

Brendan Gregg's professional blog

- [Home](#)
- [About](#)



Brendan's blog

Using SystemTap

I work at [Joyent](#) – a cloud computing company – doing performance analysis of small to large cloud environments. Most of our systems have [DTrace](#): the first widely used implementation of Dynamic Tracing, which we use for monitoring and performance analysis. But we also have some Linux, which I've been analyzing in the following situations:

- Competitive performance analysis: measuring the runtime of certain codepaths in Linux vs [SmartOS](#).
- KVM performance tuning: we run Linux as guests of the KVM [we ported](#) to [Illumos](#), and as part of that work I've studied performance from the host (via DTrace) and from the guest (via SystemTap).
- Real customer issues: replicated in a KVM lab environment for safe study with SystemTap.

I would prefer to log in to the KVM guests and run DTrace, but it's not available on those Linux systems. (In the future, we may have a way to reach into the guest using DTrace from the host to trace its internals, be it Linux or Windows.) In the meantime, I need to do some serious performance analysis on Linux (including kernel code-path latency) so I'm using [SystemTap](#).

Why I'm blogging about it

While using SystemTap, I've been keeping notes of what I've been doing, including what worked and what didn't, and how I fixed problems. It's proven a handy reference.

In some recent threads about DTrace, people have asked how it compares to SystemTap – with some commenting that they haven't had time to study either. I've been encouraged to post about my experiences, which is easy to do from my notes. I could (and probably should) get some of these into the SystemTap bug database, too.

What I'm sharing are various experiences of using SystemTap, including times where I made mistakes because I didn't know what I was doing. I'll begin with a narrative about tracing disk I/O, which connects together various experiences. After that it gets a little discontinuous, clipped from various notes. I'll try to keep this technical, positive, and constructive. Perhaps this will help the SystemTap project itself, to see what a user (with a strong Dynamic Tracing and kernel engineering background) struggles with.

Who's using my disk I/O?

There are plenty of examples out there for using DTrace (including [posts](#) on my blog), but not so many for SystemTap. A good one I've seen is titled [SystemTap: It's like dtrace for linux, yo.](#), which begins with "Who's using my disk I/O?". The author writes:

One of my favorite (now retired) interview questions for UNIX administrators is "iostat shows a lot of write activity to a device. How do you determine what program is responsible for writing the most data to this device?". Before SystemTap, the only way to answer this under Linux was to install kernel counter patches, or try to parse `block_dump`. Using the `disktop` script from the SystemTap Scripts & Tools page, it's trivial to find this kind of information out.

That's a great question!

Getting a quick sense of who is using the disks is a problem I've dealt with for over a decade, and it isn't answered well by standard tools. It's getting more important as systems get larger and more crowded, and have more applications sharing the same set of disks.

This makes for a great example of Dynamic and Static tracing, which can be used to examine disk I/O events in whatever detail is desired. By stepping through this example, I'll be able to discuss and share various experiences of using SystemTap.

disktop

`disktop.stp` is on the SystemTap page. It's also shown as the first example in section [4.2 Disk](#) of the Red Hat Enterprise Linux documentation for SystemTap.

I'll run it as the #1 example, just as the previous author did. First, I need to install SystemTap.

Installation

SystemTap has required assembly on the Linux distros I've tried, Ubuntu and CentOS. Necessary steps may include:

1. Adding the `systemtap` package
2. Adding various extra packages, such as `linux-kernel-headers`
3. Finding, downloading and installing the correct kernel debuginfo (which for my distros is usually a 610 Mbyte download)
4. Upgrading the kernel to match other package versions, and rebooting
5. Patching `/usr/share/systemtap C` code (either following instructions or on your own)
6. Patching `/usr/share/systemtap` headers for the build system (on your own)
7. Power cycling the system if the kernel freezes
8. Extra manual steps (eg, running the little script at the end of [this page](#))

It takes me about an hour each time I do this.

You may not need all these steps, or you may need more. The SystemTap engineers have said it's much easier on Red Hat Enterprise Linux (I've yet to try).

I'd recommend keeping a text file of SystemTap install (and runtime) failures, including screen output and commands tried. I've found it a great timesaver: now when I hit failures I can grep my text file to see if I've debugged that one before, and what commands I used.

Also, If you gather enough info for install failures, you'll be able to file SystemTap bugs so that the product can get better over time.

Filing Bugs

The [bug tracking system](#) is at Red Hat, and requires a login (easy to get). There is a good page on the SystemTap wiki on [How To Report Bugs](#), which lists information that can be helpful to gather. Extra info is requested for these types of problems:

The most unfortunate problems occur when systemtap compiles your probe script properly, but then the machine dies.

Some of the details I'd been forgetting to gather included:

compiler version, environment variables, machine architecture, the version of all related packages installed, if a probe module .ko file was generated – the output of modinfo FILE.ko, systemtap build process (stap -p4 -vv), /proc/modules, /proc/kallsyms, kdump image, crash analysis tool backtrace, module containing the eip (using insmod and /proc/modulesto).

These aren't all necessary, and if the process seems onerous I take it you can just chat to the engineers in #systemtap in irc.freenode.net instead. I'm told they are friendly. :-)

Bug Searching

When you go to file a bug, [search](#) to see if it's already there. I search for both open and closed bugs, as I may have hit a bug that's recently been closed (fixed), but not on the version of SystemTap I'm running. In which case, I'll still find the bug, notice that the fix date was recent, and then go refresh to the latest SystemTap. Here's an example search:

Summary: crash panic freeze hung				Resolution: ---, FIXED, INVALID, WONTFIX, WORKSFORME, MOVED	Product: systemtap	Summary: crash panic freeze
60 bugs found.						
ID	Sev	Pri	Assignee	Status	Resolution	Summary
3452	cri	P1	hunt@redhat.com	RESO	FIXE	stap crash ppc64
1234	cri	P1	jkenisto@us.ibm.com	RESO	FIXE	kprobes crashes kernel on x86_64 SMP
3278	cri	P1	jistone@redhat.com	RESO	FIXE	hrtimers cause kernel crash
1813	nor	P1	jkenisto@us.ibm.com	RESO	FIXE	crash due to kmalloc probe / kprobe-registration reentrancy
2071	nor	P1	mhiramat@redhat.com	RESO	FIXE	Probes on ISR with probes on task thread's prehandler crash the system
1175	cri	P2	fche@redhat.com	RESO	FIXE	bad key lookup crashes kernel
4542	cri	P2	mhiramat@redhat.com	RESO	FIXE	systemtap stress/current.stp crashes on 2.6.22-rc2 kernel
6964	cri	P2	fche@redhat.com	RESO	FIXE	process probes cause kernel crash on f9
11140	cri	P2	systemtap@sourceware.org	RESO	FIXE	SystemTap userspace marker in shared libraries cause probed program crash
9989	cri	P2	dsmith@redhat.com	RESO	FIXE	utrace or task-finder crash in upstream utrace git tree
3785	cri	P2	systemtap@sourceware.org	RESO	FIXE	mismatched kernel and debuginfo architecture causing system crashes
2667	cri	P2	systemtap@sourceware.org	RESO	FIXE	Kernel panic occur when kprobing callq instruction on x86_64
9740	cri	P2	mhiramat@redhat.com	RESO	FIXE	syscall.* probe causes kernel panic(double fault) on rawhide kernel/i686
5963	cri	P2	fche@redhat.com	RESO	FIXE	testsuite/systemtap/maps/pmap_egg_overflow.stp crashes on 2.6.25-0.121.rc5.git4.fc9
2725	cri	P2	rarora@redhat.com	ASSI	---	function(**) probes sometimes crash & burn
6816	cri	P3	sasath@us.ibm.com	DECN	FIXE	functionalized stp causes system crash

I tried filtering out really old bugs to reduce the list, but that excluded at least one crash that I've hit recently: [#2725](#) “function(“**”) probes sometimes crash & burn” (filed over 5 years ago, last updated over 2 years ago).

Manual Patching

As an example of the sort of issue you may hit, this is what happened on my last install (Ubuntu 11.4):

```
# stap -e 'global ops; probe syscall.*.return { ops[probe_func()] <<< 1; }'
ERROR: Build-id mismatch: "kernel" vs. "vmlinux-2.6.38-11-server" byte 0 (0x31 vs 0x09) rc 0 0
Pass 5: run failed. Try again with another '--vp 00001' option.
```

This one-liner counts system calls, which I'm running to check that everything is installed and working. It failed with the “build-id mismatch” error highlighted above, which I hadn't seen before.

This wasn't in the Red Hat bug database, but some Internet searching found a useful [post](#) that covered this error, suggesting to manually patch the /usr/share/systemtap/runtime/sym.c file. The patch was:

```
if (!strcmp(m->name, "kernel")) {
/* notes_addr = m->build_id_offset; REPLACE THIS LINE BY THE NEXT ONE */
notes_addr = _stp_module_relocate("kernel", "_stext", m->build_id_offset);
base_addr = _stp_module_relocate("kernel", "_stext", 0);
} else {
```

But this patch had already been applied. I checked the C code in the runtime directory, and started wondering if the “mismatch” was due to running an older kernel but with a newer patched sym.c. So, as an experiment, I rolled back the patch. When I ran the one-liner again, the system hung hard, requiring a power cycle. My mistake! I rolled back all my changes.

Some digging later led me to try upgrading the kernel to resolve the mismatch. That worked!

Execution

Now that SystemTap is finally installed, I'll run disktop:

```
# ./disktop.stp
WARNING: never-assigned local variable 'write' (alternatives: bytes process cmd userid parent action io_stat device read_bytes write_k
source: if (read_bytes+write+bytes) {
WARNING: never-assigned local variable 'bytes' (alternatives: write process cmd userid parent action io_stat device read_bytes write_k
source: if (read_bytes+write+bytes) {
```

At this point the machine died:

```
idrac-3POD8P1, PowerEdge R710, User:root, 43.7fps
Virtual Media File View Macros Tools Power Help
[ 5.796709] scsi 3:0:0:0: Direct-Access iDRAC LCDRIVE 0323 PQ
[ 0 ANSI: 0 CCS
[ 5.797732] sd 3:0:0:0: Attached scsi generic sg4 type 0
[ 5.798583] scsi 4:0:0:0: CD-ROM iDRAC Virtual CD 0323 PQ
[ 0 ANSI: 0
[ 5.806437] scsi 4:0:0:1: Direct-Access iDRAC Virtual Floppy 0323 PQ
[ 0 ANSI: 0 CCS
[ 5.807696] sd 3:0:0:0: [sdc] Attached SCSI removable disk
[ 5.911675] sr1: scsi3-mmc drive: 0x/0x cd/rw caddy
[ 5.911866] sr 4:0:0:0: Attached scsi generic sg5 type 5
[ 5.912001] sd 4:0:0:1: Attached scsi generic sg6 type 0
[ 5.914884] sd 4:0:0:1: [sdd] Attached SCSI removable disk

Ubuntu 11.04 bh1-kum4 tty1
bh1-kum4 login: [ 570.544560] -----[ cut here ]-----
[ 570.545596] kernel BUG at /build/builddd/linux-2.6.38/kernel/timer.c:668!
[ 570.547088] invalid opcode: 0000 [#1] SMP
[ 570.576346] last sysfs file: /sys/module/vesafb/sections/_param
[ 570.606091] CPU 10
[ 570.606554] Modules linked in: stap_1364d314af61951db6c441ebca062c32_18366 ve
safb psmouse dcdbas ghes serio_raw joydev i7core_edac edac_core power_meter hed
lp parport ses enclosure usbhid hid usb_storage uas megaraid_sas igb bnx2 dca [I
ast unloaded: stap_1364d314af61951db6c441ebca062c32_18366]
[ 570.697564]
[ 570.697567] Pid: 2849, comm: stapiot Not tainted 2.6.38-11-server #50-Ubuntu D
ell Inc. PowerEdge R710-00NH4P
[ 570.697572] RIP: 0010:[ffffffffff810767db] [<ffffffffff810767db>] mod_timer+0x
29b0x2b0
```

Specifically, the system has hung. It's not responding to keystrokes via either SSH or the console, and it's not responding to ping. The only way out seems to be power cycling the system. It has a system controller (DRAC) allowing me to power cycle it remotely.

This was a similar hang to what I saw earlier when I modified the runtime/sym.c files, so perhaps that wasn't my mistake after all. I don't know – this will need careful analysis. Fortunately I have a good place to start: the SystemTap bugs checklist mentioned earlier. I'm currently following one of the steps: getting kdump configured and enabled so that I can capture kernel crash dumps. I can't be certain what went wrong until I take a look; it's possible that the crash was in a different kernel subsystem which SystemTap happened to trigger (ie, not the SystemTap code itself).

Safe use on production systems

That wasn't the only system I've had that has crashed when running SystemTap. The [comparison](#) table on the SystemTap wiki has a row for "safe use on production systems". At the time of writing, the SystemTap column has "yes".

To write this post, my system has hung when running SystemTap more than six times (I lost count), requiring a power cycle each time. While this system seems more brittle than others on which I've run SystemTap (I usually have to run more than just disktop.stp to hit a crash), I still don't feel safe using SystemTap in production.

In the past seven years, I've used DTrace in production thousands of times, on hundreds of different systems. It's running continuously, in production, to monitor every system in the new Joyent datacenters. It can also be configured to run continuously to monitor Oracle's ZFS Storage Appliance. I've never known it to cause system crashes or hangs.

DTrace's safety record was a product of the kernel engineering culture at Sun Microsystems, and of [Bryan Cantrill](#): father of Dynamic Tracing and DTrace. Bryan wrote about [DTrace Safety](#) in 2005, explaining the risks and principle of production safety very clearly:

DTrace must not be able to accidentally induce system failure. It is our strict adherence to this principle that allows DTrace to be used with confidence on production systems.

This was also the aim of Solaris in general, as described in the [Developing Solaris](#) article, which was also written by Bryan. Here is an excerpt:

You should assume that once you putback your change, the rest of the world will be running your code *in production*. More specifically, if you happen to work in MPK17, within three weeks of putback, your change will be running on the building server that *everyone in MPK17 depends on*. Should your change cause an outage during the middle of the day, some 750 people will be out of commission for the order of an hour. Conservatively, every such outage costs Sun \$30,000 in lost time — and depending on the exact nature of who needed their file system, calendar or mail and for what exactly, it could cost much, much more.

I've included this to describe the background from which DTrace was born. I won't talk at length about SystemTap's safety, since I don't understand that very well yet. Any of the following may be true:

- The Linux kernel may be more crash-prone due to other subsystems, not SystemTap.
- The Linux kernel may have fundamentally different behaviour to Solaris, Mac OS X, and FreeBSD, such that avoiding system crashes may be inherently more difficult.
- The SystemTap architecture may introduce additional risk.
- My experience is uncommon.

Right now, all I know is that running SystemTap may crash or hang the systems I'm on. I don't know exactly why (crash dump analysis takes time), but I don't think I'm [alone](#) either.

Most of the Time

Fortunately, I think most of the time disktop.stp doesn't crash your system. Some of the time I just got this second error (but no system crash):

```
# ./disktop.stp
WARNING: never-assigned local variable 'write' (alternatives: bytes process cmd userid parent action io_stat device read_bytes write_k
source:         if (read_bytes+write+bytes) {
                  ^
WARNING: never-assigned local variable 'bytes' (alternatives: write process cmd userid parent action io_stat device read_bytes write_k
source:         if (read_bytes+write+bytes) {
                  ^
ERROR: kernel read fault at 0x7473010b00680e42 (addr) near identifier '@cast' at /usr/share/systemtap/tapset/nfs.stp:141:7
WARNING: Number of errors: 1, skipped probes: 0
Pass 5: run failed. Try again with another '--vp 00001' option.
```

The kernel read fault address changes each time. This seems to be exacerbated by running load, which I'm doing so that we have something to see with disktop. By trying again numerous times, I finally got disktop to run long enough to collect a screenshot.

Screenshot

```
# ./disktop.stp
WARNING: never-assigned local variable 'write' (alternatives: bytes process cmd userid parent action io_stat device read_bytes write_k
source:         if (read_bytes+write+bytes) {
                  ^
WARNING: never-assigned local variable 'bytes' (alternatives: write process cmd userid parent action io_stat device read_bytes write_k
source:         if (read_bytes+write+bytes) {
                  ^
[...]
Tue Oct 11 20:38:16 2011 , Average:5003498Kb/sec, Read: 25017490Kb, Write: 0Kb

  UID      PID      PPID          CMD    DEVICE    T      BYTES
    0      7491      2604          dd      sdb2     R  1073747282
    0      7499      2604          dd      sdb2     R  1073747282
    0      7507      2604          dd      sdb2     R  1073747282
    0      7515      2604          dd      sdb2     R  1073747282
    0      7523      2604          dd      sdb2     R  1073747282
    0      7531      2604          dd      sdb2     R  1073747282
    0      7539      2604          dd      sdb2     R  1073747282
    0      7547      2604          dd      sdb2     R  1073747282
    0      7555      2604          dd      sdb2     R  1073747282
    0      7563      2604          dd      sdb2     R  1073747282
^CPass 5: run failed. Try again with another '--vp 00001' option.
```

Neat! You can see the process ID, command name, disk device, direction and bytes – directly associating the process with the disk activity.

UPDATE: As pointed out in a comment on this post, the WARNINGS seen above are actually from a bug in the code (typo). For a write-only workload, the bug would cause the summary and header lines to not be printed (was that workload originally tested?). For this workload (reads present) the bug just means that those warnings are printed, but the output is all intact. It's still a little annoying to have warnings printed out (from what is a core script example), and I'm guessing that the warning code predated this script (which is from 2007).

The output shows that it's all those dd processes that are hammering the disks with reads. I could tune my filesystem to better cache those reads in DRAM, or just buy more DRAM.

Uh-oh...

The summary line above said that the disks were averaging 5003498 KB/sec, which is 4.8 GB/sec. This system has four 7200 RPM disks presented via hardware RAID as a single device. I'm pretty sure they can't do 4.8 GB/sec combined.

According to iostat, the disks are actually doing this:

```
# iostat -d 1
Linux 2.6.38-11-server (linux31)      10/11/2011      _x86_64_      (16 CPU)
[...]
Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda                 0.00         0.00         0.00         0          0
sdb                 0.00         0.00         0.00         0          0

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda                 0.00         0.00         0.00         0          0
sdb                 0.00         0.00         0.00         0          0

Device:            tps    kB_read/s    kB_wrtn/s    kB_read    kB_wrtn
sda                 0.00         0.00         0.00         0          0
sdb                 0.00         0.00         0.00         0          0
[...]
```

That's right – **nothing**.

My dd workload has been running for so long it's now just hitting out of DRAM cache, yet disktop.stp claims that they are still reading Gbytes from disk. And I'm pretty sure that the point of *disktop.stp* was to measure *disk* activity. From the SystemTap [wiki](#):

disktop.stp : List the top ten activities of reading/writing disk every 5 seconds

From the Red Hat Enterprise Linux [documentation](#):

disktop.stp outputs the top ten processes responsible for the heaviest reads/writes to disk

I've seen a couple of different versions of disktop.stp, and both suffer this issue.

Sanity Check

Here's an excerpt from the SystemTap [wiki](#) version of disktop:

```
probe kernel.function("vfs_read").return {
    if ($return>0) {
        dev = __file_dev($file)
        devname = __find_bdevname(dev, __file_bdev($file))

        if (devname!="N/A") { /*skip read from cache*/
            io_stat[pid(),execname(),uid(),ppid(),"R"] += $return
            device[pid(),execname(),uid(),ppid(),"R"] = devname
            read_bytes += $return
        }
    }
}
```

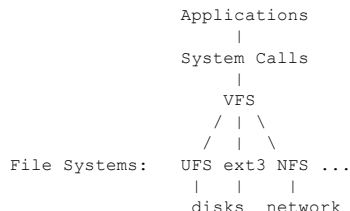
A different version exists in the Red Hat Enterprise Linux [docs](#) and the SystemTap beginners [guide](#):

```
probe vfs.read.return {
  if ($return>0) {
    if (devname!="N/A") { /*skip read from cache*/
      io_stat[pid(),execname(),uid(),ppid(),"R"] += $return
      device[pid(),execname(),uid(),ppid(),"R"] = devname
      read_bytes += $return
    }
  }
  [...]
}
```

Oh, no. Both are tracing at the Virtual File System (VFS) level, not the disk level!

Operating Systems 101

There is some code that tries to “skip read from cache” based on device information. VFS is supposed to abstract the underlying file systems and provide a well-defined interface (originally created by Sun Microsystems in 1985 for SunOS 2.0, to allow UFS and NFS to coexist).



Storage device details are not part of the VFS read interface, so to examine them in VFS read context is immediately suspicious. Perhaps Linux does it anyway (what some may call a “layer violation”).

What’s also suspicious is the lines of code in the wiki version:

```
dev = __file_dev($file)
devname = __find_bdevname(dev, __file_bdev($file))
```

The first function comes from the `vfs tapset` in `/usr/share/systemtap/tapset/vfs.stp`:

```
function __file_dev:long (file:long)
{
  d_inode = __file_inode(file)
  if (d_inode == 0)
    return 0
  return @cast(d_inode, "inode", "kernel")->i_sb->s_dev
}
```

That walks the file inode (`d_inode`), to the file system superblock (struct `super_block *i_sb`), to the device info (`dev_t *s_dev`). So far that’s just checking that the file resides on a file system that has a backing device.

For the `disktop.stp` logic to work, the second line will need to do the “skip read from cache” logic. It calls `__find_bdevname`, also in the `vfs tapset`, which takes that device info and returns the device name:

```
/*
 * We don't want to have to do a bdevname() call every time
 * we want a devname, so we'll hash them here.
 */
/* XXX: Is this hashing really that helpful? The call to bdevname()
 * isn't very involved... */
global __devnames
function __find_bdevname:string(dev:long, bdev:long)
{
  if (dev in __devnames)
    return __devnames[dev]
  else
    return __devnames[dev] = bdevname(bdev)
}
```

(I’m tempted to join the comment conversation at the top, adding, tongue-in-cheek: “/* XXX: Have you heard of SystemTap? I bet that can measure `bdevname()` cost. ;-) */”.)

`__find_bdevname()` calls `bdevname()`, which is declared in the Linux kernel source (`fs/partitions/check.c`):

```
const char *bdevname(struct block_device *bdev, char *buf)
{
  return disk_name(bdev->bd_disk, bdev->bd_part->partno, buf);
}
```

`disk_name()` just does the formatting. So I don’t see how the “skip read from cache” logic works by examining the code. It also doesn’t pass a sanity check (as previously mentioned), nor work in practise (as previously shown). It looks busted, and has been that way since the script was written in 2007.

Does it Matter?

Performance tools don’t need to be perfect to be useful. If it steers you in the right direction, you apply a fix, and the fix works, it may not matter much that the numbers were out by 5% along the way. Just bear in mind that the tool has an error margin.

Here, `disktop.stp` is measuring I/O at the VFS level instead of the disk level. Does that matter? Most disk I/O begins life as VFS I/O anyway (either directly or indirectly), so the numbers could be pretty close.

To get an idea of the difference, I measured the I/O at both the VFS and disk level on several production servers. Since I want production data, I won’t use SystemTap (for risk of system crash). Instead, I’ll pick production servers that have DTrace. The script, `vfs.d`, just traces reads to start with:

```

1  #!/usr/sbin/dtrace -qs
2
3  dtrace::BEGIN { printf("Tracing... Hit Ctrl-C to end.\n"); }
4
5  fbt::fop_read:entry
6  {
7      @io[execname, "VFS"] = sum(args[1]->uio_resid);
8  }
9
10 io:::start
11 /args[0]->b_flags & B_READ/
12 {
13     @io[execname, "disk"] = sum(args[0]->b_bcount);
14 }
15
16 dtrace::END
17 {
18     printf("    %-18s %-8s %12s\n", "EXEC", "LEVEL", "BYTES");
19     printa("    %-18s %-8s %12d\n", @io);
20 }

```

`fop_read()` is the Solaris VFS interface for read. Disk I/O is traced via the `io:::start` probe. To keep this simple, I didn't track context properly for the disk I/O, so some execnames may be misidentified as the kernel (asynchronous prefetch). Which is fine, as I'm really running it to compare the byte counts.

Here's the first production server I tried, tracing for about ten seconds:

```

[root@xxxxxx]# ./vfs.d
Tracing... Hit Ctrl-C to end.
^C
EXEC          LEVEL          BYTES
zeus.flipper   VFS              53
zeus.monitor   VFS             14336
nscd           VFS             20480
sshd           VFS             65536
zeus.zxtm      VFS             118796
mysqld         disk             622592
rotatelog      VFS             6946816
mysqld         VFS             25630141
httpd          VFS             36132135
memcached      VFS             267374584

```

memcached is pretty busy, reading over 200 Mbytes, but that's returning entirely from filesystem DRAM cache or from another VFS fronted subsystem (perhaps socket reads over socks; more DTrace can confirm). mysqld was the only application that missed out of the DRAM cache, reading about 600 Kbytes from disk. This ratio wasn't dissimilar to the other production servers.

disktop.stp, reporting VFS-based numbers, would have claimed about 300 Mbytes of disk reads. In reality it was about 0.6 Mbytes. That's out by 500x!

In some situations, this may be a worse problem than having the system crash. With crashes, you know it happened, and can take action immediately to avoid it. Here the disktop tool appears to work, but is providing bad data. You may act upon bad data without realising it for months or years.

When a tool reports bad data, the best case is when it's so bad it's *obviously* wrong. If I did run disktop.stp on the above server and had it report memcached at over 200 Mbytes, I'd immediately know something was wrong: the disks aren't doing that (as we know from iostat) and it doesn't make sense for memcached. Worst case is where the numbers are subtly wrong, so that it's not obvious.

Avoiding This

Metrics are easy. Correct metrics are hard.

Dynamic Tracing makes it easy to produce numbers for all sorts of activity, but that doesn't mean they are *right*. I learned this years ago in creating the [DTraceToolkit](#), which felt like the *TestingToolkit* as most of my time was spent testing the scripts (and I'm very grateful for the time others donated to help with this testing).

Here are a few suggestions:

Buy a pocket calculator. There are always little sums you can perform as a form of checking. Did the per second throughput exceed maximum bus, disk or network throughput? This is what alerted me to the issue with disktop.stp: more throughput than made sense.

If you can't code the workload, you can't code the script. Get good at writing mini tools that apply a *known workload*, which you can then double check with what your SystemTap script reports. I'd recommend C, and sticking to syscalls (to avoid library buffering) where possible, so that the workload applied is exactly what you coded. I've had cases where people have complained that coding (or applying) a known workload is too hard: that may be a good sign that you shouldn't attempt the script! Writing the workload will often reveal subtle details that should be handled correctly by your observability tool. It's a very good idea, wherever possible, to try to write the workload.

Chase down small discrepancies. If your SystemTap script reports close to the known workload, but not quite, it's usually worth chasing down why. I've discovered many behaviours this way, which may be small discrepancies for the test workload but larger in production. Reasons can include tracing the common path through code (aka "fast path") but not the edge cases ("slow path"). Sometimes what you try will never make sense, and you discover that the system is much more complex than you thought.

So, I'm not really surprised that the disktop.stp script is broken – I've seen a lot of this sort of mistake before. I am a little surprised that it doesn't appear to have been noticed and fixed since 2007, though.

Another thing...

The output of disktop is pretty familiar to me, having written iotop for the [DTraceToolkit](#) years earlier:

```

# iotop -C
Tracing... Please wait.
2005 Jul 16 00:34:40, load: 1.21, disk_r: 12891 KB, disk_w: 1087 KB

  UID    PID   PPID CMD          DEVICE MAJ MIN D      BYTES
   0      3     0 fsflush    cmdk0  102  4 W      512

```

0	3	0	fsflush	cmdk0	102	0	W	11776
0	27751	20320	tar	cmdk0	102	16	W	23040
0	3	0	fsflush	cmdk0	102	0	R	73728
0	0	0	sched	cmdk0	102	0	R	548864
0	0	0	sched	cmdk0	102	0	W	1078272
0	27751	20320	tar	cmdk0	102	16	R	1514496
0	27751	20320	tar	cmdk0	102	3	R	11767808

Apart from the visual similarities, both tools default to a 5 second interval. This wouldn't be the first time that [my work](#) has inspired SystemTap.

If disktop was inspired by my iotop, it'd be nice if the script mentioned that somewhere (as Red Hat consulting [did](#)). I certainly mentioned where I got the idea for iotop:

```
brendan@macbook:~> grep LeFebvre /usr/bin/iotop
# INSPIRATION: top(1) by William LeFebvre
```

It's not necessary, it's just a nice thing to do. You never know, one day you may [meet](#) the guy!

Did I get the job?

At the start of this article I mentioned a Linux interview question: "Who's using my disk I/O?", which was now considered trivial to answer thanks to disktop.

If I'm being interviewed for that job, and dived into an explanation of VFS and why all the official documentation was wrong, would I get the job?

iostat-scsi.stp

While disktop.stp is top of the disk sections of SystemTap documentation, it's not #1 on the SystemTap [Scripts & Tools](#) page. That is iostat-scsi.stp.

Scripts (by category)

- Audit
- CPU
- Disk
 - [iostat-scsi.stp](#) [\(example output\)](#)
 - [disktop.stp](#): List the top ten activities of reading/writing disk every 5 seconds([example output](#))
 - [stripesize.stp](#): Helps choose and validate a RAID array's stripe size.

Which should be useful, since the iostat(1) tool on current versions of Linux don't include all devices, such as tape drives. Fetching this sort of basic observability is a cakewalk for Dynamic Tracing.

Running it:

```
# ./iostat-scsi.stp
parse error: embedded code in unprivileged script
saw: embedded-code at ./iostat-scsi.stp:28:1
source: %{
      ^
parse error: embedded code in unprivileged script
saw: embedded-code at ./iostat-scsi.stp:32:39
source: function get_nr_sectors:long(rq:long) %{ /* pure */
      ^
parse error: command line argument index 1 out of range [1-0]
at: junk '$1' at ./iostat-scsi.stp:67:15
source: probe timer.s($1) {
      ^
parse error: command line argument index 1 out of range [1-0]
at: junk '$1' at ./iostat-scsi.stp:75:31
source: tps=(rdcount+wrcount)*100/$1
      ^
parse error: expected 'probe', 'global', 'function', or '%{'
saw: identifier 'rdblkcount' at ./iostat-scsi.stp:76:5
source: rdblkcount=rdcount ? @sum(reads[dev])/blksize : 0
      ^
parse error: command line argument index 1 out of range [1-0]
at: junk '$1' at ./iostat-scsi.stp:78:30
source: rdblkrate=rdblkcount*100/$1
      ^
parse error: expected 'probe', 'global', 'function', or '%{'
saw: identifier 'wrblkrate' at ./iostat-scsi.stp:79:5
source: wrblkrate=wrblkcount*100/$1
      ^
parse error: command line argument index 1 out of range [1-0]
at: junk '$1' at ./iostat-scsi.stp:79:30
source: wrblkrate=wrblkcount*100/$1
      ^
parse error: expected 'probe', 'global', 'function', or '%{'
saw: identifier 'printf' at ./iostat-scsi.stp:80:5
source: printf("%9s %6d.%02d %6d.%02d %6d.%02d %9d %9d\n",
      ^
parse error: expected 'probe', 'global', 'function', or '%{'
saw: identifier 'printf' at ./iostat-scsi.stp:86:3
source: printf ("\n")
      ^
10 parse error(s).
Pass 1: parse failed. Try again with another '--vp 1' option.
```

Whoa. Something is seriously wrong.

Debugging iostat-scsi.stp

Perhaps this script is dependant on a SystemTap version or feature I'm missing. Taking a look:

```
# cat -n iostat-scsi.stp
1 #! /usr/bin/env stap
```



```

2
3 global devices, reads, writes
4
5 /* data collection: SCSI disk */
6 %(kernel_v<"2.6.24" %?
7 probe module("sd_mod").function("sd_init_command") !, kernel.function("sd_init_command") {
8     device=kernel_string($SCpnt->request->rq_disk->disk_name)
9     sector_size=$SCpnt->device->sector_size
10    nr_sectors=$SCpnt->request->nr_sectors
11    devices[device] = 1
12    %(kernel_v>="2.6.19" %?
13    if ($SCpnt->request->cmd_flags & 1)
14    %:
15    if ($SCpnt->request->flags & 1)
16    %)
17        writes[device] <<< nr_sectors * sector_size
18    else
19        reads[device] <<< nr_sectors * sector_size
20    }
21    %:
22
23 function get_sector_size:long (data:long) {
24     return @cast(data, "scsi_device", "kernel<scsi/scsi_device.h>")->sector_size
25 }
26
27 %(kernel_v>="2.6.31" %?
28 %{
29 #include <linux/blkdev.h>
30 %}

```

That's just the top – the whole script is 90 lines long. It looks pretty complicated, and runs different code depending on the kernel version. I suppose it's listed first because of the “by category” ordering of the SystemTap page. Due to its complexity – not to mention the 40 lines of errors when I tried to run it – it certainly doesn't look like a good script example to place first.

USAGE

As part of debugging this script, it would be helpful if it had a header to describe version dependencies and usage. Like the following, for example:

```

# cat -n iosnoop
1  #!/usr/bin/sh
2  #
3  # iosnoop - A program to print disk I/O events as they happen, with useful
4  #           details such as UID, PID, filename, command, etc.
5  #           Written using DTrace (Solaris 10 3/05).
6  #
7  # This is measuring disk events that have made it past system caches.
8  #
9  # $Id: iosnoop 8 2007-08-06 05:55:26Z brendan $
10 #
11 # USAGE:      iosnoop [-a|-A|-DeghiNostv] [-d device] [-f filename]
12 #              [-m mount_point] [-n name] [-p PID]
[...]
```

I eventually found the problem with iostat-scsi.stp: it has to be executed in a certain way (with “-g”):

```

# stap -g iostat-scsi.stp 1
Device:      tps blk_read/s blk_wrtn/s  blk_read  blk_wrtn
sdb         27.00      0.00    216.00      0      216
[...]
```

Ok, so I was using it incorrectly. The example file even showed that, which I missed.

Suggestions

Apart from adding a header to the script, it would have been a better experience if it also printed a USAGE message if misused. Eg:

```

# iosnoop -z
/usr/bin/iosnoop: illegal option -- z
USAGE: iosnoop [-a|-A|-DeghiNostv] [-d device] [-f filename]
           [-m mount_point] [-n name] [-p PID]
[...]
```

Instead of 40 lines of errors.

I shouldn't complain, since this was from a wiki page where anyone could contribute anything. But it's also the first script listed, and doesn't work as I'd expect a “script” to work. I sometimes wonder if I'm the only person trying these things out.

What was that -g?

I just looked up -g in the SystemTap [Language Reference](#):

SystemTap supports a *guru mode* where script safety features such as code and data memory reference protection are removed. Guru mode is set by passing the -g option to the stap command. When in guru mode, the translator accepts C code enclosed between “%{” and “%}” markers in the script file. The embedded C code is transcribed verbatim, without analysis, in sequence, into generated C code.

!!! So the **first script example** is running in **guru mode** so that it can execute **arbitrary C code in kernel context without safety features**. Code that *varies* between kernel versions:

```

6  %(kernel_v<"2.6.24" %?
7  probe module("sd_mod").function("sd_init_command") !, kernel.function("sd_init_command") {

```



```

8     device=kernel_string($SCpnt->request->rq_disk->disk_name)
9     sector_size=$SCpnt->device->sector_size
10    nr_sectors=$SCpnt->request->nr_sectors
11    devices[device] = 1
12    %(kernel_v>="2.6.19" %?
13    if ($SCpnt->request->cmd_flags & 1)

```

So if you run this on a new kernel version that isn't handled by the code – which seems *very possible* given the special casing it already does for kernel versions – then you are running possibly incorrect code with safety features disabled!

And this is the *first example* on the SystemTap [Scripts & Tools](#) page.

vfsrlat.stp

Here is an example experience of things going (mostly) well with SystemTap. I was interested in the distribution of latency for cached reads on different Linux file systems, especially to see if there were any latency outliers (occasional instances of high latency, which can be caused by lock contention during file system events, for example). I wrote a SystemTap script that does this: `vfsrlat.stp`.

Script

This can be traced a few different ways. It could trace at the system call level, the VFS level, or in the file systems themselves. I picked VFS, to capture different file system types from one script. A logarithmic histogram is printed to summarize the latency (in nanoseconds), and each file system type is printed separately.

```

# cat -n vfsrlat.stp
1  #!/usr/bin/env stap
2  /*
3   * vfsrlat.stp  Summarize VFS read latency for different FS types.
4   *
5   * USAGE: ./vfsrlat.stp          # Ctrl-C to end
6   *
7   * This prints a histogram so that the presense of latency outliers can be
8   * identified.
9   *
10  * WARNING: This is known to crash systems.
11  *
12  * COPYRIGHT: Copyright (c) 2011 Brendan Gregg.
13  *
14  * CDDL HEADER START
15  *
16  * The contents of this file are subject to the terms of the
17  * Common Development and Distribution License, Version 1.0 only
18  * (the "License").  You may not use this file except in compliance
19  * with the License.
20  *
21  * You can obtain a copy of the license at Docs/cddl1.txt
22  * or http://www.opensolaris.org/os/licensing.
23  * See the License for the specific language governing permissions
24  * and limitations under the License.
25  *
26  * CDDL HEADER END
27  */
28
29 global start;
30 global latency;
31
32 probe begin
33 {
34     println("Tracing... Hit Ctrl-C to end");
35 }
36
37 probe vfs.read
38 {
39     start[tid()] = gettimeofday_ns();
40 }
41
42 probe vfs.read.return
43 {
44     if (start[tid()]) {
45         fstypep = $file->f_path->dentry->d_inode->i_sb->s_type->name;
46         latency[kernel_string(fstypep)] <<<
47             gettimeofday_ns() - start[tid()];
48         delete start[tid()];
49     }
50 }
51
52 probe end
53 {
54     foreach (fs in latency) {
55         printf("    %s (ns):\n", fs);
56         print(@hist_log(latency[fs]));
57     }
58 }

```

This was a little tricky to write. I switched between the `vfs` tapset probe `vfs.read` and the kernel function `probe kernel.function("vfs_read")`, looking for an easy way to get the file system type. I didn't find it, nor did I find any example SystemTap scripts where anyone else had done it either.

So I pulled up the Linux kernel source for VFS and mounts and started reading. I figured out that I could get it by digging from the inode to the superblock, and then the file system type name. Maybe there is an easier way – maybe I missed a tapset function (I looked, but I know this particular system has an older version of SystemTap and its tapsets). This worked for me anyway.

Screenshot

The script traced for several seconds while a single threaded read workload hit a cached file. A few file system types are truncated from this screenshot to keep it short:

```
# ./vfsrlat.stp
Tracing... Hit Ctrl-C to end
^C
[..]

sockfs (ns):
value |----- count
1024 | 0
2048 | 0
4096 | @@@@@@@@@@@@@@@@@@ 24
8192 | @@ 2
16384 | 0
32768 | 0

proc (ns):
value |----- count
256 | 0
512 | 0
1024 | @@@@ 4
2048 | @ 1
4096 | @@@@@@@@@@@@@@ 14
8192 | 0
16384 | @@@@ 5
32768 | @@@ 3
65536 | @ 1
131072 | 0
262144 | @ 1
524288 | 0
1048576 | 0

devpts (ns):
value |----- count
256 | 0
512 | 0
1024 | @@@@ 4
2048 | @@@@@@@@@@@@@@@@@@ 22
4096 | @@@@@@@@@@ 8
8192 | @ 1
16384 | 0
32768 | 0
~
16777216 | 0
33554432 | 0
67108864 | @@@@@@@@@@ 10
134217728 | @@@@@@@@@@ 8
268435456 | @@@ 3
536870912 | @@ 2
1073741824 | 0
2147483648 | 0

debugfs (ns):
value |----- count
256 | 0
512 | 0
1024 | @ 2
2048 | @@@@@@@@@@@@@@@@@@ 88
4096 | @ 2
8192 | 0
16384 | 0

ext4 (ns):
value |----- count
256 | 0
512 | 0
1024 | 16
2048 | 17
4096 | 4
8192 | @@@@@@@@@@@@@@@@@@ 16321
16384 | 50
32768 | 1
65536 | 13
131072 | 0
262144 | 0
Pass 5: run failed. Try again with another '--vp 00001' option.
```

Great!

For this test, the ext4 file system latency is very fast and consistent. This also caught other file system types, including proc, showing their latency distributions. This is pretty interesting.

If you haven’t read these type of graphs before, the left column (“value”) shows the time ranges in nanoseconds. The right column (“count”) shows the number of occurrences at that range. The middle part shows an ASCII graph so that you quickly get an idea of the distribution.

As an example of reading the above output: The ext4 graph above (last) showed 16,321 reads that were between 8,192 and 16,383 nanoseconds (in other words: 16 thousand reads between 8 and 16 microseconds). The slowest read was in the 65 to 131 us range. Most of the I/O were consistantly fast: a distribution I like to see.

25% wider ASCII graphs

This example lets me describe some favourable features for SytemTap vs DTrace:

- The “devpts” graph has a “~” where lines were compressed, keeping more detail on the screen.
- While not shown here, the same statistic variable can be reprocessed via different actions other than @hist_log, which saves declaring multiple aggregations.

- Typos when dereferencing structs will print suggestions (see below).

```
# ./typo.stp
semantic error: unable to find member 'd_blah' for struct dentry (alternatives:
  d_flags d_seq d_hash d_parent d_name d_inode d_iname d_count d_lock d_op d_sb
  d_time d_fsdata d_lru d_u d_subdirs d_alias): operator '->' at ./typo.stp:45:34
  source:          fstypep = $file->f_path->dentry->d_blah->i_sb->s_type->name;
```

Pass 2: analysis failed. Try again with another '--vp 01' option.

- The width of the ASCII graph is 50 characters, which is 10 characters more than DTrace!

DTrace:

```
value  ----- Distribution ----- count
8192 |                                     0
```

SystemTap:

```
value |----- count
8192 |                0
```

I also noticed some unfavourable differences (although these are minor irritations):

- The ranges are padded by double “0” lines, when one would be sufficient.
- The output had to be printed in a foreach loop; DTrace will automatically generate this report.
- The graphs are not vertically aligned.

WARNING

I’ve found that running this script (like others) can crash the system.

I also hit the following when running it for the first time after boot:

```
# ./vfsrlat.stp
staprun:insert_module:112: ERROR: Error mapping '/tmp/stap0D0lk8/stap_37198ab8d430e8abf66eb925c924d56_4095.ko': Invalid argument
Error, 'stap_37198ab8d430e8abf66eb925c924d56_4095' is not a zombie systemtap module.
Retrying, after attempted removal of module stap_37198ab8d430e8abf66eb925c924d56_4095 (rc -5)
staprun:insert_module:112: ERROR: Error mapping '/tmp/stap0D0lk8/stap_37198ab8d430e8abf66eb925c924d56_4095.ko': Invalid argument
Pass 5: run failed. Try again with another '--vp 00001' option.
```

I cleared the ~/.systemtap directory and reran it, which worked.

There’s More...

I’ve encountered more issues with SystemTap, but this article is getting too long. I’ll summarize some of them here:

1. bad error messages
2. another –vp
3. slow startup time
4. no start message
5. multi user issues
6. stale cache
7. leaking modules
8. function(“”) crashes
9. no profile-997
10. more typing
11. non-intuitive
12. incomplete documentation
13. root escalation

1. Bad error messages

One of my earliest experiences was to Ctrl-C what seemed to be a stuck one-liner:

```
# stap -e 'global a; probe syscall.*.return { if (execname() == "httpd") { a <<< gettimeofday_ns() - @entry(gettimeofday_ns()); } } pr
^Csemantic error: symbol without referent: identifier 'ret' at /usr/share/systemtap/tapset/aux_syscalls.stp:1421:6
  source:          if (ret == 0)
```

Pass 2: analysis failed. Try again with another '--vp 01' option.

Oh? It’s complaining about a symbol from the aux_syscalls.stp tapset. But I didn’t edit that file, and I couldn’t see what was wrong by browsing the code.

It’s also asking for another “–vp 01” option. But I didn’t use one in the first place.

Trying the same one-liner again:

```
# stap -e 'global a; probe syscall.*.return { if (execname() == "httpd") { a <<< gettimeofday_ns() - @entry(gettimeofday_ns()); } } pr
^CPass 3: translation failed. Try again with another '--vp 001' option.
```

Huh? Now it’s complaining about “translation”. The one-liner seems fine – I don’t understand this error message, or the previous one. And it’s still asking for another “–vp” option, this time “001”, even though I didn’t use that option.

Trying again:

```
# stap -e 'global a; probe syscall.*.return { if (execname() == "httpd") { a <<< gettimeofday_ns() - @entry(gettimeofday_ns()); } } pr
^Cmake[1]: *** [/tmp/stap0NYN2b/stap_0e6f9ae5278258ba3028234e5a98831b_499668.o] Interrupt
```

```
make: *** [_module_/tmp/stapONYN2b] Interrupt
Pass 4: compilation failed. Try again with another '--vp 0001' option.
```

A third different error message for the *exact same one-liner*! This time “Interrupt” – was this my Ctrl-C?

Trying *again*, this time with the “time” command to see how long I’m waiting before the Ctrl-C:

```
# time stap -e 'global a; probe syscall.*.return { if (execname() == "httpd") { a <<< gettimeofday_ns() - @entry(gettimeofday_ns()); } }
^Cmake[1]: *** [/tmp/stap2y7BN1/stap_0e6f9ae5278258ba3028234e5a98831b_499668.o] Interrupt
make: *** [_module_/tmp/stap2y7BN1] Interrupt
Pass 4: compilation failed. Try again with another '--vp 0001' option.

real    0m11.495s
user    0m3.130s
sys     0m0.470s
```

11 seconds. Surely that’s enough time for this one-liner to begin.

I’ll try waiting even longer, and add the “-v” option to see what SystemTap is doing:

```
# time stap -e 'global a; probe syscall.*.return { if (execname() == "httpd") { a <<< gettimeofday_ns() - @entry(gettimeofday_ns()); } }
Pass 1: parsed user script and 72 library script(s) using 73484virt/21588res/2112shr kb, in 130usr/20sys/148real ms.
Pass 2: analyzed script: 789 probe(s), 12 function(s), 22 embed(s), 789 global(s) using 334408virt/132016res/3220shr kb, in 1800usr/37
Pass 3: translated to C into "/tmp/stapNRsa9/stap_0e6f9ae5278258ba3028234e5a98831b_499668.c" using 331972virt/136688res/8008shr kb, i
Pass 4: compiled C into "stap_0e6f9ae5278258ba3028234e5a98831b_499668.ko" in 21000usr/720sys/22306real ms.
Pass 5: starting run.
^CERROR: empty aggregate near identifier 'print' at <input>:1:134
WARNING: Number of errors: 1, skipped probes: 0
Pass 5: run completed in 0usr/200sys/3613real ms.
Pass 5: run failed. Try again with another '--vp 00001' option.

real    0m29.311s
user    0m23.620s
sys     0m1.450s
```

Now it’s complaining of an “empty aggregate”, an error message which finally makes sense: that aggregate is empty since there wasn’t activity to trace.

The “-v” output explained why I got the previous error messages: they were happening during the other passes by SystemTap to build up the program, and I was hitting Ctrl-C too early. But I still don’t know what the error messages actually mean. Suggesting that there were errors in a tapset file that I didn’t modify – simply because I hit Ctrl-C too soon – is a seriously confusing error message.

After this experience, I use “-v” all the time.

2. Another -vp

I still don’t understand the “-vp” error message, though. It even happens with:

```
# stap -e 'probe begin { println("Hello World!"); }'
Hello World!
^CPass 5: run failed. Try again with another '--vp 00001' option.
```

Hm. Ok, adding “another” one:

```
# stap --vp 00001 -e 'probe begin { println("Hello World!"); }'
Pass 5: starting run.
Hello World!
^CPass 5: run completed in 10usr/0sys/1776real ms.
Pass 5: run failed. Try again with another '--vp 00001' option.
```

It wants *another* one?

```
# stap --vp 00001 --vp 00001 -e 'probe begin { println("Hello World!"); }'
Pass 5: starting run.
Running /usr/bin/staprun -v /tmp/stapyRUftX/stap_8cc179a1da0850e296eed475a0c13f72_584.ko
Hello World!
^Cstapio:cleanup_and_exit:402 detach=0
stapio:cleanup_and_exit:419 closing control channel
staprun:remove_module:200 Module stap_8cc179a1da0850e296eed475a0c13f72_584 removed.
Pass 5: run completed in 10usr/20sys/3096real ms.
Pass 5: run failed. Try again with another '--vp 00001' option.
Running rm -rf /tmp/stapyRUftX
```

another??

```
# stap --vp 00001 --vp 00001 --vp 00001 -e 'probe begin { println("Hello World!"); }'
Pass 5: starting run.
Running /usr/bin/staprun -v -v /tmp/stapRXqPdd/stap_8cc179a1da0850e296eed475a0c13f72_584.ko
staprun:main:268 modpath="/tmp/stapRXqPdd/stap_8cc179a1da0850e296eed475a0c13f72_584.ko", modname="stap_8cc179a1da0850e296eed475a0c13f72_584.ko"
staprun:init_staprun:206 init_staprun
staprun:insert_module:60 inserting module
staprun:insert_module:79 module options: _stp_bufsize=0
staprun:check_signature:246 checking signature for /tmp/stapRXqPdd/stap_8cc179a1da0850e296eed475a0c13f72_584.ko
Certificate db /etc/systemtap/staprun permissions too loose
staprun:check_signature:258 verify_module returns 0
staprun:init_ctl_channel:34 Opened /sys/kernel/debug/systemtap/stap_8cc179a1da0850e296eed475a0c13f72_584/.cmd (3)
staprun:close_ctl_channel:53 Closed ctl fd 3
execing: /usr/lib/systemtap/stapio -v -v /tmp/stapRXqPdd/stap_8cc179a1da0850e296eed475a0c13f72_584.ko
stapio:main:37 modpath="/tmp/stapRXqPdd/stap_8cc179a1da0850e296eed475a0c13f72_584.ko", modname="stap_8cc179a1da0850e296eed475a0c13f72_584.ko"
stapio:init_stapio:317 init_stapio
stapio:init_ctl_channel:34 Opened /sys/kernel/debug/systemtap/stap_8cc179a1da0850e296eed475a0c13f72_584/.cmd (3)
stapio:stp_main_loop:494 in main loop
stapio:init_relayfs:238 initializing relayfs
stapio:init_relayfs:262 attempting to open /sys/kernel/debug/systemtap/stap_8cc179a1da0850e296eed475a0c13f72_584/trace0
```

```

stapio:init_relayfs:262 attempting to open /sys/kernel/debug/systemtap/stap_8cc179alda0850e296eed475a0c13f72_584/trace1
stapio:init_relayfs:268 ncpus=1, bulkmode = 0
stapio:init_relayfs:345 starting threads
stapio:stp_main_loop:571 probe_start() returned 0
Hello World!
^Cstapio:signal_thread:41 sigproc 2 (Interrupt)
stapio:stp_main_loop:505 signal-triggered 1 exit rc 0
stapio:stp_main_loop:563 got STP_REQUEST_EXIT
stapio:stp_main_loop:505 signal-triggered 1 exit rc 0
stapio:stp_main_loop:556 got STP_EXIT
stapio:cleanup_and_exit:402 detach=0
stapio:close_relayfs:362 closing
stapio:close_relayfs:381 done
stapio:cleanup_and_exit:419 closing control channel
stapio:close_ctl_channel:53 Closed ctl fd 3
stapio:cleanup_and_exit:432 removing stap_8cc179alda0850e296eed475a0c13f72_584
staprun:parse_modpath:335 modpath="/lib/modules/2.6.38-8-server/systemtap/stap_8cc179alda0850e296eed475a0c13f72_584.ko"
staprun:main:268 modpath="/lib/modules/2.6.38-8-server/systemtap/stap_8cc179alda0850e296eed475a0c13f72_584.ko", modname="stap_8cc179a1
staprun:init_staprun:206 init_staprun
staprun:remove_module:168 stap_8cc179alda0850e296eed475a0c13f72_584
staprun:init_ctl_channel:34 Opened /sys/kernel/debug/systemtap/stap_8cc179alda0850e296eed475a0c13f72_584/.cmd (3)
staprun:close_ctl_channel:53 Closed ctl fd 3
staprun:remove_module:188 removing module stap_8cc179alda0850e296eed475a0c13f72_584
staprun:remove_module:200 Module stap_8cc179alda0850e296eed475a0c13f72_584 removed.
Spawn waitpid result (0x822): 130
Pass 5: run completed in 0usr/30sys/4599real ms.
Pass 5: run failed. Try again with another '--vp 00001' option.
Running rm -rf /tmp/stapRXqPdd
Spawn waitpid result (0x802): 0

```

Ok, I am **not** adding any more “--vp 00001”s – this is enough to read just to debug Hello World!

Reading through the above output, I can see “Certificate db /etc/systemtap/staprun permissions too loose”. Maybe that’s the problem?

```

# ls -l /etc/systemtap/staprun
ls: cannot access /etc/systemtap/staprun: No such file or directory

```

I should ask the SystemTap engineers about this one...

I have noticed that if scripts call the “exit()” action, they don’t produce this error. Is every script supposed to call exit() at some point? From the SystemTap Beginners Guide:

SystemTap scripts continue to run until the exit() function executes. If the users wants to stop the execution of the script, it can interrupted manually with Ctrl+C.

It doesn’t say I need the exit() on Ctrl-C. Neither the Beginners Guide nor the Language Reference explain the --vp, either.

3. Slow startup time

Some of the previous errors occurred when I hit Ctrl-C too soon, interrupting the SystemTap build process. It was easy to do, since for the time that one-liner finally did work, I hit Ctrl-C as soon as the script began running and got:

```

real    0m29.311s
user    0m23.620s
sys     0m1.450s

```

Which is almost 30 seconds – just for the one-liner to *begin* tracing.

The “-v” output includes times for each stage:

```

Pass 1: parsed user script and 72 library script(s) using 73484virt/21588res/2112shr kb, in 130usr/20sys/148real ms.
Pass 2: analyzed script: 789 probe(s), 12 function(s), 22 embed(s), 789 global(s) using 334408virt/132016res/3220shr kb, in 1800usr/37
Pass 3: translated to C into "/tmp/stapNRsa9/stap_0e6f9ae5278258ba3028234e5a98831b_499668.c" using 331972virt/136688res/8008shr kb, i
Pass 4: compiled C into "stap_0e6f9ae5278258ba3028234e5a98831b_499668.ko" in 21000usr/720sys/22306real ms.
Pass 5: run completed in 0usr/200sys/3613real ms.

```

The longest stage is the compilation stage, taking over 22 seconds.

DTrace usually starts up in a few seconds, and in 30 seconds I may be on my third one-liner. With SystemTap, I’m still waiting for the first to gather data. The startup time needs to be greatly improved.

4. No start message

A minor irritation is that without the “-v” option, there is no message to indicate that SystemTap has begun tracing. And since, as just mentioned, it can take up to 30 seconds, you can spend a long time wondering if anything is happening yet. A number of times I’ve hit Ctrl-C only to find it hadn’t began tracing yet, and I had to start all over again (a reminder to use “-v”).

DTrace has a default message, which is suppressed when you use the “quiet” option. If I do use the quiet option, I always try to print a begin message of some sort, either “Tracing... Hit Ctrl-C to end” or column headers. That lets the end user know that tracing has begun.

5. Multi user issues

With DTrace, multiple users can usually run the same one-liners or scripts at the same time. Here’s what happened when I tried with SystemTap:

```

window1# stap -e 'probe begin { println("Hello World!"); }'
Hello World!

window2# stap -e 'probe begin { println("Hello World!"); }'
Error inserting module '/tmp/stapYJ8JXf/stap_8cc179alda0850e296eed475a0c13f72_584.ko': File exists
Error, 'stap_8cc179alda0850e296eed475a0c13f72_584' is not a zombie systemtap module.
Retrying, after attempted removal of module stap_8cc179alda0850e296eed475a0c13f72_584 (rc -3)
Error inserting module '/tmp/stapYJ8JXf/stap_8cc179alda0850e296eed475a0c13f72_584.ko': File exists

```

Pass 5: run failed. Try again with another '--vp 00001' option.

Not only did it not allow two users to run Hello World, but it's also example of another confusing error message: "not a zombie systemtap module"?

It appears that systemtap takes a hash (md5?) of the script, and uses that as a unique identifier. You'd think the identifier would include a process ID, but I'm guessing it doesn't as part of the SystemTap cache system (under ~/.systemtap) that caches previously built modules to reduce runtime. I'm guessing (hoping) that you disable the cache system, multiple users can run the same script. Which reminds me:

6. Stale cache

Sometimes SystemTap gets into a totally weird state where scripts and one-liners that worked a moment ago are all failing. The remedy is usually to delete the entire ~/.systemtap directory, which contains cached modules.

And, not to belabor the point, but the error messages for this issue are also totally confusing. The following:

```
error: 'struct hrtimer' has no member named 'expires'
```

Wasted two hours of my life. I picked through the runtime C code to comprehend what it meant. The ~/.systemtap directory cache was stale.

7. Leaking modules

I don't know how this happened:

```
# lsmod
Module                Size  Used by
dtracedrv             3688140  0
stap_cf432894d9effefc870913a509ff9d5b_7636  50928  0
ppdev                 17113  0
psmouse              73535  0
parport_pc           36959  1
serio_raw            13166  0
i2c_piix4            13303  0
lp                   17789  0
parport              46458  3 ppdev,parport_pc,lp
floppy               74120  0
```

And I don't know what that stap module is for. There's no stap processes running right now.

8. function("**") crashes

This bug was filed over 5 years ago, and last updated over 2 years ago:

[Bug 2725](#) "function("**") probes sometimes crash & burn"

It's not clear if this is inherently difficult to fix with SystemTap, or not considered a priority to fix. With DTrace, I use this functionality often. It's limiting to have to keep avoiding it in SystemTap due to this known bug.

As an example: last Friday I was asked to help with a network issue, where a server had lost connectivity on one of its 10 GbE interfaces. Various possibilities had already been checked, including swapping the optics and fibre with known-to-be-good ones, without success. The driver was checked from the usual OS ways, dumping various error counters and messages, and it seemed fine. I was pulled in to check the kernel, specifically: is there a bug in the driver? Is it doing something abnormal, or does it have an abnormal internal state that isn't visible via the OS tools?

I hadn't DTraced this driver in a while, and certainly didn't remember what functions it used. Listing them showed there were hundreds available, but that's not much use unless they are being *called* so that I can trace them. I used wildcarding to see which of the hundreds were being called, which turned out to be only several, and then traced them to reveal internal state. I did this in parallel with a working server, and all the extra details I was digging out proved to be the same. I then moved to other areas of the TCP/IP stack by increasing the wildcard to eventually match everything as a list of candidates. But I wasn't finding a smoking gun, making the theory of a bug in the driver looking less likely. More physical analysis in the datacenter found that the optics in the network switch (not the server) were dead – which was missed by an earlier search. (Yes, this might have been a better example if Dynamic Tracing actually found the issue directly; instead, it's an example of Dynamic Tracing finding the *absence* of an issue).

With SystemTap and this bug, I couldn't do wildcarding as part of this analysis, as it leads to system crashes. While one network interface was down, others were still up, and this was production. Instead, I'd read through the list of available probes and use a trial-and-error approach: tracing one to see if it was being used, then moving on to the next one. With SystemTap startup times being slow (as mentioned earlier) and hundreds of functions available, this could have taken over an hour, instead of the minutes it did take. This was also a production emergency where other company experts were pulled in to help: so this wouldn't have just wasted my time – it'd waste theirs, too.

9. no profile-997

While I'm on the topic of probes that don't work, here is another that I use frequently – almost *every* time I do DTrace analysis. This samples kernel stack traces at 997 Hertz, to get a quick idea of where the CPU time is spent in the kernel:

```
# dtrace -n 'profile-997 { @[stack()] = count(); }'
dtrace: description 'profile-997 ' matched 1 probe
[...]
    unix`page_exists+0x67
    genunix`vmu_calculate_seg+0x3ac
    genunix`vmu_calculate_proc+0x259
    genunix`vmu_calculate+0xd2
    genunix`vm_getusage+0x23a
    genunix`rusagesys+0x64
    unix`sys_syscall+0x17a
    213
[...]
```

I ran this on an older system and caught a hot codepath I'm quite familiar with, caused by memory capping software (rcapd) scanning memory pages. I remember going from this sampled stack to Dynamic Tracing by picking some of those functions and tracing entry to return time: useful data that helped us comprehend the magnitude of this issue in production, which was later fixed (thanks Jerry!).

Apart from sampling kernel stacks, by changing “stack()” to “ustack()” (usually with a pid or execname predicate) this can sample application user-level stacks too. Many more customizations are possible.

The 997 Hertz is a deliberate choice so that the sampling doesn’t run in lockstep with a kernel activity, resulting in over-samples or under-samples. I’ll sometimes use 1234 Hertz, too.

Unfortunately, this didn’t work for me with SystemTap:

```
# stap -e 'global s; probe timer.hz(997) { s[backtrace()] <<< 1; }
  probe end { foreach (i in s+) { print_stack(i); printf("\t%d\n", @count(s[i])); } exit(); }'
^C      12220
Pass 5: run failed. Try again with another '--vp 00001' option.
```

It seems that backtrace() doesn’t work with arbitrary Hertz. Even if it did work, it looks like the timer.hz probe only fires on one CPU, instead of profiling across them all as desired.

The timer.profile probe does fire on all CPUs, and backtrace() does work. However, it only runs at 100 Hertz:

```
# stap -e 'global s; probe timer.profile { s[backtrace()] <<< 1; }
  probe end { foreach (i in s+) { print_stack(i); printf("\t%d\n", @count(s[i])); } exit(); }'
^C
[...]
      14
0xffffffff815d8229 : _raw_spin_unlock_irqrestore+0x19/0x30 [kernel]
0xffffffff81096525 : clockevents_notify+0x45/0x170 [kernel]
0xffffffff81335da0 : intel_idle+0xe0/0x120 [kernel]
0xffffffff814b698a : cpuidle_idle_call+0xaa/0x1b0 [kernel]
0xffffffff8100a266 : cpu_idle+0xa6/0xf0 [kernel]
0xffffffff815cfb6d : start_secondary+0xbc/0xbe [kernel]
0x0

      18
0xffffffff81335d8a : intel_idle+0xca/0x120 [kernel]
0xffffffff814b698a : cpuidle_idle_call+0xaa/0x1b0 [kernel]
0xffffffff8100a266 : cpu_idle+0xa6/0xf0 [kernel]
0xffffffff815cfb6d : start_secondary+0xbc/0xbe [kernel]
0x0

     1434
Pass 5: run failed. Try again with another '--vp 00001' option.
```

10. More typing

The previous one-liners reminded me of another issue. Here they are side by side:

```
dtrace -n 'profile-997 { @[stack()] = count(); }'

stap -e 'global s; probe timer.hz(997) { s[backtrace()] <<< 1; }
  probe end { foreach (i in s+) { print_stack(i); printf("\t%d\n", @count(s[i])); } exit(); }'
```

SystemTap was created after DTrace, adding to the syntax the term “probe” and forcing the predeclaration of variables. This hasn’t improved usability for me. I don’t know why they wouldn’t just support the DTrace syntax, or at least improve upon it. Another baffling change was to lowercase the “BEGIN” probe to be “begin”, contrary to not only DTrace but also to awk (and Perl) from where it originated. Was it necessary to sacrifice usability just to be different?

The rest of the above one-liner was long to make the stack traces readable. Without the end action, the default output of SystemTap is:

```
# stap -e 'global s; probe timer.profile { s[backtrace()] <<< 1; }'
^Cs["0xffffffff81335d8a 0xffffffff814b698a 0xffffffff8100a266 0xffffffff815cfb6d 0x0"] @count=0x5347 @min=0x1 @max=0x1 @sum=0x5347 @av
s["0xffffffff815d8229 0xffffffff81096525 0xffffffff81335da0 0xffffffff814b698a 0xffffffff8100a266 0xffffffff815cfb6d 0x0"] @count=0x4f
s["0xffffffff81335d8a 0xffffffff814b698a 0xffffffff8100a266 0xffffffff815bb095 0xffffffff81ae1c90 0xffffffff81ae1388 0xffffffff81ae145
s["0xffffffff814b68a2 0xffffffff814b698a 0xffffffff8100a266 0xffffffff815cfb6d 0x0"] @count=0xa @min=0x1 @max=0x1 @sum=0xa @avg=0x1
s["0xffffffff814b68a6 0xffffffff814b698a 0xffffffff8100a266 0xffffffff815cfb6d 0x0"] @count=0x4 @min=0x1 @max=0x1 @sum=0x4 @avg=0x1
s["0xffffffff81165430 0xffffffff811655a1 0xffffffff8100bfc2 0x293 0xd000000000000002 0x36d0000000000000 0x3a36d00000000000 0x973a36d00
s[""] @count=0x1 @min=0x1 @max=0x1 @sum=0x1 @avg=0x1
Pass 5: run failed. Try again with another '--vp 00001' option.
```

These are just stack frame addresses, which are not human readable. By default, DTrace prints the stack trace aggregation with human readable translations. As a one-liner I use often, this reduces typing and improves useability.

I’ve found that SystemTap often requires more typing to achieve the same result as DTrace. There are some exceptions where SystemTap is shorter, which reminds me of the next topic:

11. Non-intuitive

Here’s an example where SystemTap is slightly less typing than DTrace:

```
dtrace -n 'syscall::read:entry { self->s = timestamp; }
  syscall::read:return /self->s/ { @ = quantize(timestamp - self->s); self->s = 0; }'

stap -e 'global s; probe syscall.read:return {
  s <<< gettimeofday_ns() - @entry(gettimeofday_ns()); } probe end { print(@hist_log(s)); }'
```

This is a basic one-liner, tracing the time for the read() system call. Let me try to describe what each of these do:

With DTrace: the entry to the read() syscall is traced and a timestamp is saved into the self->s variable, associated with the current thread. The return for the read() syscall is also traced, which checks for the presence of the self->s variable to ensure the entry was seen and the timestamp recorded. If present, a delta time is calculated and saved into a quantize aggregation. The self->s variable is then set to zero, clearing that variable.

With SystemTap: the return to the read() syscall is traced, and the delta time between the current timestamp and the timestamp from the entry to the read() syscall is saved into the “s” statistic. The entry timestamp was retrieved by passing an action into an @entry() clause, which automatically instantiates an entry probe to execute the action, and returns the value. Storage for the @entry() clause result is implicitly created, checked on the return probe (I think), and then freed after use (I think). This is a shortcut to avoid

explicitly probing “syscall.read.entry” and the timestamp variable. The end probe uses the `@hist_log()` action to print a logarithmic histogram.

Dynamic Tracing is hard to learn. The implicit behaviour of the `@entry()` action makes this more difficult to follow than DTrace’s simple explicit version. It also complicates an important task when using and reusing these scripts: scanning the probes to gauge the overhead. Single probe scripts have much less overhead than pairs of probes that allocate and free a variable between them. This difference is clear with the DTrace version, but not the SystemTap version.

This is just a minor point, since the solution should be to simply not use this shortcut.

12. Incomplete documentation

I mentioned “I think” a couple of times in my description above for `@entry`. The `@entry()` action is not explained in any of the documentation I’ve searched, including the reference manual:

SystemTap Language Reference

@entry 0 of 0

While there are a lot of reference docs for SystemTap, this isn’t the first time I’ve found something missing. It’s also something that has some implicit behaviour, for which I’d really like to read the documentation. It is used in the example scripts.

13. Root escalation

This did get fixed in SystemTap version 1.6, so make sure you are on that version:

[CVE-2011-2502](#) systemtap: insufficient security check when loading uprobes kernel module.

Which provided a root escalation from any user in the stapusr group via a one-liner.

I installed systemtap via apt-get a few days ago, and got version 1.3 (other times I compile from the latest source). So there may still be some vulnerable versions out there (although, Debian do backport security fixes, so their 1.3 may have this fix – I don’t know).

Must stop...

I could go on, but I’ve already spent more time than I should on this, and yet I’ve only scratched the surface. There are many more scripts and tapsets and functionality I haven’t mentioned yet, including user-land tracing. Exploring (and wrestling) with these could become a day job for the next several months.

I’d summarize my SystemTap experience so far as “painful”, due to:

- **installation**: not a default install, and not easy to get running on the systems I tried
- **reliability**: system crashes, intermittent failures (kernel read fault errors)
- **useability**: slow startup time, confusing error messages, extra typing

I am using SystemTap to solve real issues (including analyzing performance differences between OSes that do and don’t have DTrace), so it has brought some amount of value, but at the cost of countless hours of work.

By detailing these experiences I’ve probably helped the SystemTap project in some way (similar to the feedback I sent the DTrace engineers during its development), and maybe everything I’ve listed will be fixed in a year or two from now. But, the future could also be very different.

Past, Present, Future

The SystemTap project has been running for over six years, with contributions from about [one hundred engineers](#). They’ve completed a lot of work on an incredibly difficult engineering problem – Dynamic Tracing – and made a good effort.

The years ahead may be very different though, now that Oracle has [announced](#) that they are porting DTrace to Linux. Their current version may be an even worse [experience](#) than SystemTap. However, it’s early days. What’s interesting is that they are the first Linux vendor to believe that it is both possible and *should* be done. Based on that precedent, other Linux vendors may follow suit.

It’s possible that SystemTap is, and has always been, on the wrong trajectory. The slow startup times caused by compilation – an architectural decision by SystemTap – may never be completely fixed. [Bryan](#) understands this better than I do from his work on DTrace, and wrote [recently](#) about the SystemTap compilation stages:

It would be incorrect to say that we considered this and dismissed it for DTrace, because never for a moment would we have considered something so obviously asinine. I elaborated on our thinking in <http://dtrace.org/blogs/bmc/2005/07/19/dtrace-safety/>; while that post never mentions SystemTap by name, it is very much aimed at what I perceived to be the reckless design decisions that they had made.

When DTrace was released in 2005 as open source, I remember telling Sun sales folk that they probably had three months of competitive advantage before Linux would have the code up and running. They weren’t too happy about that, and there was some resentment at Jonathan for open sourcing the crown jewels of Solaris 10. Little did we know what would happen in the years to come.

I’ll still be using SystemTap in the short term future while DTrace does not exist on the Linux systems I have, in the lab. My next tasks are to get kdump to collect kernel crash dumps, try other Linux distributions, upgrade to the latest SystemTap, and take better notes (version numbers of everything).

Conclusion

Dynamic plus Static Tracing – with the ability to write custom tools that summarize data in-kernel and are safe for production use – is the greatest performance technology of my lifetime, and should be remembered as one of the greatest inventions in computing. The scope, depth and utility of this observability is mind blowing.

In this article I explored the SystemTap version of this technology, and ran into various issues including system crashes. In summary, I would not recommend the current version of SystemTap for production use at all, at least in the production environments that I’m familiar with. I also discovered that a key example from the documentation, `disktop.stp`, doesn’t make sense – raising questions on how well SystemTap is being tested or used. I have mentioned various other issues, including confusing error messages and slow startup times, that make the tool difficult to use.

I'm particularly worried about people getting the wrong impression of Dynamic Tracing based on SystemTap. SystemTap is *not* DTrace, and looks as if it'll be a while yet before it is even close. That is, if it is even on the right trajectory.


During the past seven years, DTrace has become an essential part of my day job in performance analysis, and it's unthinkable to return to a time without it. I've hoped that should my career move to Linux (anything's possible), I'd learn the SystemTap syntax quickly, and be able to carry on with the Dynamic Tracing methodology that I've been perfecting for years. That now seems much harder than I hoped.

I'm expecting a response from the SystemTap developers will be: "it's better on Red Hat Enterprise Linux". This may certainly be true. But if I'm going to switch OSes to do Dynamic Tracing, then I'm switching to an OS with DTrace.

Posted on October 15, 2011 at 4:07 pm by Brendan Gregg · [Permalink](#)
In: [Linux](#) · Tagged with: [linux](#), [systemtap](#)

15 Responses

[Subscribe to comments via RSS](#)

1.  Written by Brendan Gregg
on October 15, 2011 at 4:12 pm
[Permalink](#)


I'm hoping for comments that are useful, constructive and technical. Real names please, and preferably company name as well. As Adam said on a recent post after being slashdotted: "If you want to wax vitriolic about licensing, do it somewhere else. If you want to talk about various technologies, great." :-)

2.  Written by Chris
on October 16, 2011 at 4:54 am
[Permalink](#)

Wow, SystemTap was on my todo list, but now you've completely put me off it. Right from the start, that 'try again with' type message is exactly the sort of thing I hate about so many Linux implementations. I want a production safe well instrumented tool to work with, but this sort of thing makes it just seem like you're prodding at the kernel with a stick.

I'm just in the process of testing RHEL 6.1 and it amazes me that an enterprise ready distribution like RedHat still ships binaries with no man pages, where there are 2-3 different ways to achieve the same thing (LDAP authentication being something I've recently been configuring), and fairly major changes (example, kdump memory sizing) merit little or no documentation. Blog posts are not documentation!

The nature of Linux development is always going to make it difficult to manage this, but am I unreasonable to expect at least a little more?

- o  Written by Joe
on October 17, 2011 at 1:36 pm
[Permalink](#)

I have to agree. I would NEVER run some tool on my production machines that "Could" crash it...

Looks like the tools are worth while, but I will wait for a ppa before I install and test :)

Love the technology, though!


Good work!
Joe

3.  Written by Justin
on October 16, 2011 at 5:23 pm
[Permalink](#)

WARNING: never-assigned local variable 'write' (alternatives: bytes process cmd userid parent action io_stat device read_bytes write_bytes __devnames): identifier 'write' at ./disktop.stp:45:17
source: if(read_bytes+write+bytes) {

This seems to indicate that the line should be

if(read_bytes+write_bytes) {

- o  Written by Brendan Gregg
on October 16, 2011 at 11:40 pm
[Permalink](#)

Oh, you are right, thanks! I assumed that was just because I didn't do a write workload, but it's actually just a bug in the code:

```
if(read_bytes+write+bytes) {
```

which was fixed for the Red Hat Enterprise Linux documentation version:

```
if(read_bytes+write_bytes) {
```

which also rewrote the script to use the vfs tapset. It'd be nice if they updated the script on the SystemTap wiki when updating that version (if not to use the tapset, at least to fix that bug). Or at the very least, had a version number more precise than "2007", to help steer me to the latest version. (This also further reinforces the feeling that no one has really tried these scripts before.)



4. Written by [cs](#)
on October 17, 2011 at 12:23 am
[Permalink](#)

Additionally, you might find this thread amusing:
<http://lwn.net/Articles/301285/>

Choice quote:

>Ok, first peeve. If you tell me one more time to use "more" -v options, I'm going to beat you with a rusty tire-iron. HOW MANY MORE???? The message should say exactly the option that will give information about the kind of failure that just happened.



5. Written by [Frank Ch. Eigler](#)
on October 17, 2011 at 7:06 am
[Permalink](#)

Brendan, please see an extended response at my blog:
http://web.elastic.org/~fcbe/blog2/archive/2011/10/17/using_systemtap_better



6. Written by [Hugh Brown](#)
on October 17, 2011 at 10:27 am
[Permalink](#)

(Sorry, just re-read the "real names please" part...please feel free to delete my earlier, pseudonymous comment. Oh, and I'm a sysadmin for a Canadian university...no commercial interest on either side of the fence, just an interested user.)

I love Linux dearly, and have found Solaris frustrating in many ways — but oh lord, I love DTrace. I miss it a lot. I really, really wish that either it was available on Linux already, or that something better than SystemTap was here. As it is, ST just seems awful.

Thanks for taking the time to not only evaluate SystemTap, but to write up your experience in a thoroughly helpful fashion. I would have been so angry by the end of it that...well, it wouldn't have been nice. I hope the SystemTap engineers take notice.



7. Written by [Adam Leventhal](#)
on October 17, 2011 at 7:26 pm
[Permalink](#)

Brendan, it's great to see how the other half live. I especially like the part about the profile probes — if it's hard to do something useful, I'm sure useless/misleading is good enough! And for the sake of fairness, the DTrace docs — while you've made the world much better — still sometimes require opening `dt_open.c` for confirmation.



8. Written by [mango](#)
on October 18, 2011 at 3:26 am
[Permalink](#)

systemtap looks like crap.

However, you can use existing tools to check what uses your disk IO:

```
# man iotop
```

iotop – simple top-like I/O monitor



9. Written by [Nigel Smith](#)
on October 18, 2011 at 3:41 pm
[Permalink](#)

I think one of the big problems for SystemTap is that, AFAIK, they don't have a (clone of) Brendan Gregg, to develop and test scripts, blog about it, in-depth,

consistently over the years, write books, etc. Brendan, your contribution to promoting DTrace has been immense, and SystemTap just does not seem to have anyone on your level (AFAIK).

With Solaris, FreeBSD, and derivatives, DTrace comes as part of the kernel. If a kernel change should break DTrace then everyone would consider the kernel as broken, and it would be quickly fixed.

AFAIK, SystemTap is being developed without that close coupling to the Linux kernel. If the kernel changes, and breaks SystemTap, well I don't think Linus would delay releasing the latest and greatest. It would be up to the SystemTap people to fix the problems, as & when.

And SystemTap just does not seem to have got close enough to being an serious & professional tool, getting wide enough adoption, for it to be considered an essential component of the Linux system.

Does SystemTap a standard and robust set of scripts with which to test and validate in combination with a given Linux kernel release?

As most of the development of SystemTap seems to be by RedHat, I would expect you would have more success with one of their distributions, like RHEL or Fedora.



10. Written by Keith Chambers
on October 30, 2011 at 10:36 pm
[Permalink](#)

I've used SystemTap in production with RHEL5 and RHEL6 for the past 2 years. I'm not a proper c developer but do write my own low level scripts that often bypass the standard SystemTap tapsets. I haven't run in to any stability issues.

I've never used Dtrace. In fact, I only discovered SystemTap while searching for a Dtrace equivalent for Linux. Dtrace could very well be better. But under RHEL SystemTap is pretty decent.



11. Written by [Nico Williams](#)
on November 1, 2011 at 7:36 pm
[Permalink](#)

FYI, when I started working on Linux as part of working on Lustre at Sun I tried ST, and gave up after just a few days. Your experience obviously went much further and deeper than mine, but it wasn't better for it. Paul Fox's DTrace port -early days though it was- gave me much better results even though I was necessarily stuck on too old a kernel version (I got to where FBT worked pretty well as long as I was careful enough to not probe toxic symbols). Even with DTrace running, the lack of frame pointers in the typical Linux kernel is rather frustrating. No DTrace, no mdb... how poor cousin Linux is!

The whole idea of writing probe action code in C, after the seeing the Truth (DTrace's byte-compiled/evaluated probe action code) is... well, there are no words to describe how dumb that is. I'd have preferred using a heavy-duty VM (JVM, V8, some LISP — anything) over object code!!! Though any generic VM would still have been an incorrect choice: it's critical to ensure that looping is not possible, and that's precisely what DTrace does and can do by having its own VM.



12. Written by [Nico Williams](#)
on November 1, 2011 at 7:37 pm
[Permalink](#)

(Whenever I found toxic symbols I sent Paul e-mail. The problem was finding those toxic symbols.)



13. Written by Heron Peak
on December 16, 2011 at 11:38 am
[Permalink](#)

Generally, simply removing the \$HOME/.systemtap directory can clear up "ERROR: Build-id mismatch" problems.

[Subscribe to comments via RSS](#)

[« Previous post](#)
[Next post »](#)

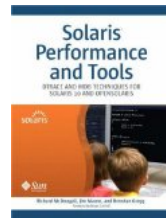
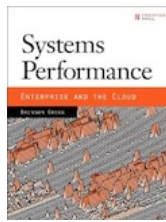
• Recent Posts

- [A New Challenge](#)
- [Another 10 Performance Wins](#)
- [Benchmarking the Cloud](#)
- [Cloud Performance Training](#)
- [Systems Performance: available now](#)
- [Open Source Systems Performance](#)
- [The TSA Method](#)
- [Control T for TENEX](#)
- [The USE Method: Unix 7th Edition Performance Checklist](#)
- [The USE Method: FreeBSD Performance Checklist](#)
- [The USE Method: Mac OS X Performance Checklist](#)

- [Memory Leak \(and Growth\) Flame Graphs](#)

FOLLOW ME ON [twitter](#)

• My Books



• Tags

[7410](#) [analytics](#) [art](#) [benchmarking](#) [book](#) [cloud](#) [cloud analytics](#) [CPI](#) [dtrace](#) [example](#) [experimental](#) [filesystem](#) [frequencytrail](#) [heatmaps](#) [illumos](#) [iscsi](#) [javascript](#) [joyent](#) [L2ARC](#) [latency](#) [limits](#) [linux](#) [macosx](#) [methodology](#) [mysql](#) [NAS](#) [nfs](#) [off-cpu](#) [omnios](#) [performance](#) [personal](#) [PICs](#) [pid provider](#) [slides](#) [SLOG](#) [smartos](#) [solaris](#) [SSD](#) [statistics](#) [talk](#) [testing](#) [usemethod](#) [video](#) [visualizations](#) [ZFS](#)

• People

- [Adam Leventhal](#) [dtrace.org](#)
- [Brendan Gregg](#) [dtrace.org](#) (professional)
- [Brendan Gregg](#) [blogspot](#) (personal)
- [Bryan Cantrill](#) [dtrace.org](#)
- [Dave Pacheco](#) [dtrace.org](#)
- [Deirdré Straughan](#) [beginningwithi.com](#)
- [Jim Mauro](#) [sun.com](#)
- [Robert Mustacchi](#) [dtrace.org](#)

• Links

- [Brendan's homepage](#)
- [Joyent](#)
- [SolarisInternals](#)

• Meta

- [Log in](#)
- [Entries RSS](#)
- [Comments RSS](#)
- [WordPress.org](#)
- Copyright 2013 Brendan Gregg, all rights reserved

© [Brendan's blog](#).

Powered by [WordPress](#) and [Grey Matter](#).