Sharing data saves space, reduces data skew, and improves data

SSHFS – Installation and Performance

Jeff Layton

File sharing is one of the cornerstones of client/server computing, HPC, and many other architectures. Fundamentally, the ability to share the same data with a number of clients is very appealing because it saves space (capacity), ensures that every client has the latest data, and improves data management. However, this comes at a price, because you now have to administer and manage a central file server, as well as the client tools that allow the data to be accessed.

management. We look at the SSHFS shared filesystem, put it through some performance tests, and show you how to tune it.

A whole class of filesystems tend to be ignored: those based on FUSE (Filesystems in Userspace). These filesystems are run in user space rather than the traditional kernel space. They have many advantages, which I won't elaborate on here; instead, I would rather illustrate their usefulness through an example of a userspace shared filesystem, SSHFS.

SSHFS is a FUSE-based userspace client that mounts and interacts with a remote filesystem as though the filesystem were local (i.e., shared storage). It uses SSH as the underlying protocol and SFTP as the transfer protocol, so it's as secure as SFTP. (I'm not a security expert, so I can't comment on the security of SSH.) SSHFS can be very handy when working with remote filesystems, especially if you only have SSH access to the remote system. Moreover, you don't need to add or run a special client tool on the client nodes or a special server tool on the storage node. You just need SSH active on your system. Almost all firewalls allow port 22 access, so you don't have to configure anything extra, such as NFS or CIFS. You just need one open port on the firewall – port 22. All the other ports can be blocked.

In this article, I'm going to discuss SSHFS in several ways. First, I want to discuss installing and running SSHFS on Linux and show how easy it is to get running. Next, I want to discuss performance and tuning options for SSHFS, and I will compare it to NFS.

Installing SSHFS on Linux

Installing SSHFS is fairly easy. The first step for any FUSE-based filesystem is to make sure that FUSE itself is included in the kernel or as a module. Many distributions already come with FUSE prebuilt. On my CentOS 6.5 desktop, I can check the modules associated with the running kernel (FUSE is usually built as a module):

```
[laytonjb@home4 HPC_028]$ ls -lsa /lib/modules/2.6.32-431.5.1.el6.x86_64/kernel/fs/fuse
total 168
  4 drwxr-xr-x  2 root root   4096 Mar 20 20:09 ./
  4 drwxr-xr-x 30 root root   4096 Mar 20 20:09 ../
 20 -rwxr--r--  1 root root  17160 Feb 11 20:47 cuse.ko*
140 -rwxr--r--  1 root root 141296 Feb 11 20:47 fuse.ko*
```

Another simple way is to check the modules that are loaded into memory:

```
[laytonjb@home4 HPC_028]$ lsmod
Module                  Size  Used by
fuse                   73530  0
...
```

These are just two ways you can check to see whether FUSE has been built for your kernel. Another way is to install the kernel source for your distribution and then look at the kernel configuration, *.config*, in the kernel source directory module.

If you need to build the FUSE module for your system, you should follow the directions on the FUSE homepage, but basically it consists of the common steps one uses to build almost any open source package:

```
$ ./configure
$ make
$ make install
```

These steps build a kernel module for the currently running kernel, but only if you have the source for your latest kernel and the *configure* script can find it. Be sure to do the last step, *make install*, as root).

Once FUSE is installed and working, the next step is to install SSHFS, either by using the package management tools for your distribution or by building from source. If you need to build it from source, the same simple build steps of many open source applications apply,

```
$ ./configure --prefix=/usr
$ make
$ make install
```

where the last step is performed as root. Just be sure to read the SSHFS page for package prerequisites. Notice that I installed SSHFS into */usr*. The SSHFS binary is installed into */usr/bin* which is in the standard path (handy tip). Now, simply try the command *sshfs -V*,

```
$ sshfs -V
SSHFS version 2.5
FUSE library version: 2.8.3
fusermount version: 2.8.3
using FUSE kernel interface version 7.12
```

to make sure everything is installed correctly.

Testing SSHFS with Linux

Now that SSHFS is installed on my desktop, I want to test it and see how it works. In general, I'll use SSHFS to mount a directory from my desktop named *CLUSTERBUFFER2* to my test system. The contents of the directory are:

```
[laytonjb@home4 CLUSTERBUFFER2]$ ls -s
total 3140
   4 BLKTRACE/                      4 NFSIOSTAT/   296 SE_strace.ppt
   4 CHECKSUM/                      4 NPB/           4 STRACE/
  32 Data_storage_Discussion.doc    4 OLD/           4 STRACE_NEW_PY/
   4 FS_AGING/                      4 OTHER2/        4 STRACENG/
   4 FS_Scan/                       4 PAK/           4 STRACE_PY/
 324 fs_scan.tar.gz               164 pak.tar.gz     4 STRACE_PY2/
   4 IOSTAT/                        4 PERCEUS/       4 STRACE_PY_FILE_POINTER/
1300 LWESFO06.pdf                   4 REAL_TIME/   944 Using Tracing Tools.ppt
   4 LYLE_LONG/                     4 REPLAYER/
[laytonjb@home4 CLUSTERBUFFER2]$
```

On my test system, I'm going to use SSHFS to mount this directory as */home/laytonjb/DATA*. This is the mountpoint for the directory (just as NFS filesystems have mountpoints), but first, I need to create the directory */home/laytonjb/DATA* on the test system:

```
[laytonjb@test8 ~]$ mkdir CLUSTERBUFFER2
[laytonjb@test8 ~]$ cd CLUSTERBUFFER2
[laytonjb@test8 CLUSTERBUFFER2]$ ls -s
total 0
```

It’s a good idea to have a mountpoint (directory) that is empty, because when the remote filesystem is mounted, it will “cover up” any files in the directory.

Mounting a filesystem using SSHFS is easy. Remember that because it's a userspace filesystem, any user can create these with no admin intervention. The basic form of the SSHFS command is as follows:

```
$ sshfs user@home:[dir] [local dir]
```

For my example, the command I used is:

```
[laytonjb@test8 ~]$ sshfs 192.168.1.4:/home/laytonjb/CLUSTERBUFFER2 \
/home/laytonjb/CLUSTERBUFFER2
laytonjb@192.168.1.4's password:
```

Notice that I used the full paths for both the remote directory and the local directory. Also notice that I had to type in my password on the desktop (the storage server).

Now I should take a look at the directory on the test system to make sure it actually worked:

```
[laytonjb@test8 ~]$ cd DATA
[laytonjb@test8 DATA]$ ls -s
total 3140
   4 BLKTRACE                      4 NFSIOSTAT    296 SE_strace.ppt
   4 CHECKSUM                      4 NPB            4 STRACE
  32 Data_storage_Discussion.doc   4 OLD            4 STRACE_NEW_PY
   4 FS_AGING                      4 OTHER2         4 STRACENG
   4 FS_Scan                       4 PAK            4 STRACE_PY
 324 fs_scan.tar.gz              164 pak.tar.gz     4 STRACE_PY2
   4 IOSTAT                        4 PERCEUS        4 STRACE_PY_FILE_POINTER
1300 LWESFO06.pdf                  4 REAL_TIME    944 Using Tracing Tools.ppt
   4 LYLE_LONG                     4 REPLAYER
```

SSHFS has a multitude of options. I recommend reading the man page because it's not too long. One main point you want to pay attention to is the mapping of UIDs and GIDs (user IDs and group IDs) between systems. If the permissions on the client system don't match the server system, you might have to do some mapping between the two systems. Fortunately, SSHFS has the ability to read a file that contains the mapping information and take care of mapping between the systems.

SSHFS Performance on Linux

One of the things I'm curious about is the performance of SSHFS. Although it's not exactly a replacement for NFS, it can be viewed as a shared filesystem protocol. That is, a number of clients can mount a filesystem from a central storage server just like NFS. Given that the protocol is SFTP, I was curious about performance, particularly from a single client. Does encryption/decryption adversely affect performance? To help answer this question and others, I decided to do some performance testing.

On my desktop, I have a Samsung 840 SSD that is attached via a SATA 2 connection (6Gbps) and mounted as */data*. It is formatted ext4 using the defaults (I'm running CentOS 6.5). I will use this as the testing filesystem. The desktop system has the following basic characteristics:

- Intel Core i7-4770L processor (four cores, eight with HyperThreading, running at 3.5GHz)
- 32GB of memory (DDR3-1600)
- GigE NIC
- Simple GigE switch
- CentOS 6.5 (updates current as of 3/29/2014)

The test system that mounts the desktop storage has the following characteristics:

- CentOS 6.5 (updates current as of 3/29/2014)
- GigaByte MAA78GM-US2H motherboard
- AMD Phenom II X4 920 CPU (four cores)
- 8GB of memory (DDR2-800)
- GigE NIC

I'll be using the following three tests from IOzone for testing: (1) sequential writes and re-writes, (2) sequential reads and re-reads, and (3) random input/output operations per second (IOPS; 4KB record sizes). Both NFS and SSHFS will be tested using all three tests.

Testing starts with baseline results using the defaults that come with the Linux distributions. Then, I'll test some configuration that have been tuned in two ways: (1) SSHFS mount options and (2) TCP tuning on both the server and the client. The second type of tuning affects the NFS results as well.

Baseline Testing

For these tests, I used the defaults in CentOS 6.5 for both SSHFS and TCP. In the case of NFS, I exported */data/laytonjb* from my desktop (192.168.1.4), which takes on the role of a server, to my test machine (192.168.1.250), which takes on the role of the client. I then mounted the filesystem as */mnt/data* on the test system. In the case of SSHFS, I mounted */data/laytonjb* as */home/laytonjb/DATA*.

The first test I ran was a sequential write test using IOzone. The command line I used was the same for both NFS and SSHFS:

```
./iozone -i 0 -r 64k -s 16G -w -f iozone.tmp > iozone_16G_w.out
```

The command only runs the sequential write and re-write tests (*-i 0*) using 64KB record sizes (*-r 64k*) with a 16GB file (*-s 16G*). I also kept the file for the read tests (*-w*).

The sequential read tests are very similar to the write tests. The IOzone command line is:

```
./iozone -i 1 -r 64k -s 16G -f iozone.tmp > iozone_16G_r.out
```

The command only runs the sequential read and re-read tests (*-i 1*) using the same record size and file size as the sequential write tests.

The random IOPS testing was also done using IOzone. The command line is pretty simple as well:

```
./iozone -i 2 -r 4k -s 8G -w -O -f iozone.tmp > iozone_8G_random_iops.out
```

The option *-i 2* runs random read and write tests. I chose to use a record size of 4KB (typical for IOPS testing) and an 8GB file (a 16GB file took too long to complete). I also output the results in operations/second (ops/s; *-O*) to the output file.

Normally, if I were running real benchmarks, I would run the tests several times and report the average and standard deviation. However, I'm not running the tests for benchmarking purposes; rather, I want to get an idea of the comparative performance between SSHFS and NFS. However, I did run the tests a few times to get a feel for the spread in the results and make sure the variation wasn't too large. Table 1 contains the baseline results, which used the defaults for CentOS 6.5, SSHFS, and NFS.

Table 1: Baseline IOzone results for NFS and SSHFS

Notice that for sequential writes and re-writes, and sequential reads and re-reads, SSHFS performance is about half the NFS performance. The random write IOPS performance is very similar for SSHFS and NFS, but the random read IOPS performance for SSHFS is about half the NFS performance.

Protocol	Test	Results
Sequential (MBps)		
NFS Baseline	Write	83.996
SSHFS Baseline	Write	43.182
NFS Baseline	Re-write	87.391
SSHFS Baseline	Re-write	49.459
NFS Baseline	Read	114.301
SSHFS Baseline	Read	54.757
NFS Baseline	Re-read	114.314
SSHFS Baseline	Re-read	63.667
Random (ops/s)		
NFS Baseline	Write IOPS	13,554
SSHFS Baseline	Write IOPS	12,574
NFS Baseline	Read IOPS	5,095
SSHFS Baseline	Read IOPS	2,859

SSHFS Optimizations (OPT1)

The next set of results uses some SSHFS mount options to improve performance; in particular:

- *-o cache=yes*
- *-o kernel_cache*
- *-o compression=no*
- *-o large_read*
- *-o Ciphers=arcfour*

The first option turns on caching and the second option allows the kernel to cache as well. The third option turns off compression, and the fourth option allows large reads (which could help read performance).

The fifth option switches the encryption algorithm to Arcfour which is about the fastest encryption algorithm, with performance very close to no encryption (cipher). It doesn't provide the best encryption, but it is fast, and I'm looking for the fastest possible performance [Note: I believe the default is the aes-128 cipher].

Table 2 adds the SSHFS optimization results to Table 1. The optimizations are listed as "OPT1" in the table.

Table 2: Baseline and OPT1 IOzone results for NFS and SSHFS

Protocol	Test	Results
Sequential (MBps)		
NFS Baseline	Write	83.996
SSHFS Baseline	Write	43.182
SSHFS OPT1	Write	81.646
NFS Baseline	Re-write	87.391

SSHFS Baseline	Re-write	49.459
SSHFS OPT1	Re-write	85.747
NFS Baseline	Read	114.301
SSHFS Baseline	Read	54.757
SSHFS OPT1	Read	112.987
NFS Baseline	Re-read	114.314
SSHFS Baseline	Re-read	63.667
SSHFS OPT1	Re-read	131.187
Random (ops/s)		
NFS Baseline	Write IOPS	13,554
SSHFS Baseline	Write IOPS	12,574
SSHFS OPT1	Write IOPS	21,533
NFS Baseline	Read IOPS	5,095
SSHFS Baseline	Read IOPS	2,859
SSHFS OPT1	Read IOPS	3,033

Notice that the performance of SSHFS OPT1 virtually matches the NFS performance for sequential write, re-write, and read performance. Also notice, however, that the sequential re-read performance for SSHFS is better than the NFS performance. Actually, it's better than wired speed, so some caching is obviously affecting the results.

Also notice that the random write IOPS performance for SSHFS is better than the NFS performance by quite a margin (almost 2x). Again, perhaps this is a sign that some sort of caching is going on. However, notice that the random read IOPS performance, even with the SSHFS tuning options, is about the same as for the SSHFS Baseline performance. Moreover, both are not as good as the NFS Baseline results.

SSHFS and TCP Optimizations (OPT2)

Now that some SSHFS options have been tried, I'll look at some TCP tuning options to improve performance. These tests build on the previous set of tests, so I will have SSHFS *and* TCP tuning options.

The TCP optimizations used primarily affect the TCP *wmem* (write buffer size) and *rmem* (read buffer size) values along with some maximum and default values. These tuning options were taken from an article on OpenSSH tuning. The parameters were put into */etc/sysctl.conf*. Additionally, the MTU for both the client and server were increased (the client could only be set to a maximum of *6128*). The full list of changes is below.

- MTU increase
  - 9000 on "server" (Intel node)
  - 6128 on "client" (AMD node)
- *sysctl.conf*values:
  - *net.ipv4.tcp_rmem = 4096 87380 8388608*
  - *net.ipv4.tcp_wmem = 4096 87380 8388608*
  - *net.core.rmem.max = 8388608*
  - *net.core.wmem.max = 8388608*

- *net.core.netdev.max_backlog = 5000*
- *net.ipv4.tcp_window_scaling = 1*

I cannot vouch for each and every network setting, but adjusting the *wmem* and *rmem* values is a very common method for improving NFS performance.

Table 3 lists the results for NFS and SSHFS. The SSHFS results, which include both SSHFS and TCP tuning are labeled “OPT2.” The TCP tuning options also affect NFS performance, so these tests are re-run and listed as “NFS TCP Optimizations.”

Table 3: Baseline, OPT1, and OPT2 IOzone results for NFS and SSHFS

Protocol	Test	Results
Sequential (MBps)		
NFS Baseline	Write	83.996
NFS TCP Optimizations	Write	96.334
SSHFS Baseline	Write	43.182
SSHFS OPT1	Write	81.646
SSHFS OPT2	Write	111.864
NFS Baseline	Re-write	87.391
NFS TCP Optimizations	Re-write	92.349
SSHFS Baseline	Re-write	49.459
SSHFS OPT1	Re-write	85.747
SSHFS OPT2	Re-write	119.196
NFS Baseline	Read	114.301
NFS TCP Optimizations	Read	120.010
SSHFS Baseline	Read	54.757
SSHFS OPT1	Read	112.987
SSHFS OPT2	Read	119.153
NFS Baseline	Re-read	114.314
NFS TCP Optimizations	Re-read	120.189
SSHFS Baseline	Re-read	63.667
SSHFS OPT1	Re-read	131.187
SSHFS OPT2	Re-read	136.081
Random (ops/s)		
NFS Baseline	Write IOPS	13,554
NFS TCP Optimizations	Write IOPS	13,345
SSHFS Baseline	Write IOPS	12,574

SSHFS OPT1	Write IOPS	21,533
SSHFS OPT2	Write IOPS	29,802
NFS Baseline	Read IOPS	5,095
NFS TCP Optimizations	Read IOPS	5,278
SSHFS Baseline	Read IOPS	2,859
SSHFS OPT1	Read IOPS	3,033
SSHFS OPT2	Read IOPS	3,237

The results are very interesting. First, notice that the TCP optimizations improve NFS sequential write, re-write, read, and re-read performance a bit, but the random write and read IOPS performance is about the same.

The SSFHS performance is also very interesting (SSHFS OPT2). The sequential write and re-write performance increases by a fair amount (by about one third). The performance is much higher than NFS or SSHFS OPT1. On the other hand, the sequential read and re-read performance of SSHFS OPT2 is higher than the OPT1 values and only just a bit higher than the NFS results with the TCP tuning.

The random write IOPS performance for SSHFS OPT2 is much larger than NFS or even the SSHFS OPT1 performance. It is more than two times the NFS Baseline values, but the random read IOPS performance for SSHFS OPT2 is still not as good as the NFS performance, only achieving about 60% of the TCP optimized NFS performance.

SSHFS and TCP Optimizations – Increased Encryption (OPT3)

Up to this point, I changed the encryption to a cipher that had a minimal effect on performance. With this change, the results in Table 3 indicate that some aspects of SSHFS performance can be much better than NFS. However, what happens if I use the default encryption (aes-128)? Does performance take a nose dive, or does it stay fairly competitive with NFS?

I re-ran the tests for SSHFS using the options for SSHFS OPT2, but removing the option *-o Ciphers=arcfour* from the SSHFS *mount* command. The performance results are labeled “SSHFS OPT3” in Table 4.

Table 4: Baseline, OPT1, OPT2, and OPT3 IOzone results for NFS and SSHFS

Protocol	Test	Results
Sequential (MBps)		
NFS Baseline	Write	83.996
NFS TCP Optimizations	Write	96.334
SSHFS Baseline	Write	43.182
SSHFS OPT1	Write	81.646
SSHFS OPT2	Write	111.864
SSHFS OPT3	Write	51.325
NFS Baseline	Re-write	87.391
NFS TCP Optimizations	Re-write	92.349
SSHFS Baseline	Re-write	49.459
SSHFS OPT1	Re-write	85.747

SSHFS OPT2	Re-write	119.196
SSHFS OPT3	Re-write	56.328
NFS Baseline	Read	114.301
NFS TCP Optimizations	Read	120.010
SSHFS Baseline	Read	54.757
SSHFS OPT1	Read	112.987
SSHFS OPT2	Read	119.153
SSHFS OPT3	Read	58.231
NFS Baseline	Re-read	114.314
NFS TCP Optimizations	Re-read	120.189
SSHFS Baseline	Re-read	63.667
SSHFS OPT1	Re-read	131.187
SSHFS OPT2	Re-read	136.081
SSHFS OPT3	Re-read	67.588
Random (ops/s)		
NFS Baseline	Write IOPS	13,554
NFS TCP Optimizations	Write IOPS	13,345
SSHFS Baseline	Write IOPS	12,574
SSHFS OPT1	Write IOPS	21,533
SSHFS OPT2	Write IOPS	29,802
SSHFS OPT3	Write IOPS	14,100
NFS Baseline	Read IOPS	5,095
NFS TCP Optimizations	Read IOPS	5,278
SSHFS Baseline	Read IOPS	2,859
SSHFS OPT1	Read IOPS	3,033
SSHFS OPT2	Read IOPS	3,237
SSHFS OPT3	Read IOPS	2,860

In examining the results in Table 4, it is easy to see the effect on performance of switching to the default encryption. The sequential write, re-write, read, and re-read performance is just about cut in half relative to the SSHFS OPT3 performance. Performance is still better than the SSHFS Baseline results, but not by a great deal. The random IOPS performance, both write and read, was also just about cut in half, bringing performance very close to the SSFS Baseline results. These results really reinforce how greatly encryption algorithms can affect the performance of SSHFS.

SSHFS and TCP Optimizations – Compression (OPT4)

Depending on your situation, SSHFS might need stronger encryption than Arcfour, but you would still like to improve performance. Does SSHFS have options that can help? For these tests, I turned compression “on” hoping that this

might improve performance – or at least illustrate the effect of compression on performance.

The results when compression is turned on are labeled “OPT4” in Table 5. I've left the other results in the table for comparison, despite the fact that the table is getting rather long.

Table 5: Baseline, OPT1, OPT2, OPT3, and OPT4 IOzone results for NFS and SSHFS

Protocol	Test	Results
Sequential (MBps)		
NFS Baseline	Write	83.996
NFS TCP Optimizations	Write	96.334
SSHFS Baseline	Write	43.182
SSHFS OPT1	Write	81.646
SSHFS OPT2	Write	111.864
SSHFS OPT3	Write	51.325
SSHFS OPT4	Write	78.600
NFS Baseline	Re-write	87.391
NFS TCP Optimizations	Re-write	92.349
SSHFS Baseline	Re-write	49.459
SSHFS OPT1	Re-write	85.747
SSHFS OPT2	Re-write	119.196
SSHFS OPT3	Re-write	56.328
SSHFS OPT4	Re-write	79.589
NFS Baseline	Read	114.301
NFS TCP Optimizations	Read	120.010
SSHFS Baseline	Read	54.757
SSHFS OPT1	Read	112.987
SSHFS OPT2	Read	119.153
SSHFS OPT3	Read	58.231
SSHFS OPT4	Read	158.020
NFS Baseline	Re-read	114.314
NFS TCP Optimizations	Re-read	120.189
SSHFS Baseline	Re-read	63.667
SSHFS OPT1	Re-read	131.187
SSHFS OPT2	Re-read	136.081
SSHFS OPT3	Re-read	67.588
SSHFS OPT4	Re-read	158.925

Random (ops/s)		
NFS Baseline	Write IOPS	13,554
NFS TCP Optimizations	Write IOPS	13,345
SSHFS Baseline	Write IOPS	12,574
SSHFS OPT1	Write IOPS	21,533
SSHFS OPT2	Write IOPS	29,802
SSHFS OPT3	Write IOPS	14,100
SSHFS OPT4	Write IOPS	19,326
NFS Baseline	Read IOPS	5,095
NFS TCP Optimizations	Read IOPS	5,278
SSHFS Baseline	Read IOPS	2,859
SSHFS OPT1	Read IOPS	3,033
SSHFS OPT2	Read IOPS	3,237
SSHFS OPT3	Read IOPS	2,860
SSHFS OPT4	Read IOPS	2,892

For better comparison, the data in Table 5 is plotted in Figures 1-6. Each plot is one of the test results.

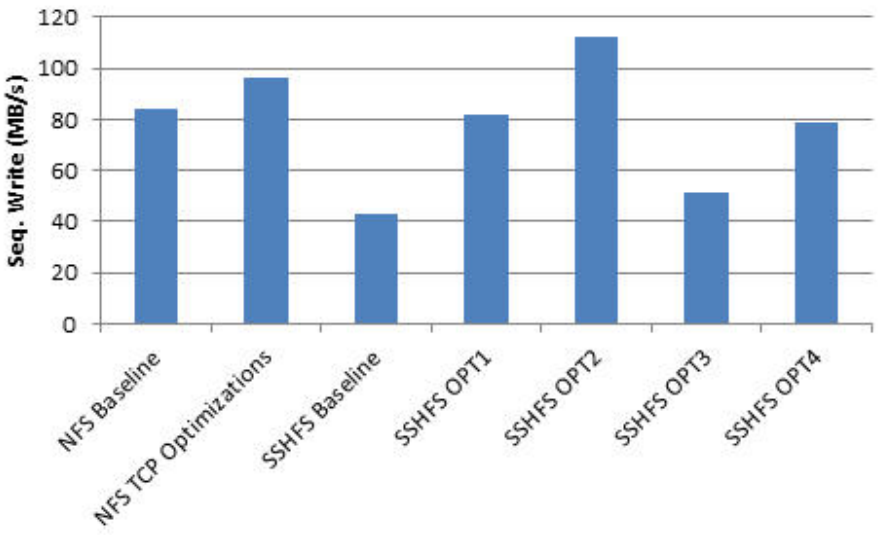

Figure 1: IOzone results for Sequential Write test.

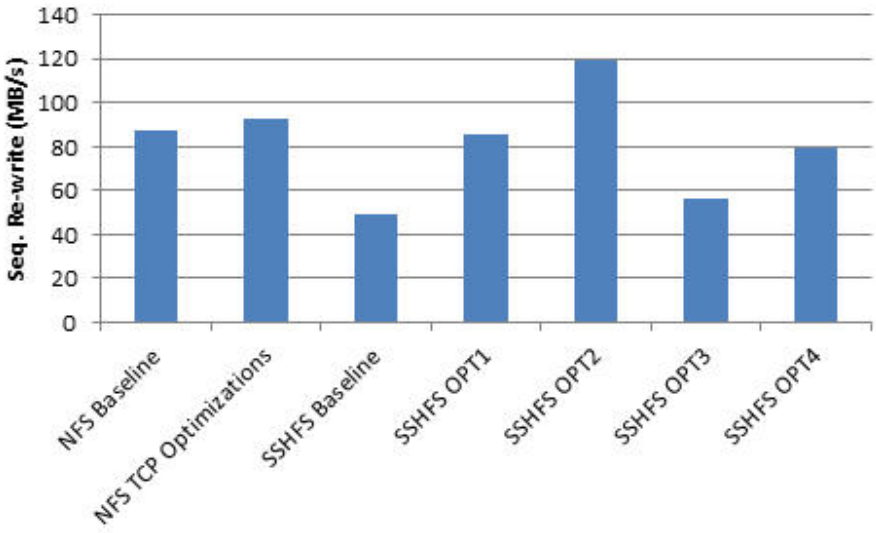

Figure 2: IOzone results Sequential Re-write test.

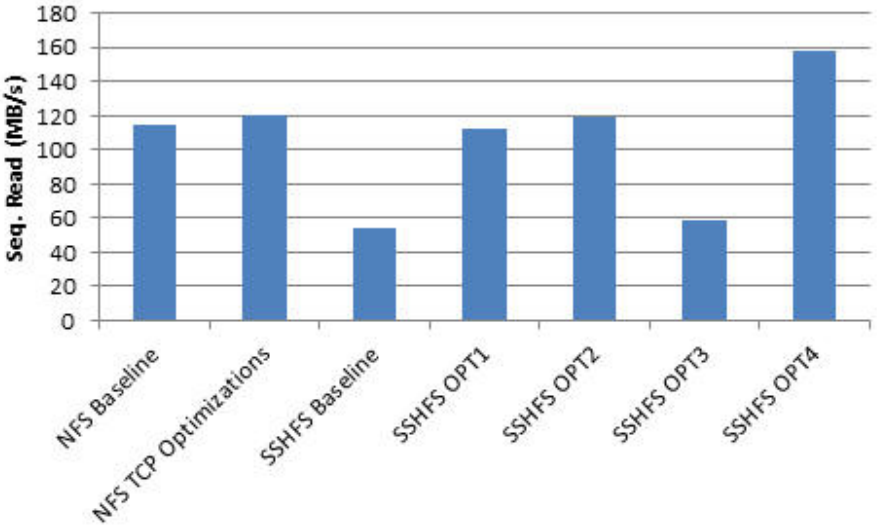

Figure 3: IOzone results Sequential Read test.

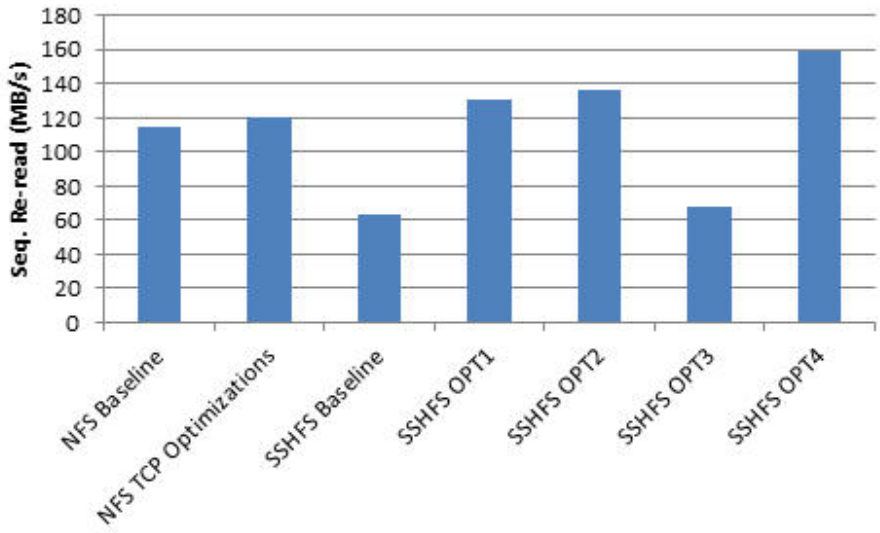

Figure 4: IOzone results for Sequential Re-read test.

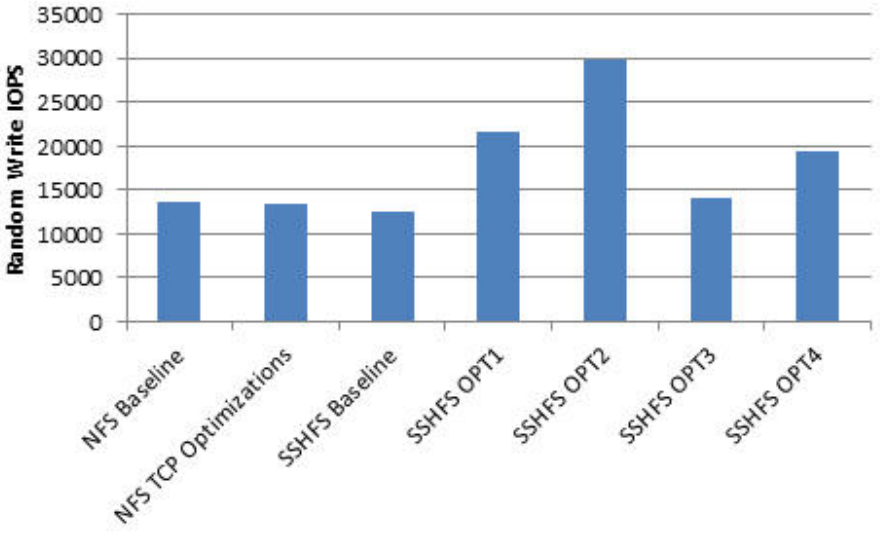

Figure 5: IOzone results for Random Write IOPS test.

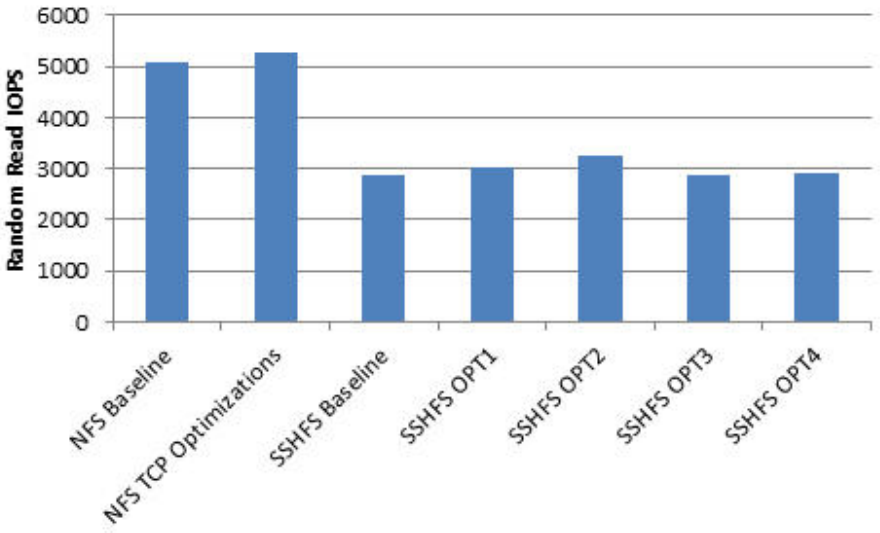

Figure 6: IOzone results for Random Read IOPS test.

A quick glance at the results illustrates that, for these tests, turning compression on (OPT4) seemed to have greatly improved performance. The sequential write and re-write performance has increased by about 50% relative to the SSHFS OPT3 results and the SSHFS Baseline results. The sequential write and re-write results are not as good as using weak encryption (SSHFS OPT2), but turning on compression recovers some of the lost performance when using better encryption (aes-128). It's also interesting to note that the SSHFS OPT4 results are close to the NFS Baseline results.

The sequential read and re-read performance improvement resulting from turning on compression is remarkable. The results increase by roughly a factor of three times compared with the SSHFS OPT3 and SSHFS Baseline results. Read performance is well above NFS performance as well. The results are also faster than wire speed, so I am definitely seeing the effect of compression, caching, or both on read performance.

The effect of compression on Random Write IOPS performance is also remarkable. Turning on compression increased the Random Write IOPS performance by about 50% relative to SSHFS OPT3 and SSHFS Baseline

performance. However, compression has very little effect on Read IOPS. The SSHFS OPT4 results are about the same as SSHFS OPT3 and SSHFS Baseline results.

CPU Comparison of SSHFS and NFS

During the tests, I noticed that CPU activity on the SSHFS client and server was higher compared with NFS by watching GKrellM. I did a few screenshots that seem to represent what I was seeing on both the client and the server, although these are only point-in-time screenshots, whereas I should really be examining the systems during the entire run and do time comparisons (perhaps in my copious free time). Regardless, Figure 7 compares the GKrellM screenshot on the server for the three tests (sequential write and re-write, sequential read and re-read, and random IOPS) for the NFS run with optimized TCP settings and SSHFS with the OPT4 settings (increased encryption and compression tuned on).

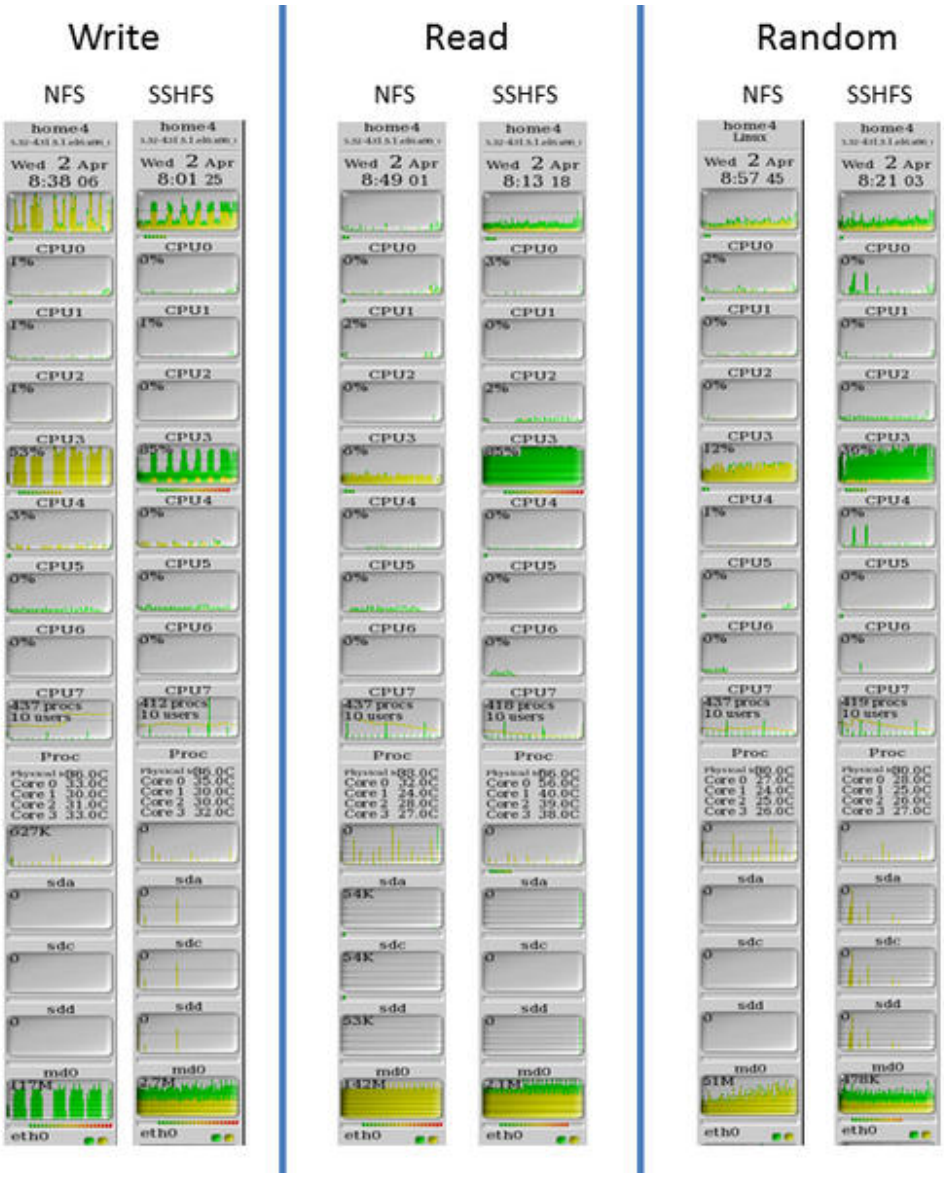

Figure 7: GKrellM for the server during the sequential write, sequential read, and random IOPS testing.

It might be a little difficult to compare NFS and SSHFS in Figure 7, but in general, you can see that the SSHFS approach used more CPU resources than NFS on the server. For writes, the server CPU load on CPU4 peaked at about 53% for NFS, and was 85% for the SSHFS runs. For the reads, NFS had a very low CPU usage (~6%), whereas SSFHS hit about 85%. [Note: I tried to take the screenshots at about the same point in each run.]

Figure 8 shows the same comparison, but for the client.

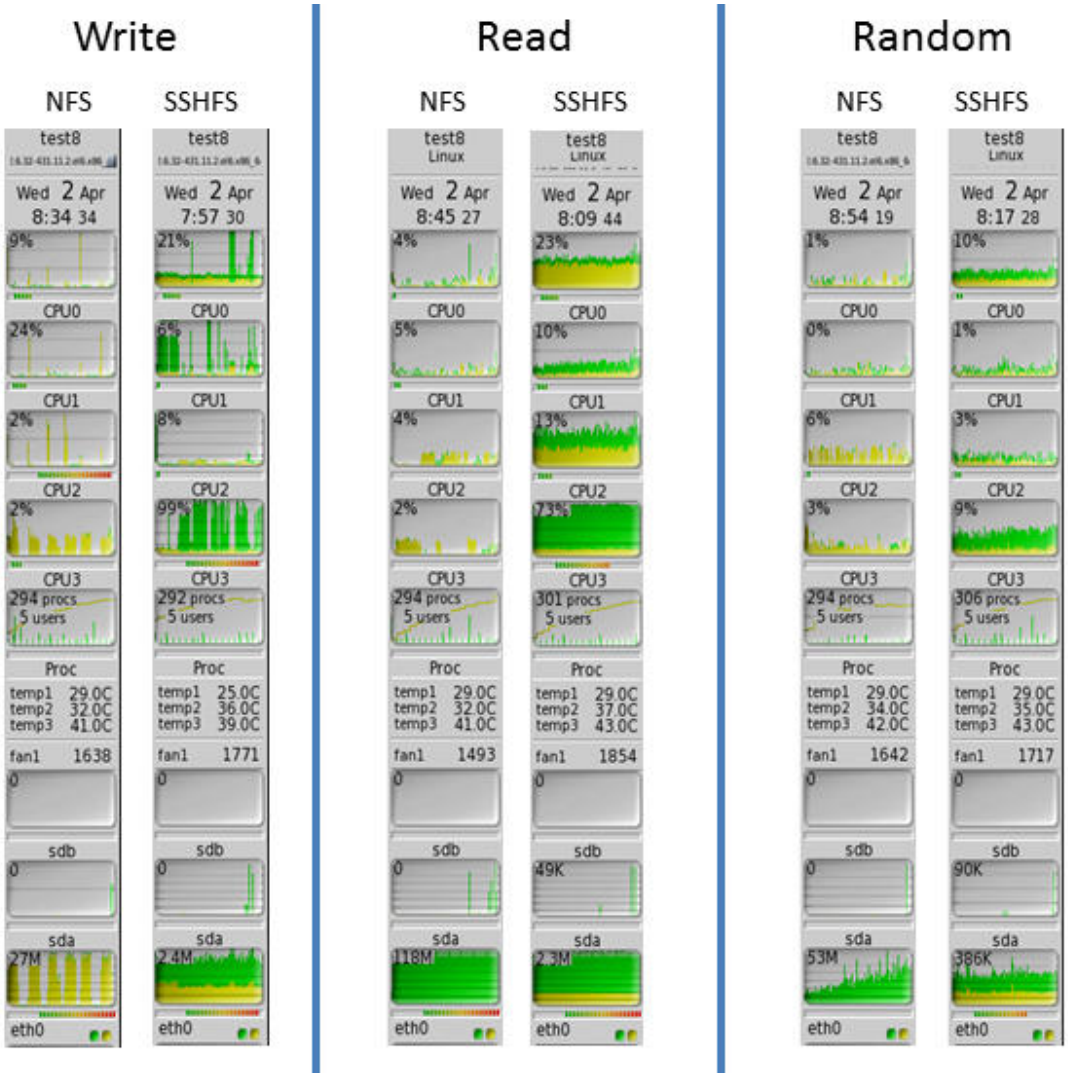

Figure 8: GKrellM for the client during the sequential write, read, and random IOPS testing.

You can see that SSHFS used more CPU resources than NFS, particularly for the read test.

Summary

Sharing data between clients has many advantages. The most obvious is having just a single copy of the data, thereby reducing capacity, data skew, and data management and, overall, making life much easier. Although several shared storage options are available,other options are often overlooked. In this article, I looked at a userspace option named SSHFS, whichis anSSH-basedfilesystem, and I discussed performance testing and tuning.

SSHFS is really simple to build, install, and use. Because it's in user space, any user can mount a remote filesystem using only SSH (port 22). Therefore you only need to have one open port on the client, port 22. Moreover, because users can take advantage of it whenever needed, it does not involve admin intervention. It works quite well in my simple tests, but I wasn't pushing it very hard.

If you want to use SSHFS in place of other shared filesystems, performance is usually one of the first topics of conversation. I ran some very simple single-client tests against a server connected to the client via GigE. I used IOzone to run sequential write and re-write tests, sequential read and re-read tests, and random write and read IOPS testing. These tests were also run on NFS for a comparison to SSHFS. To make things a little more interesting, I tried some SSHFS options and some TCP options with an eye toward improving performance.

The results were quite interesting. With a little tuning, SSHFS performance was pretty close to NFS performance for sequential write and re-write, even when default encryption (aes-128) was used. When compression was turned on, sequential read and re-read performance was actually better than NFS and better than wire speed. Random write

IOPS performance was always very good, either coming close to NFS performance or achieving 50% better performance. However, random read IOPS performance was never better than 60% of NFS performance.

Because SSHFS uses SSH, the server has to encrypt the data before sending, and the client has to decrypt the data once it is received. This imposes a CPU load on the client and server. A quick comparison of CPU load on the server and client for all three workloads for both NFS and SSHFS revealed that the load on a single core was much greater for SSHFS than for NFS. The increased workload on the client during sequential read tests was a little higher than I expected.

SSHFS is an underestimated filesystem that could be useful in a great many scenarios. According to the limited tests I ran, the performance can be quite good – good enough that I would seriously consider using it in HPC.

Don't forget that SSH is a protocol and not dependent on the OS, so you have SSHFS clients for other operating systems, such as Windows. Maybe that would make an interesting follow-up article!