource usage problem. Is 80% a problem, or 60%? The following table has been provided as a guide,

Utilisation per disk	Performance problem
100%	probably
75%	maybe
50%	maybe
25%	maybe
0%	no

Figure 25 What %utilisation is bad

There is no simple rule for identifying disk throughput problems based on utilisation percents. It is a complex system that is dependant on your applications, how they use the disks, why they use the disks, and *what* it is that is "bad". If you already have an idea for what %utilisation is bad for *your* server based on experience, then stick with that.

There are, at least, two sides to the simple rule argument. Briefly,

for simple rules: Give me a simple rule to check whether I have a disk performance problem. No, I don't have time to learn the complexities of performance analysis. Just give me any rule as a starting point to get a handle on disk performance. No, I don't much care if the rule over simplifies things – anything is better than nothing.

against simple rules: There is no simple rule for disk performance. It is really dependant on your application, disk configuration and what constitutes a problem. Providing a simple rule may discourage proper analysis of other disk statistics, which may clearly point out the real problem.

Myth

Myth: Heavy disk I/O is always bad.

Fact: Sometimes doing a lot of disk I/O is the price of doing business.

You may study how the application uses the disks, how the disks are performing, how software caching is configured – only to discover that all is well. Sometimes it is unavoidable that an application does heavy disk I/O.

To provide a starting point[35], it would be fair to say that a sustained per disk utilisation of 60% or more over an interval of minutes rather than seconds is likely to reduce the performance of your application. Since disks are a slow resource (compared to memory, for example), any sustained disk utilisation over 20% is certainly interesting.

To answer what %utilisation is *bad*, we need to ask ourselves what "bad" actually means. Does it mean a client experiences a slow response time?

Once we have our %utilisation value, it doesn't tell us if that utilisation is "bad" or not. All it tells us are which processes are keeping the disks busy.

Tip

Measuring response times.

If your clients experience slow response times, consider using a program to simulate client activity at regular intervals, time the response, and write this info to a log. This data is great for pattern analysis.

Another technique to get such data is to instrument the application itself. There are many ways to do this, such as TNF tracing and DTrace.

35 from Chapter 21 of "Configuring & Tuning Databases on the Solaris Platform" which has a good reference of "what to look for" on a system. Also see "Configuration and Capacity Planning on Sun Solaris Servers".

The revenge of random I/O

Lastly, lets return to the random vs sequential disk I/O issue. We've made great effort to measure the time consumed by disk events, as this accurately reflects the extra time caused by random disk activity vs faster sequential events. So a value of 60% utilisation (by disk I/O time) is indeed meaningful, and does mean that there is another 40% of capacity available. Great.

But now we have a new problem to contend with: How do we know how much of this 60% disk utilisation measurement was random events, and how much was sequential?

What!? Didn't we just take that into account? Well, yes. We turned apples and oranges into a more generic "fruit" measurement by using a meaningful algorithm to take account of their appleyness or orangeyness. Okay. And now that we have such a meaningful value, we can't tell what the "fruit" originally was. Without eating it, of course.

Or, consider the scenario where you have a raft floating down an Amazonian river. Standing on the raft are a number of elephants and monkeys, as depicted in Figure 26.[36] To calculate how utilised the raft is, you take the combined weights of the elephants and monkeys and then divide by the raft's capacity. That may tell you the raft is 75% utilised. How many elephants and how many monkeys are there from this value?

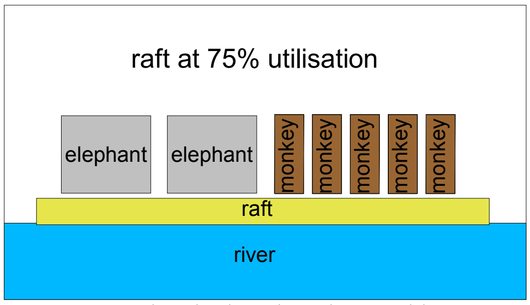

Figure 26 The elephant/monkey problem

This problem is another example of details lost when summarising into one value. If a process is driving the disks at 10% utilisation, it may actually be driving the disks in a clumsy random way, which if improved may take the utilisation down to 2%. The value of 10% is not final, there is more to learn about what events constituted a utilisation of 10%.

The point is, while our disk utilisation percent is a useful measurement **it is not the end of the story**.

For further reading on traps when creating performance metrics, read Chapter 26 from "Configuring & Tuning Databases on the Solaris Platform".

36 A pointless diagram.

3.1.5. More TNF tracing

To complete our discussion on TNF tracing, the following tools deserve a mention.

taz

An earlier tool to use TNF probes for disk I/O analysis is **taz**[37], the Tasmanian Devil Disk Tool by Richard McDougall. It is bundled as RMCtaz, which provides a text based tool called "`taz`" and a GUI based tool called "`taztool`". `taz` prints disk activity with details such as block address, service time and seek distance. `taztool` plots disk activity by block address and time.

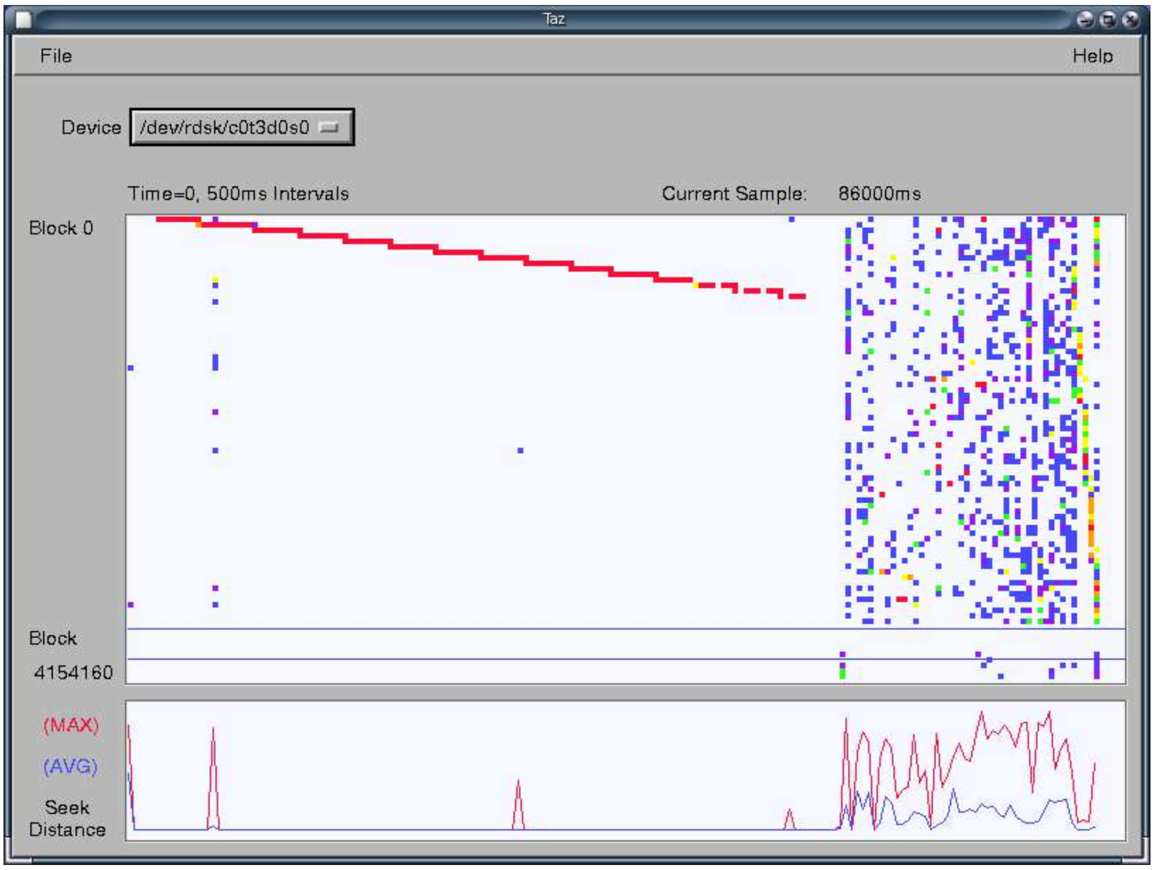

Figure 27 taztool plotting sequential then random disk I/O

A screenshot of `taztool` is in Figure 27.[38] While `taztool` was running, sequential disk activity was caused using the `dd` command (as was done in Figure 9), and then random disk activity was caused using the `find` command (as with Figure 10).

In the upper plot, `dd`'s sequential disk activity is drawn as a heavy red line beginning at block 0 and then seeking gradually across the disk. In the last third of the plot we can see `find`'s random disk activity, shown as scattered block addresses of cooler colours (indicating smaller sizes). The way `taztool` has visualised this disk behaviour is very effective.

The lower plot indicates seek distance, with blue the average and red the maximum. This plot begins dead flat for sequential activity then becomes mountainous for random activity, as expected.

37 http://www.solarisinternals.com/si/tools/taz/index.php
38 this was RMCtaz ver 1.1 on a Solaris 8 server with a 32 bit kernel (ver 1.1 needs a 32 bit kernel).

TNFView

This is a GUI tool to plot TNF data by time and by thread. It is part of the TNF Tracing Tools collection, downloadable from the Sun Developers website[39]. It is really cute and helps trawl through TNF data quickly, in a variety of creative ways.

TNF tracing dangers

`psio` was written as a demonstration tool[40]. Since it enabled kernel tracing, something that is not commonly used, I considered the possibility that this could trigger an undiscovered bug in kernel code or cause an unwanted conflict. I warned against running `psio` in production until I had studied the implications of TNF tracing more thoroughly[41].

It turns out that TNF tracing is fairly safe, there are no known bugs as of July 2005. Probably the worst problem is the kernel ring buffer that TNF uses. You must provide a size – but what size? The `psio` tool uses 300 Kb, plus an extra 100 Kb per second of the interval – but that's really just a guess. For very busy servers that may not be enough, and the ring buffer may silently drop packets. A "-b" option was provided with `psio` so a larger size could be picked from the command line.

DTrace from Solaris 10 is similar to TNF tracing as probes can be activated that write data to a buffer, and we can match on the same strategy and biodone probes with process details. However there are thousands more probes to pick from, customisable actions to run for each event, and the buffer size problem has been solved.

Buried Treasure

TNF tracing is still quite useful for Solaris 9 and earlier, before DTrace was available. Developers can add trace points into their production code so that performance statistics can be enabled and fetched as needed.

However few developers have been near TNF tracing. It seems an undiscovered treasure from older Solaris, that has now been surpassed by DTrace.

39 http://developers.sun.com/solaris/developer/support/driver/tools/tnftl.html. There is also an introduction to TNFView at http://developers.sun.com/solaris/articles/tnf.html, by John Murayamo.
40 after I read Sun Performance and Tuning and saw the opportunity to present TNF data in a more meaningful way
41 Many emails of thanks, none to say "psio crashed my server!"

3.1.6. DTrace

DTrace is a tool added to Solaris 10 that allows users to write their own troubleshooting or performance analysis scripts in a comfy C-like language. If you are new to DTrace, the following should serve as an introduction.

DTrace is the Holy Grail of tracing tools, and is arguably one of the greatest achievements in operating systems for over a decade. Some of its capabilities are similar to existing tools: such as tracking syscalls with `truss`, library calls with `apptrace`, user functions with `truss -ua.out`, and navigating both kernel and user virtual memory with `mdb`. However DTrace goes further with first of its kind features, such as dynamically tracing all kernel functions, and being lightweight and safe to use – unlike `truss`[42].

For analysing disk utilisation by process, DTrace extends what we were achieving with TNF tracing. With DTrace it is more powerful, more reliable and safer to use.

fbt probes

Now, if I were *really* lazy, I could use DTrace to pull the same TNF probes out. The following lists them,

```
# dtrace -ln 'fbt::*tnf_probe:entry'
   ID   PROVIDER            MODULE                          FUNCTION NAME
 3461        fbt           genunix                 biodone_tnf_probe entry
 3988        fbt           genunix           bdev_strategy_tnf_probe entry
```

Figure 28   The TNF probes from DTrace

These probes give me access to the same data I was reading from `tnfdump`. (TNF called the bdev_strategy function just “strategy”).

We begin to tap the real power of DTrace when we access the kernel functions for bdev_strategy and biodone themselves, including access to all the input arguments and return values,

```
# dtrace -ln 'fbt::bdev_strategy:,fbt::biodone:'
   ID   PROVIDER            MODULE                          FUNCTION NAME
 8422        fbt           genunix                     bdev_strategy entry
 8423        fbt           genunix                     bdev_strategy return
11184        fbt           genunix                           biodone entry
11185        fbt           genunix                           biodone return
```

Figure 29   Phwaorrr, these are the real strategy and biodone functions

The above is a list of probes. A probe traces a single event and has a four components to its name: provider, module, function, name. The provider could be described as a library of related probes, here we are looking at fbt, function boundary tracing, a raw provider of kernel functions. The “FUNCTION” for the probes is the kernel function name, such as bdev_strategy. The last component, “NAME”, provides a probe for the entry to the function and the return – this lets us fetch both the input arguments and the return value.

42 See http://www.brendangregg.com/DTrace/dtracevstruss.html for a showdown of DTrace vs truss.

The following traces the probes live,

```
# dtrace -n 'fbt::bdev_strategy:,fbt::biodone:'
dtrace: description 'fbt::bdev_strategy:,fbt::biodone:' matched 4 probes
CPU     ID                    FUNCTION:NAME
  0   8422             bdev_strategy:entry
  0   8423            bdev_strategy:return
  0   8422             bdev_strategy:entry
  0   8423            bdev_strategy:return
[...]
```

Figure 30 Tracing live probes

DTrace gives us access to the arguments of functions, both bdev_strategy and biodone have one argument - a pointer to the buf struct for this disk I/O event. From here we can walk through some kernel structures to fetch plenty of information about this disk event, such as vnode, inode and vfs pointer, although it is some work to do so. Fortunately DTrace has a provider that does the walking[43] for you, "io".

io probes

The io provider allows us to trace disk events with ease. It provides "io:::start" and "io:::done" probes, which for disk events corresponds to the strategy and biodone probes previously used.

```
# dtrace -lP io
   ID   PROVIDER            MODULE                          FUNCTION NAME
 1425         io               nfs                          nfs4_bio done
 1426         io               nfs                          nfs3_bio done
 1427         io               nfs                           nfs_bio done
 1428         io               nfs                          nfs4_bio start
 1429         io               nfs                          nfs3_bio start
 1430         io               nfs                           nfs_bio start
 9915         io           genunix                           biodone done
 9916         io           genunix                           biowait wait-done
 9917         io           genunix                           biowait wait-start
 9926         io           genunix                     default_physio start
 9927         io           genunix                      bdev_strategy start
 9928         io           genunix                           aphysio start
```

Figure 31 Probes available from the io provider

In Figure 31 we list the probes from the io provider. This provider also tracks NFS events, raw disk I/O events and asynchronous disk I/O events.

The probes io::biowait:wait-start and io::biowait:wait-done track when the thread begins to wait, and when the wait has completed. This could be used to study *thread response time*, if needed.[44]

43 See /usr/lib/dtrace/io.d for details
44 This is really useful, by the way.

Details about each I/O event are provided by three arguments to these io probes. Their DTrace variable names and contents are[45],

- args[0], bufinfo. Useful details from the buf struct.
- args[1], devinfo. Details about the device: major and minor numbers, instance name, ...
- args[2], fileinfo. Details about the filename, pathname, filesystem, offset ...

These contain all of the desired details. Sheer luxury.

I/O size one liner

Fetching I/O event details with DTrace is very easy. The following command tracks PID, process name, I/O event size and is a one liner,

```
# dtrace -n 'io:::start { printf("%d %s %d",pid,execname,args[0]->b_bcount); }'
dtrace: description 'io:::start ' matched 6 probes
CPU     ID                    FUNCTION:NAME
  0   9927              bdev_strategy:start 1122 grep 16384
  0   9927              bdev_strategy:start 1122 grep 57344
  0   9927              bdev_strategy:start 1122 grep 16384
  0   9927              bdev_strategy:start 1122 grep 57344
  0   9927              bdev_strategy:start 1122 grep 40960
  0   9927              bdev_strategy:start 1122 grep 57344
  0   9927              bdev_strategy:start 1122 grep 57344
  0   9927              bdev_strategy:start 1122 grep 8192
[...]
```

Figure 32   One liner for I/O size by process

This assumes that the correct PID is on the CPU for the start of an I/O event, which is fine.

For disk response time we'll need to measure the time at the start and the end of each event. As was done by the `psio` tool, we associate the end event to the start event using the extended device number and the block address as the unique key.

I've written two programs that use DTrace to calculate disk response time, `iotop` and `iosnoop`.

45 full documentation is in the "io" chapter in the DTrace Guide, http://docs.sun.com/db/doc/817-6223

iotop

iotop[46] is a freeware program that uses DTrace to print disk I/O summaries by process, for details such as size (bytes) and disk I/O time. The following is the default output of version 0.75, which prints size summaries and refreshes the screen every five seconds,

```
# iotop
2005 Oct 24 23:52:41,  load: 0.07,  disk_r:  17460 Kb,  disk_w:     20 Kb

  UID    PID   PPID CMD              DEVICE  MAJ MIN D            BYTES
    0   2673   2672 locale           cmdk0   102   0 R             1024
    0   2674   2671 sshd             cmdk0   102   0 R             4096
    0   2675   2674 sh               cmdk0   102   0 R             4096
    0      3      0 fsflush          cmdk0   102   0 W             8192
    0   2666   2663 sshd             cmdk0   102   0 R             8192
    0   2671    378 sshd             cmdk0   102   0 W            12288
    0   2671    378 sshd             cmdk0   102   0 R            12288
    1    116      1 kcfd             cmdk0   102   0 R           131072
    0   2669   2668 find             cmdk0   102   0 R           356352
    0   2668   2667 bart             cmdk0   102   0 R         17156096
```

Figure 33   Default output of iotop

In the above output, the bart process was responsible for around 17 Mb of disk read. Disk I/O time summaries can also be printed "-o", which is a more accurate measure of disk utilisation. Here we demonstrate this with an interval of 10 seconds,

```
# iotop -o 10
2005 Oct 24 19:15:02,  load: 0.05,  disk_r:  11553 Kb,  disk_w:     12 Kb

  UID    PID   PPID CMD              DEVICE  MAJ MIN D         DISKTIME
    0   2070    378 sshd             cmdk0   102   0 W              489
    0   2078   2077 sh               cmdk0   102   0 R             8701
    0   2079   2078 sh               cmdk0   102   0 R            15728
    0   2065      1 nscd             cmdk0   102   0 R            22912
    0   2077   2076 sshd             cmdk0   102   0 R            26900
    0   2080   2078 sort             cmdk0   102   0 R            32558
    0   2070    378 sshd             cmdk0   102   0 R           218454
    0   2079   2078 find             cmdk0   102   0 R           673775
    0   2078   2077 bart             cmdk0   102   0 R          1657506
```

Figure 34   Measing disk I/O time by process

Note that iotop is printing totals, not per second values. In Figure 34 we read 11.5 Mb from disk during those 10 seconds (disk_r), with the top process "bart" (PID 2078) consuming 1.66 seconds of disk time. For this 10 second interval, 1.66 seconds would equate to a utilisation value of 17%.

46 It is available in the DTraceToolkit, http://www.opensolaris.org/os/commmunty/dtrace/dtracetoolkit

`iotop` can print %I/O utilisation using the "`-P`" option, here we also demonstrate "`-C`" to prevent the screen from being cleared and provide a rolling output instead,

```
# iotop -CP 1
2005 Oct 24 23:46:06,  load: 0.30,  disk_r:    324 Kb,  disk_w:      0 Kb

  UID    PID   PPID CMD                DEVICE  MAJ MIN D   %I/O
    0   2631    942 bart               cmdk0   102   0 R     44
    0   2630    942 find               cmdk0   102   0 R     49

2005 Oct 24 23:46:07,  load: 0.30,  disk_r:    547 Kb,  disk_w:      0 Kb

  UID    PID   PPID CMD                DEVICE  MAJ MIN D   %I/O
    0   2630    942 find               cmdk0   102   0 R     44
    0   2631    942 bart               cmdk0   102   0 R     50

2005 Oct 24 23:46:08,  load: 0.31,  disk_r:    451 Kb,  disk_w:      0 Kb

  UID    PID   PPID CMD                DEVICE  MAJ MIN D   %I/O
    0   2630    942 find               cmdk0   102   0 R     43
    0   2631    942 bart               cmdk0   102   0 R     48
[...]
```

Figure 35   Viewing %disk I/O

In the above output we can see the `find` and `bart` processes jostling for disk I/O. The command executed was "`find /var | bart create -I`", which outputs a database containing checksums for every file in /var. This causes heavy disk activity, as `find` churns through the numerous directories in /var and `bart` reads the file contents.

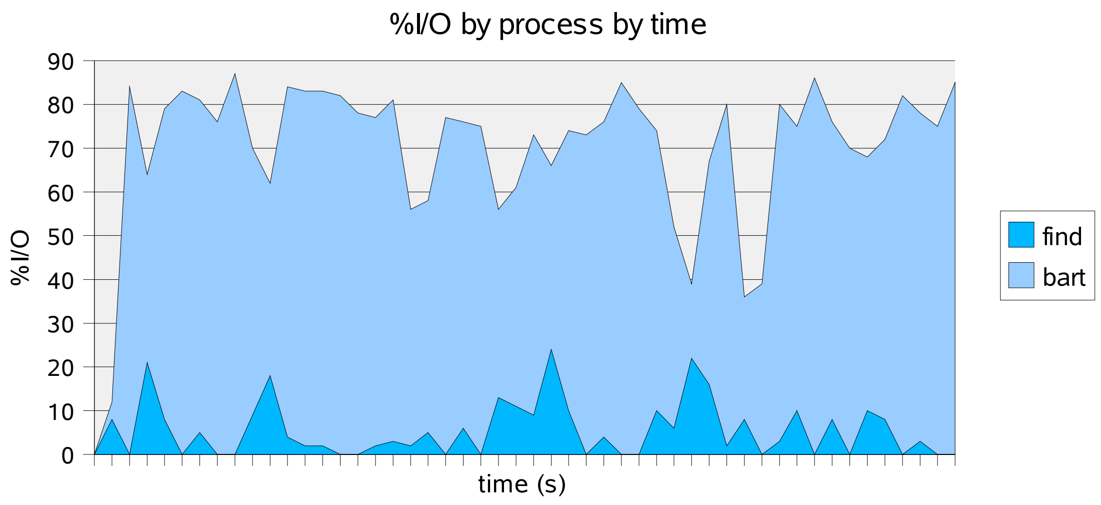

Figure 36   %I/O as find and bart read through /usr

Figure 36 plots %I/O as `find` and `bart` read through /usr. This time `bart` causes heavier %I/O as there are bigger files to read, and fewer directories for `find` to traverse.

Lets check if iotop's %I/O passes a sanity check; we compare an `iotop` output with an `iostat` output for the same interval while running a `tar` command to create test load,

```
# iotop -CP 10 1
2005 Oct 24 23:39:55,  load: 0.14,  disk_r:  86681 Kb,  disk_w:    214 Kb

  UID    PID   PPID CMD              DEVICE  MAJ MIN D   %I/O
    0      3      0 fsflush          cmdk0   102   0 W      0
    0      0      0 sched            cmdk0   102   0 W      0
    0    942    938 bash             cmdk0   102   0 R      0
    0   2593    942 tar              cmdk0   102   0 R     56
```

Figure 37   iotop output for test load

```
# iostat -xnz 10 2
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
    2.4    0.4   26.1    1.2  0.0  0.0    1.3    1.7   0   0 c0d0
    0.0    0.0    0.0    0.0  0.0  0.0    0.2   48.1   0   0 c1t0d0
                    extended device statistics
    r/s    w/s   kr/s   kw/s wait actv wsvc_t asvc_t  %w  %b device
  313.8    5.2 8685.5   21.4  0.1  0.9    0.3    2.7   6  56 c0d0
```

Figure 38   iostat output for test load

The %I/O values compare well between the `iostat` and `iotop` outputs.

Other options in iotop can be listed using "`-h`" (remember this is version 0.75),

```
# iotop -h
USAGE: iotop [-C] [-D|-o|-P] [-j|-Z] [-d device] [-f filename]
             [-m mount_point] [-t top] [interval [count]]

                -C      # don't clear the screen
                -D      # print delta times, elapsed, us
                -j      # print project ID
                -o      # print disk delta times, us
                -P      # print %I/O (disk delta times)
                -Z      # print zone ID
                -d device       # instance name to snoop
                -f filename     # snoop this file only
                -m mount_point  # this FS only
                -t top          # print top number only
   eg,
        iotop         # default output, 5 second samples
[...]
```

Figure 39   Listing iotop's options

iosnoop

iosnoop is a freeware program that uses DTrace to monitor disk events live[47]. The default output prints straightforward details such as PID, block address and size,

```
# iosnoop
  UID   PID D    BLOCK   SIZE       COMM PATHNAME
    0  1570 R   172636   2048       grep /etc/default/autofs
    0  1570 R   102578   1024       grep /etc/default/cron
    0  1570 R   102580   1024       grep /etc/default/devfsadm
    0  1570 R   108310   4096       grep /etc/default/dhcpagent
    0  1570 R   102582   1024       grep /etc/default/fs
    0  1570 R   169070   1024       grep /etc/default/ftp
    0  1570 R   108322   2048       grep /etc/default/inetinit
    0  1570 R   108318   1024       grep /etc/default/ipsec
    0  1570 R   102584   2048       grep /etc/default/kbd
    0  1570 R   102588   1024       grep /etc/default/keyserv
    0  1570 R   973440   8192       grep /etc/default/lu
    0  1570 R   973456   8192       grep /etc/default/lu
[...]
```

Figure 40   Default output of iosnoop

In the above output, we can see a grep process is reading several files from the /etc/default directory.

Options allow us to dig deeper. Here we use "-e" for the device name (DEVICE), and "-o" for the disk response time (DTIME) which uses the adaptive disk I/O time algorithm previously discussed,

```
# iosnoop -eo
DEVICE  DTIME         UID   PID D    BLOCK   SIZE       COMM PATHNAME
cmdk0   176             0  1604 R   103648   1024         ls /etc/volcopy
cmdk0   172             0  1604 R   103664   1024         ls /etc/wall
cmdk0   189             0  1604 R   103680   1024         ls /etc/whodo
cmdk0   9705            0  1604 R   171246   1024         ls /etc/rmt
cmdk0   464             0  1604 R   171342   1024         ls /etc/aliases
cmdk0   3929            0  1604 R   389290   1024         ls /etc/chroot
cmdk0   7631            0  1604 R   342798   1024         ls /etc/fuser
cmdk0   172             0  1604 R   342830   1024         ls /etc/link
cmdk0   169             0  1604 R   342862   1024         ls /etc/mvdir
[...]
```

Figure 41   iosnoop with device name and disk times

In Figure 41 the disk response time, "DTIME", is printed in microseconds. The largest event, 9705 us or 9.705 ms, corresponds to a jump in block address where the disk heads would have seeked. The sequential disk events have a much smaller time, around 170 us or 0.17 ms.

47 It is also in the DTraceToolkit. iosnoop was my first released tool using DTrace, written during the Solaris 10 beta program before the io provider existed.

By default iosnoop provides a PID column, the "-o" gives the "DTIME" column needed to see disk I/O time by process. This is disk utilisation information by process, this verbose event style output that iosnoop provides is suitable for when more details about disk I/O are needed.

A list of available options for iosnoop can be fetched using "-h". This is from iosnoop version 1.55,

```
# iosnoop -h
USAGE: iosnoop [-a|-A|-DeghiNostv] [-d device] [-f filename]
               [-m mount_point] [-n name] [-p PID]
       iosnoop          # default output
                -a      # print all data (mostly)
                -A      # dump all data, space delimited
                -D      # print time delta, us (elapsed)
                -e      # print device name
                -g      # print command arguments
                -i      # print device instance
                -N      # print major and minor numbers
                -o      # print disk delta time, us
                -s      # print start time, us
                -t      # print completion time, us
                -v      # print completion time, string
                -d device       # instance name to snoop
                -f filename     # snoop this file only
                -m mount_point  # this FS only
                -n name         # this process name only
                -p PID          # this PID only
   eg,
        iosnoop -v      # human readable timestamps
        iosnoop -N      # print major and minor numbers
        iosnoop -m /    # snoop events on filesystem / only
```

Figure 42 iosnoop available options

Of particular interest is the major and minor numbers option, "-N". The block addresses printed are relative to the disk slice, so understanding them accurately requires disk slice details as well. Otherwise what may appear to be similar block addresses may in fact be on different slices or disks.

Favourite

I like using iosnoop as it provides raw data that isn't heavily processed and summarised, as with iotop.

Often with iotop I find myself asking – why is that utilisation so high? what is that process doing? etc.

When looking at iosnoop outputs you already have the answers. Although, for heavy disk activity the iosnoop output can scroll *very* fast.

Plotting disk activity

Using the "`-t`" option for `iosnoop` prints the disk *completion* time in microseconds[48]. In combination with "`-N`", we can plot disk events by a process on one slice. Here we fetch the data for the `find` command,

```
# iosnoop -tN
TIME           MAJ MIN   UID   PID D    BLOCK   SIZE       COMM PATHNAME
131471871356   102   0     0  1693 R   201500   1024       find /usr/dt
131471877355   102   0     0  1693 R   198608   8192       find <none>
131471879231   102   0     0  1693 R   198640   8192       find <none>
131471879788   102   0     0  1693 R   198656   8192       find <none>
131471880831   102   0     0  1693 R   198672   8192       find <none>
```

Figure 43 Disk I/O events with completion times

which contains the time (printed above in microseconds since boot) and block address. These will be our X and Y coordinates. We check we remain on the same slice (major and minor numbers) and then plot it,

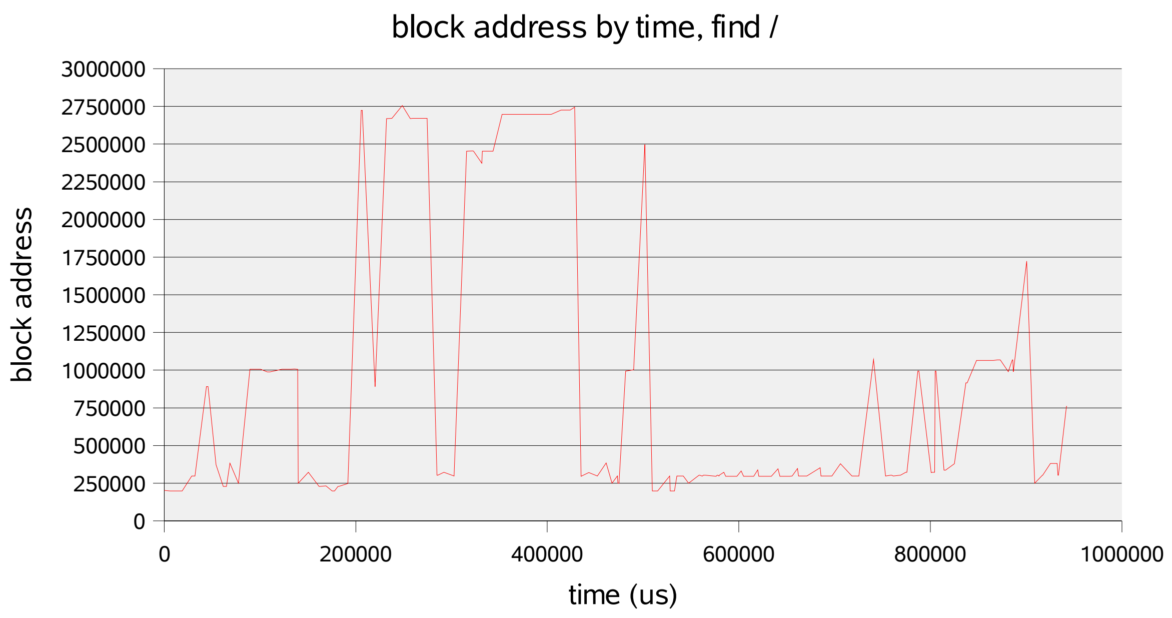

Figure 44 Plotting disk activity, a random I/O example

A "`find /`" command was run to generate **random** disk activity, which can be seen in Figure 44. As the disk heads seeked to different block addresses the position of the heads is plotted in red.

Woah now, are we *really* looking at disk head seek patterns? We are looking at block addresses for biodone functions in the block I/O *driver*. We aren't using some X-ray vision to look at the heads themselves.

Now, if this is a simple disk device then the block address *probably* relates to disk head location[49]. But if this is a virtual device, say a storage array, then block addresses could map to anything, depending on the storage layout. However we could at least say that a large jump in block address *probably* means a seek. But not so fast – storage arrays often have large front end caches, so while block I/O driver *thinks* that the event has completed it may have just been cached on the array.

48 if you are on a multi-CPU server, it's a good idea to sort on this field afterwards.

49 even "simple" disks these days map addresses in firmware to an internal optimised layout, all we know is the image of the disk that its firmware presents. The classic example here is "sector zoning".

The block addresses do help us visualise the pattern of *completed disk activity*. So long as we know that *completed* means the block I/O driver *thinks* they completed. For simple disks that is probably the case, for complex devices we must remember that disk events are remapped and may also be cached by the device.

Now for a demonstration of **sequential** disk I/O. A `dd` command is used on a raw disk device to deliberately read sequential blocks,

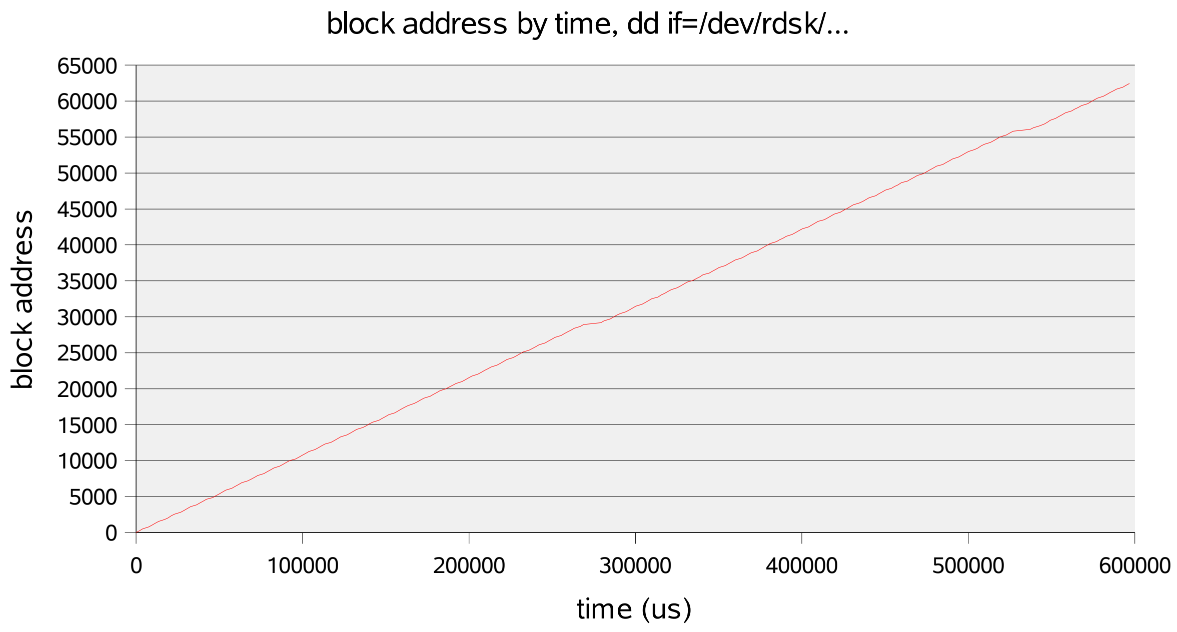

Figure 45 Plotting disk activity, sequential access of a raw device

Which is clearly sequential activity (and somewhat boring too).

What may be slightly more interesting is to run `dd` on the block device instead. A block device is a buffered device, so some areas are found in the cache,

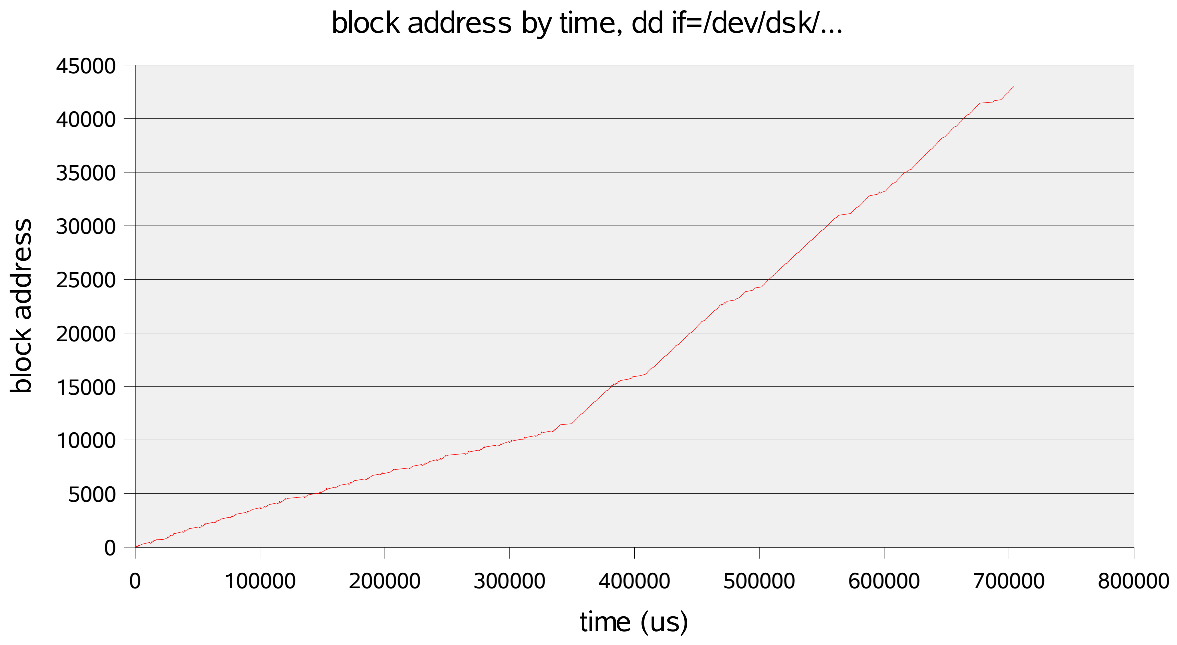

Figure 46 Plotting disk activity, sequential access of a block device

The steeper parts are caused by fewer disk reads as data is found in the page cache.

Plotting Concurrent Activity

Previously we discussed concurrent disk activity and included a plot (Figure 17) to help us understand how these events may occur. Since DTrace can easily trace this, it's irresistible to include a plot of *actual* activity.

The following DTrace script was written to provide input for a spreadsheet. We match on a device by its major and minor numbers, then print out timestamps as the first column and block addresses for strategy and biodone events in the remaining columns.

```
#!/usr/sbin/dtrace -s

#pragma D option quiet

io:genunix::start
/args[1]->dev_major == 102 && args[1]->dev_minor == 0/
{
        printf("%d,%d,\n", timestamp/1000, args[0]->b_blkno);
}

io:genunix::done
/args[1]->dev_major == 102 && args[1]->dev_minor == 0/
{
        printf("%d,,%d\n", timestamp/1000, args[0]->b_blkno);
}
```

Figure 47 Capture raw driver event data for plotting

The output of the DTrace script in Figure 47 was plotted as Figure 48, using timestamps as X coordinates.

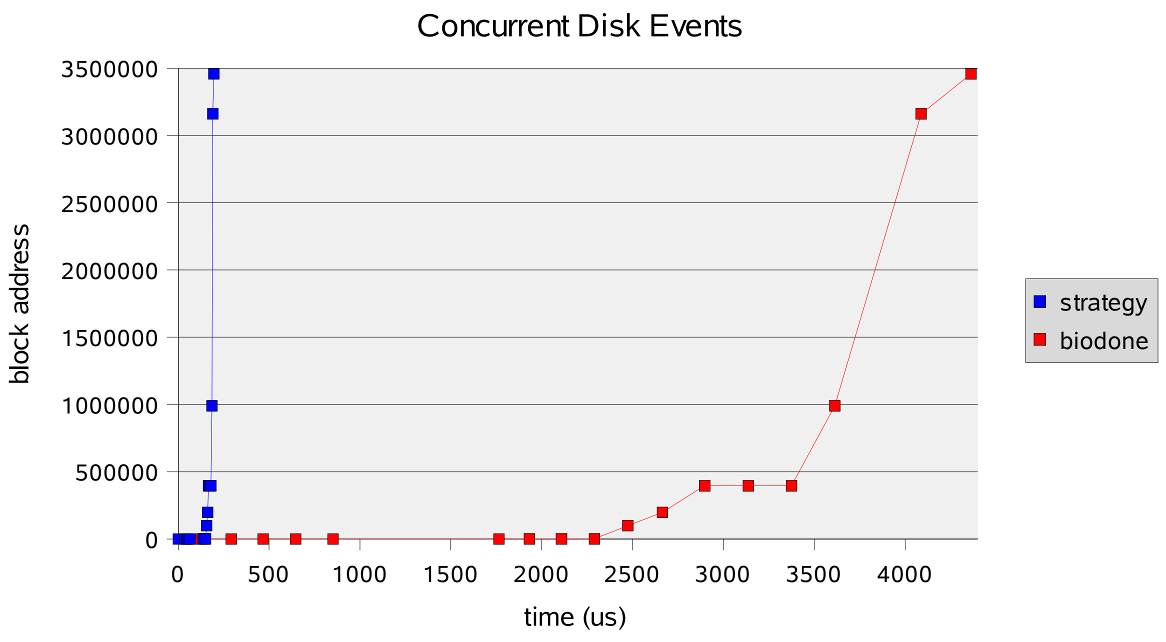

Figure 48 Plotting raw driver events: strategy and biodone

We can see many quick strategies followed by slower biodones, as the disk catches up at mechanical speeds.

Other DTrace tools

Since we are on the topic of disk I/O by process, there are other tools from the **DTraceToolkit** that provide disk I/O details. These details aren't utilisation values, but they are useful anyway. A tour for many of these tools was recently published as a feature article in Sys Admin magazine[50]. I'll mention a couple here.

`bitesize.d` is a very simple DTrace program[51] that prints a distribution plot of the *size* of disk events by process. This helps us understand if a process is reading the disk by taking large “bites” or small ones.

```
# bitesize.d
Sampling... Hit Ctrl-C to end.
^C

     PID  CMD
    2705  find /\0

           value  ------------- Distribution ------------- count
             512 |                                         0
            1024 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 459
            2048 |                                         0

    2706  bart create -I\0

           value  ------------- Distribution ------------- count
             512 |                                         0
            1024 |@@@@@@                                   443
            2048 |@@                                       155
            4096 |@@@                                      217
            8192 |@                                        51
           16384 |                                         32
           32768 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@             2123
           65536 |                                         0
```

Figure 49   bitesize.d example, disk I/O size distributions

In Figure 49 we can see the `find` command has made 459 disk events, all of which fell into the 1024 byte bucket (1024 to 2047 bytes). This is due to the `find` command reading through many small directory files.

The `bart` process has a much more interesting distribution, since it is reading the file contents it can issue much larger I/O requests for large files. We can see that most of its events were 32 Kb to 63 Kb.

If an application must go to disk, we generally like to see larger disk events rather than smaller. Larger is often an indication of sequential access, or read ahead access, both of which help performance. Smaller events can be an indication of scattered or random I/O access.

50 “Observing I/O Behavior with the DTraceToolkit”, Ryan Matteson, December 2005 issue. This has also been available online at http://www.samag.com/documents/sam0512a .

51 as a one liner: dtrace -n 'io:::start { @size[execname] = quantize(args[0]->b_bcount); }'

seeksize.d provides distributions of the *seek distance* of disk events by process. To demonstrate this script, a "`find / | bart create -I`" command is executed,

```
# seeksize.d
Sampling... Hit Ctrl-C to end.
^C

     PID  CMD
    2912  find /\0

           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@                                  11
               1 |                                         0
               2 |                                         0
               4 |                                         0
               8 |@                                        2
              16 |@                                        1
              32 |                                         0
              64 |@                                        1
             128 |@                                        2
             256 |@@                                       3
             512 |@@                                       3
            1024 |                                         0
            2048 |@@                                       3
            4096 |@                                        1
            8192 |@                                        1
           16384 |@@@@@@@                                  10
           32768 |@@@@@@                                   9
           65536 |@@@@                                     6
          131072 |@@                                       4
          262144 |@                                        1
          524288 |@@@@                                     5
         1048576 |@                                        1
         2097152 |@@@@                                     6
         4194304 |                                         0
[...]
```

Figure 50   seeksize.d example, find's random activity

In Figure 50 we have measured the activity of a `find` process. The value here is the seek size in units of sectors, and the `find` command has seeked over a wide number of different distances.

Also in the `seeksize.d` output was the activity by the `bart` process,

```
[...]
    2913  bart create -I\0

           value  ------------- Distribution ------------- count
              -1 |                                         0
               0 |@@@@@@@@@@@@@@@@@@@@@@@@@@               583
               1 |                                         0
               2 |                                         4
               4 |@                                        13
               8 |@                                        24
              16 |@                                        26
              32 |@                                        28
              64 |                                         4
             128 |@                                        17
             256 |@                                        26
             512 |@                                        32
            1024 |@                                        23
            2048 |@                                        14
            4096 |                                         6
            8192 |                                         1
           16384 |@@                                       44
           32768 |@@                                       50
           65536 |                                         7
          131072 |                                         5
          262144 |                                         1
          524288 |                                         6
         1048576 |                                         0
```

Figure 51   seeksize.d example, bart's sequential activity

Over half of the disk events requested for the `bart` command incurred a seek size of 0 – which is sequential disk activity. There was also some degree of seeking.

Other disk I/O tools in the DTraceToolkit are either in the top directory (for the most popular ones), or in the “Disk” subdirectory (for the rest).

4. Conclusion

Utilisation by process is best measured in terms of disk I/O time, I/O size is at best an approximation.

An adaptive disk I/O time algorithm was presented to best measure the time consumed by the disk device to satisfy the request. This disk response time can be calculated as an absolute value, such as in milliseconds, or as a percentage. Due to the asymmetric nature of disk resources, the disk I/O percentage was calculated in terms of a single disk – not a percentage of the disk capacity for the entire server.

There are no by process statistics to track disk I/O time, such as in procfs. Disk I/O time can be measured event wise using TNF tracing in Solaris 9 and earlier, and by using DTrace in Solaris 10 onwards. DTrace has allowed new tools such as `iosnoop` and `iotop` to be written, which provide both disk I/O time and size information by process. `iotop` can print a percent disk utilisation by process, which best answers the goal of this paper.

Presenting disk I/O time by process as a single value has lost some details about the original I/O events, especially whether they were random or sequential events. This is a trade-off incurred when simplifying a complex system down to a single value, in this case a percent disk utilised by process. This is fine as long as we bear in mind what our disk utilisation measurement really is – a handy summary. If needed, deeper details can then be fetched using tools such as the `iosnoop` program.

5. References

1. Cockcroft, A., *Sun Performance and Tuning – Java and the Internet*, 2nd Edition, Prentice Hall, 1998.
2. Cockcroft, A., *Clarifying disk measurements and terminology*, SunWorld Online, September 1997.
3. Cockcroft, A., *Solving the iostat disk mystery*, SunWorld Online, October 1998.
4. Gregg, B. D., *DTrace Tools*, http://www.brendangregg.com/dtrace.html.
5. Matteson, R., *Observing I/O Behavior with the DTraceToolkit*, Sys Admin Magazine, December 2005.
6. Mauro, J., McDougall, R., *Solaris Internals*, Prentice Hall, 2001.
7. OpenSolaris, *DTrace Community*, http://www.opensolaris.org/os/community/dtrace.
8. Packer, A. N., *Configuring & Tuning Databases on the Solaris Platform*, Prentice Hall, 2002.
9. Sun Microsystems, *Solaris Dynamic Tracing Guide*, http://docs.sun.com/app/docs/doc/817-6223.
10. Wong, B., *Configuration and Capacity Planning on Sun Solaris Servers*, Prentice Hall, 1997.

6. Acknowledgements

People who have contributed thoughts, ideas, reviews, ...

- Claire Black
- Dr Rex di Bona
- Allan Packer

7. Glossary

Since new terms have been introduced in this paper, the following glossary has been compiled.

- **Adaptive Disk I/O Time** – an algorithm to closely estimate the Disk Response Time from a series of strategy and biodone events.
- **Application Response Time** – the time from the start of an application event to the end. the response time experienced by the client.
- **Asymmetric Resource Utilisation** – is where load cannot be easily shared across components of a resource. For example, network interface cards.
- **bdev_strategy** – see strategy.
- **biodone** – the block I/O driver function that receives the completion of a disk event.
- **Disk Response Time** – the time consumed by the disk to service a particular event.
- **Disk Utilisation by Process** – a measure of how a process is causing the disks to be. Presented in terms of a single disk, hence 400% utilisation means 4 disks at 100%, or 8 at 50%, or some such combination.
- **Driver Response Time** – the time from the driver request for a disk event to its completion.
- **DTrace** – the Dynamic Tracing facility in the Solaris 10 operating system.
- **Duelling Banjos** – a condition where two processes repeatedly access either end of a disk, causing each other to seek further than would be expected.
- **procfs** – the process filesystem, /proc. This contains by process statistics.
- **Random Disk I/O** – where a series of disk events access block addresses scattered across the disk. This causes high seek times.
- **Sequential Disk I/O** – where a series of disk events access adjacent or sequential block addresses. This increases disk performance as it reduces seek time.
- **Symmetric Resource Utilisation** – is where load is easily shared across components of a resource. For example, banks of RAM.
- **strategy** – the block I/O driver function that requests a disk event.
- **Target Fixation** – a term to describe a condition a fighter pilot may experience when focusing too heavily on a target at the expense of other dangers. Here it was used to warn against grepping your PID and missing other import disk event details. (Any excuse to use fighter pilot terminology).
- **Thread Response Time** – the time from the thread blocking on an event, to waking up again.
- **TNF tracing** – Trace Normal Form tracing. A facility to add trace probes into production code, from Solaris 2.5. Some kernel probes are also provided by default.