**Using DTrace to find block sizes of ZFS, NFS and iSCSI**

A member of our Storage Team, Cloud Engineer Goran Mekic, told me that documentation and books on DTrace were not terribly helpful to him in finding out what function he should trace. Goran said that he would have been extremely happy if he could refer to a blog post like the one below before starting with his task, so he offered to share the process because it is more important than the measurement itself.

**So in that spirit of sharing – let’s walk through Goran’s instructions on finding block sizes of ZFS, NFS, and iSCSI using DTrace…**

The first step is finding the right probe. Sometimes you can just filter existing probes hoping the output will contain the right one.

dtrace -l | grep zfs | grep write  
...  
76274 fbt zfs zfs\_write entry  
76275 fbt zfs zfs\_write return  
...

Let's find that function in the source code. To reduce time of search it is recommended to use ripgrep instead grep.

cd /usr/src  
rg '^zfs\_write'  
sys/contrib/openzfs/module/zfs/zfs\_vnops.c  
406:zfs\_write(znode\_t \*zp, zfs\_uio\_t \*uio, int ioflag, cred\_t \*cr)

sys/contrib/openzfs/module/os/linux/zfs/zfs\_vnops\_os.c  
344:zfs\_write\_simple(znode\_t \*zp, const void \*data, size\_t len,

sys/contrib/openzfs/module/os/freebsd/zfs/zfs\_vnops\_os.c  
648:zfs\_write\_simple(znode\_t \*zp, const void \*data, size\_t len,

Note that I searched for files where "zfs\_write" starts at the begining of the line. That is because FreeBSD coding style(9) is the following:

/\*  
 \* Write the bytes to a file.  
 \*  
 \* IN: zp - znode of file to be written to.  
 \* uio - structure supplying write location, range info,  
 \* and data buffer.  
 \* ioflag - O\_APPEND flag set if in append mode.  
 \* O\_DIRECT flag; used to bypass page cache.  
 \* cr - credentials of caller.  
 \*  
 \* OUT: uio - updated offset and range.  
 \*  
 \* RETURN: 0 if success  
 \* error code if failure  
 \*  
 \* Timestamps:  
 \* ip - ctime|mtime updated if byte count > 0  
 \*/  
int  
zfs\_write(znode\_t \*zp, zfs\_uio\_t \*uio, int ioflag, cred\_t \*cr)  
{  
 ...  
}

Searching only for lines that start with zfs\_write reduces the number of entries returned by ripgrep. We see that the second argument, or args[1] in DTrace, is zfs\_uio\_t. In most cases when you see \_t type you are actually dealing with "typedef struct <name> { ... } <name>\_t" kind of construct, hence the following command to find it.

rg '\} zfs\_uio\_t'  
sys/contrib/openzfs/lib/libspl/include/sys/uio.h  
76:} zfs\_uio\_t;

sys/contrib/openzfs/include/os/linux/spl/sys/uio.h  
83:} zfs\_uio\_t;

sys/contrib/openzfs/include/os/freebsd/spl/sys/uio.h  
44:} zfs\_uio\_t;

When we peak inside FreeBSD's uio.h we find the following:

typedef struct zfs\_uio {  
 struct uio \*uio;  
} zfs\_uio\_t;

Then we find "struct uio":

rg 'struct uio \{'  
share/man/man9/uio.9  
41:struct uio {

share/doc/papers/fsinterface/[fsinterface.ms](http://fsinterface.ms/)724:struct uio {

share/doc/papers/fsinterface/slides.t  
116:struct uio {

sys/sys/uio.h  
55:struct uio {

In the last file, this is the definition of the structure:

struct uio {  
 struct iovec \*uio\_iov; /\* scatter/gather list \*/  
 int uio\_iovcnt; /\* length of scatter/gather list \*/  
 off\_t uio\_offset; /\* offset in target object \*/  
 ssize\_t uio\_resid; /\* remaining bytes to process \*/  
 enum uio\_seg uio\_segflg; /\* address space \*/  
 enum uio\_rw uio\_rw; /\* operation \*/  
 struct thread \*uio\_td; /\* owner \*/  
};

What we want is the uio\_resid, as per comment that's the number of bytes. So our probe looks like this:

fbt:zfs:zfs\_write:entry  
{  
 @zfsw = quantize(args[1]->uio->uio\_resid);  
}

This will give us distribution of byte sizes, and for a test machine which should otherwise be idle, the only process writing stuff should be

dd if=/dev/random of=/tmp/output.dd bs=16k

By using write and random data we ensure that no cache is going to interfere with our measurement. The output of probe while running dd is the following:

value ------------- Distribution ------------- count  
 0 | 0  
 1 | 8  
 2 | 0  
 4 | 0  
 8 | 0  
 16 | 0  
 32 | 2  
 64 | 1  
 128 | 0  
 256 | 0  
 512 | 2  
 1024 | 1  
 2048 | 0  
 4096 | 0  
 8192 | 0  
 16384 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 74378  
 32768 | 0

As there are other processes writing on the same machine, we get more than just 16k writes. Let's try to do the same for NFS. First, we need to set up NFS.

cat /etc/rc.conf  
mountd\_enable="YES"  
mountd\_flags="-r"  
nfs\_server\_enable="YES"  
nfsv4\_server\_enable="YES"  
rpcbind\_enable="YES"

You will need to (re)start mountd, nfsd and rpcbind services, create ZFS dataset and share it over NFS:

service nfsd restart  
service rpcbind restart  
service mountd restart

zfs create zroot/var/nfs  
zfs set sharenfs=on zroot/var/nfs  
mount -t nfs -o rsize=8192,wsize=8192 127.0.0.1:/var/nfs /mnt

As default ZFS block size is 4kB, I chose 8kB in NFS client to see what will happen. For finding the probe let's try the same technique we did for ZFS:

dtrace -l | grep nfs | grep write  
...  
51671 fbt kernel nfsvno\_write entry  
51672 fbt kernel nfsvno\_write return  
...

I must point out that we were really lucky searching for the probes. If luck is on your side, embrace it with both hands and forget about more elaborate techniques! Now, let's see what the actual function looks like.

rg '^nfsvno\_write'  
sys/fs/nfsserver/nfs\_nfsdport.c  
1124:nfsvno\_write(struct vnode \*vp, off\_t off, int retlen, int \*stable,

Third argument, or args[2] in DTrace, looks like what we need. Let's try this:

fbt:kernel:nfsvno\_write:entry  
{  
 @nfsw = quantize(args[2]);  
}

And we get the following distribution:

value ------------- Distribution ------------- count  
 4096 | 0  
 8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 63908  
 16384 | 0

Great! Because no matter the ZFS prefered block size of 4kB and dd bs=16k parameter, NFS respects the mount option wsize=8192. If we combine the probes into a DTrace script like this:

fbt:zfs:zfs\_write:entry  
{  
 @zfsw = quantize(args[1]->uio->uio\_resid);  
}

fbt:kernel:nfsvno\_write:entry  
{  
 @nfsw = quantize(args[2]);  
}

With the same dd command we get the following output:

value ------------- Distribution ------------- count  
 2 | 0  
 4 | 4  
 8 | 2  
 16 | 0  
 32 | 1  
 64 | 0  
 128 | 0  
 256 | 1  
 512 | 1  
 1024 | 0  
 2048 | 0  
 4096 | 6  
 8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 103598  
 16384 | 0  
  
  
 value ------------- Distribution ------------- count  
 4096 | 0  
 8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 103598  
 16384 | 0

We see that NFS and ZFS align perfectly up to a last digit. This is all great in laboratory but in production you probably have multiple datasets and NFS clients. Let's find out which dataset it was actually written to. To do that we will use first argument to zfs\_write, as comment indicates that's the destination. I already showed how you can ripgrep and use editor to figure out arguments, but now let's use a shortcut. DTrace comes with function print() which can tell you what kind of struct is the first argument:

dtrace -i 'fbt:zfs:zfs\_write:entry { print(args[0]); exit(0); }'  
CPU ID FUNCTION:NAME  
 53 76274 zfs\_write:entry znode\_t \* 0xfffff80be9e0b1d0

Nice, we know it's a pointer, but what's inside? Add \* infront "args" and find out:

dtrace -i 'fbt:zfs:zfs\_write:entry { print(\*args[0]); exit(0); }'  
CPU ID FUNCTION:NAME  
 43 76274 zfs\_write:entry znode\_t {  
 uint64\_t z\_id = 0x290200  
 kmutex\_t z\_lock = {  
 struct lock\_object lock\_object = {  
 const char \*lo\_name = 0xffffffff8267282a  
 u\_int lo\_flags = 0x22710000  
 u\_int lo\_data = 0  
 struct witness \*lo\_witness = 0  
 }  
 volatile uintptr\_t sx\_lock = 0x1  
 }  
 krwlock\_t z\_parent\_lock = {  
 struct lock\_object lock\_object = {  
 const char \*lo\_name = 0xdeadc0dedeadc0de  
 u\_int lo\_flags = 0xdeadc0de  
 u\_int lo\_data = 0xdeadc0de  
 struct witness \*lo\_witness = 0xdeadc0dedeadc0de  
 }  
 volatile uintptr\_t sx\_lock = 0xdeadc0dedeadc0de  
 }  
 krwlock\_t z\_name\_lock = {  
 struct lock\_object lock\_object = {  
 const char \*lo\_name = 0xdeadc0dedeadc0de  
 u\_int lo\_flags = 0xdeadc0de  
 u\_int lo\_data = 0xdeadc0de  
 struct witness \*lo\_witness = 0xdeadc0dedeadc0de  
 }  
 volatile uintptr\_t sx\_lock = 0xdeadc0dedeadc0de  
 }  
 zfs\_dirlock\_t \*z\_dirlocks = 0xdeadc0dedeadc0de  
 zfs\_rangelock\_t z\_rangelock = {  
 avl\_tree\_t rl\_tree = {  
 struct avl\_node \*avl\_root = 0  
 int (\*)() avl\_compar = zfs.ko`zfs\_rangelock\_compare  
 size\_t avl\_offset = 0x8  
 ulong\_t avl\_numnodes = 0  
 }  
 kmutex\_t rl\_lock = {  
 struct lock\_object lock\_object = {  
 const char \*lo\_name = 0xffffffff826824bd  
 u\_int lo\_flags = 0x22710000  
 u\_int lo\_data = 0  
 struct witness \*lo\_witness = 0  
 }  
 volatile uintptr\_t sx\_lock = 0x1  
 }  
 zfs\_rangelock\_cb\_t \*rl\_cb = 0xffffffff823f1810  
 void \*rl\_arg = 0xfffff801637dfae0  
 }  
 boolean\_t z\_unlinked = 0  
 boolean\_t z\_atime\_dirty = 0  
 boolean\_t z\_zn\_prefetch = 0xdeadc0de  
 boolean\_t z\_is\_sa = 0x1  
 boolean\_t z\_is\_ctldir = 0xdeadc0de  
 boolean\_t z\_suspended = 0xdeadc0de  
 uint\_t z\_blksz = 0x10c00  
 uint\_t z\_seq = 0x7a473d  
 uint64\_t z\_mapcnt = 0  
 uint64\_t z\_dnodesize = 0x200  
 uint64\_t z\_size = 0x10ae1  
 uint64\_t z\_pflags = 0x40800000004  
 uint32\_t z\_sync\_cnt = 0  
 uint32\_t z\_sync\_writes\_cnt = 0  
 uint32\_t z\_async\_writes\_cnt = 0  
 mode\_t z\_mode = 0x81a4  
 kmutex\_t z\_acl\_lock = {  
 struct lock\_object lock\_object = {  
 const char \*lo\_name = 0xffffffff8264b421  
 u\_int lo\_flags = 0x22710000  
 u\_int lo\_data = 0  
 struct witness \*lo\_witness = 0  
 }  
 volatile uintptr\_t sx\_lock = 0x1  
 }  
 zfs\_acl\_t \*z\_acl\_cached = 0  
 krwlock\_t z\_xattr\_lock = {  
 struct lock\_object lock\_object = {  
 const char \*lo\_name = 0xffffffff8264b431  
 u\_int lo\_flags = 0x22710000  
 u\_int lo\_data = 0  
 struct witness \*lo\_witness = 0  
 }  
 volatile uintptr\_t sx\_lock = 0x1  
 }  
 nvlist\_t \*z\_xattr\_cached = 0  
 uint64\_t z\_xattr\_parent = 0  
 uint64\_t z\_projid = 0  
 list\_node\_t z\_link\_node = {  
 struct list\_node \*list\_next = 0xfffff802f3600c50  
 struct list\_node \*list\_prev = 0xfffff802af923a80  
 }  
 sa\_handle\_t \*z\_sa\_hdl = 0xfffff8029f5dca68  
 struct zfsvfs \*z\_zfsvfs = 0xfffff80146e92000  
 vnode\_t \*z\_vnode = 0xfffff802c89bac40  
 char \*z\_cached\_symlink = 0  
 uint64\_t z\_uid = 0x3e9  
 uint64\_t z\_gid = 0x3e9  
 uint64\_t z\_gen = 0x3b71a  
 uint64\_t [2] z\_atime = [ 0x64031489, 0x37bba788 ]  
 uint64\_t z\_links = 0x1  
}

Now that's a lot, but with some knowledge you can figure out that vnode\_t is what you need next:

dtrace -i 'fbt:zfs:zfs\_write:entry { print(\*args[0]->z\_vnode); exit(0); }'  
...  
 struct mount \*v\_mount = 0xfffffe025cb56100  
...

dtrace -i 'fbt:zfs:zfs\_write:entry { print(\*args[0]->z\_vnode->v\_mount); exit(0); }'  
...  
 struct statfs mnt\_stat = {  
 ...  
 char [1024] f\_mntonname = [ "/usr/home" ]  
 ...  
 }  
...

What we need in the end is:

fbt:zfs:zfs\_write:entry  
{  
 mount = stringof(args[0]->z\_vnode->v\_mount->mnt\_stat.f\_mntonname);  
 @["zfs", mount] = quantize(args[1]->uio->uio\_resid);  
}

zfs /tmp  
 value ------------- Distribution ------------- count  
 16 | 0  
 32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 3  
 64 | 0  
  
 zfs /  
 value ------------- Distribution ------------- count  
 128 | 0  
 256 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1  
 512 | 0  
  
 zfs /usr/home  
 value ------------- Distribution ------------- count  
 2 | 0  
 4 |@@@@@@@@@@ 1  
 8 | 0  
 16 | 0  
 32 | 0  
 64 |@@@@@@@@@@@@@@@@@@@@ 2  
 128 | 0  
 256 |@@@@@@@@@@ 1  
 512 | 0  
  
 zfs /var/nfs  
 value ------------- Distribution ------------- count  
 8192 | 0  
 16384 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 52822  
 32768 | 0

Following same principle of finding vnode\_t and getting to f\_mntonname, the whole dtrace script is:

fbt:kernel:nfsvno\_write:entry  
{  
 mount = stringof(args[0]->v\_mount->mnt\_stat.f\_mntonname);  
 @["nfs", mount] = quantize(args[2]);  
}

fbt:zfs:zfs\_write:entry  
{  
 mount = stringof(args[0]->z\_vnode->v\_mount->mnt\_stat.f\_mntonname);  
 @["zfs", mount] = quantize(args[1]->uio->uio\_resid);  
}

nfs /var/nfs  
 value ------------- Distribution ------------- count  
 4096 | 0  
 8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 130727  
 16384 | 0  
  
 zfs /var/nfs  
 value ------------- Distribution ------------- count  
 4096 | 0  
 8192 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 130727  
 16384 | 0

If you look at zfs\_read and nfsvno\_read, you'll notice they have the same arguments as their write counterparts, which means just replace "write" with "read" and you have your ZFS and NFS read probes. For iSCSI we need to do a lot more detective work. First, let's prepare ZFS  
dataset.

zfs create -s -V 10G zroot/var/vm1  
dtrace -l | grep zvol | grep write  
70121 fbt zfs zvol\_cdev\_write entry  
70122 fbt zfs zvol\_cdev\_write return  
74042 fbt zfs zvol\_replay\_write entry  
74043 fbt zfs zvol\_replay\_write return  
76290 fbt zfs zvol\_log\_write entry  
76291 fbt zfs zvol\_log\_write return

Let's try to use zvol\_log\_write.

cd /usr/src  
rg '^zvol\_log\_write'  
sys/contrib/openzfs/module/zfs/zvol.c  
530:zvol\_log\_write(zvol\_state\_t \*zv, dmu\_tx\_t \*tx, uint64\_t offset, uint64\_t size, int sync)

So we need args[3] for size and something in zv to identify zvol.

dtrace -i 'fbt:zfs:zvol\_log\_write:entry { print(\*args[0]); exit(0); }'  
CPU ID FUNCTION:NAME  
 62 76290 zvol\_log\_write:entry zvol\_state\_t {  
 char [256] zv\_name = [ "zroot/var/vm1" ]  
...

So the final probe is

fbt:zfs:zvol\_log\_write:entry  
{   
 @[args[0]->zv\_name] = quantize(args[3]);  
}

zroot/var/vm1  
 value ------------- Distribution ------------- count  
 8192 | 0  
 16384 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 59464  
 32768 | 0

The command used to write to zvol is:

dd if=/dev/random of=/dev/zvol/zroot/var/vm1 bs=16k

Let's configure iSCSI first.

cat /etc/ctl.conf  
portal-group pg0 {  
 discovery-auth-group no-authentication  
 listen 0.0.0.0:3260  
}

target [iqn.2012-06.com](http://iqn.2012-06.com/).example:target0 {  
 alias "Example target"  
 auth-group no-authentication  
 portal-group pg0  
 lun 0 {  
 path /dev/zvol/zroot/var/vm1  
 }  
}

service ctld enable  
service ctld start

It is advisable to use another machine as client, because that way your client/server results will not interfere with each other.

service iscsid enable  
service iscsid start  
iscsictl -A -p <ip> -t [iqn.2012-06.com](http://iqn.2012-06.com/).example:target0  
gpart create -s gpt da0  
gpart add -t freebsd-ufs /dev/da0  
newfs -U /dev/da0p1  
mkdir -p /media/da0p1  
mount /dev/da0p1 /media/da0p1

You will get /dev/daN device and you should replace da0 with it. For iSCSI, first error I made is that I assumed that the probe will come from fbt:iscsi module. That was wrong! I needed fbt:cfiscsi. When unsure which probe is the right one, you can do a little test. You can tell DTrace to measure which probe fires the most.

dtrace -i 'fbt:cfiscsi::entry { @[probefunc] = count(); }'  
 cfiscsi\_pdu\_done 3  
 cfiscsi\_pdu\_queue\_cb 3  
 cfiscsi\_callout 8  
 cfiscsi\_datamove\_out 476  
 cfiscsi\_datamove 479  
 cfiscsi\_done 479  
 cfiscsi\_pdu\_queue 954  
 cfiscsi\_handle\_data\_segment 1902  
 cfiscsi\_receive\_callback 1907

cd /usr/src  
rg '^cfiscsi\_receive\_callback'  
sys/cam/ctl/ctl\_frontend\_iscsi.c  
327:cfiscsi\_receive\_callback(struct icl\_pdu \*request)

Using print() like before we get our probe. It is a bit more special than the ones we used before because of how iSCSI works. If there is no data carried in the request, ip\_data\_len will be zero, so we filter out those calls.

fbt:cfiscsi:cfiscsi\_receive\_callback:entry  
/ args[0]->ip\_data\_len > 0 /  
{  
 @ = quantize(args[0]->ip\_data\_len);  
}

value ------------- Distribution ------------- count  
 65536 | 0  
 131072 | 1  
 262144 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1537  
 524288 | 0

Combining the probes gives us

fbt:cfiscsi:cfiscsi\_receive\_callback:entry  
/ args[0]->ip\_data\_len > 0 /  
{  
 @iscsiw = quantize(args[0]->ip\_data\_len);  
}

fbt:zfs:zvol\_log\_write:entry  
{   
 @zvolw[args[0]->zv\_name] = quantize(args[3]);  
}

value ------------- Distribution ------------- count  
 131072 | 0  
 262144 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1965  
 524288 | 0  
  
 zroot/var/vm1  
 value ------------- Distribution ------------- count  
 524288 | 0  
 1048576 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 492  
 2097152 | 0

When we do the math, we can conclude that results are pretty close and that the difference was probably caused by rounding of quantize() function:

262144 \* 1965 = 515112960  
1048576 \* 492 = 515899392  
difference = 786432

Similar combination of counting probes, grepping for "zvol", reading a bit of source code and printing arguments gives the following probe for zvol read

fbt:zfs:zvol\_geom\_bio\_start:entry  
{   
 zv = (zvol\_state\_t \*)args[0]->bio\_to->private;  
 @[zv->zv\_name] = quantize(args[0]->bio\_length);  
}

To make it easier you can "pkg install automount" and "service devd restart" so that it mounts iSCSI target as soon as /dev/daN is available.

Sometimes you just can’t find the proper probe, and that’s the case with iSCSI read. All probes that fire for read, also get triggered on write, so I couldn’t find the way to distinguish read from write.

**And there you have it! I hope this is helpful.**