Shared File System Experiences

Denise Underwood-Hannagan, DOD and *Patrick McQueeney*, CRI

ABSTRACT: *As our site's disk space requirements grow, the ability to make data available across systems, avoiding duplication, becomes an ever more important consideration. Therefore, the concept of Shared File Systems (SFS) has become very attractive to our site. This paper describes our experiences with CRI's implementation of SFS as it has matured.*

Introduction

As a site with multiple Crays, disk storage has a number of inherent challenges. The typical users on most machines want all of the space, so disk space is always in demand. Users with accounts on multiple machines understandably want to have access to their data from each machine. However, duplicating data across machines is costly in terms of buying disks, maintaining changes to the data across the machines, and backing up the same data multiple times.

With these considerations in mind, CRI's implementation of Shared File Systems (SFS) became very attractive to our site. SFS is a method of multiple machines sharing UNIX file systems using semaphores through hardware connections to common disk storage. SFS is an alternative to Network File Systems (NFS) with the advantage of all machines being equal with respect to the disks.

History

We first became involved with SFS as a test site for the non-SSD implementation. We were using a network HiPPI disk, ND14, as the shared media and a software version of a semaphore device. This semaphore device was a program that ran on a Sun workstation and handled all the semaphore activities. We were running at Unicos 7.C.3 plus numerous mods for the SFS implementation. This initial effort was strictly a proof of concept for our site.

Our next dealings with SFS came at Unicos levels 8.0.2 and 8.0.3. At Unicos 8.0.2 we reached the functionality we saw in our initial testing but were now using the new HiPPI-based semaphore device, H-SMP. The H-SMP is a Sparc-based device produced by Performance Technologies, Inc. which includes an S-bus HiPPI card. The H-SMP has the code necessary for semaphore functionality provided in a PROM. This H-SMP allowed us more flexibility as we were able to put it directly on the existing HiPPI network.

At Unicos 8.0.3, we saw improvements in performance and stability, but still had challenges with SFS that prevented it from being used in a production environment. With Unicos 8.0.4 plus a set of stabilization mods, we reached a level where we felt that the product would be beneficial to our user groups. At this point our SFS filesystem was released to a small user community who has been quite pleased with its flexibility and performance.

Configuration

Our implementation of CRI's SFS consists of two Model-E machines, one H-SMP, an ND14, and an existing HiPPI network. The first machine, the semaphore box, and the ND14 are connected to the same NSC PS32 HiPPI switch. The second system uses an additional PS32 and fiber optics in order to get to the common HiPPI switch. (see Figure 1).

The ND14 was configured with 2 facilities, each made up of a small RAID 1 partition and a large RAID 5 partition. We decided to use one facility for the filesystem and one for the shared lock region (SLR) and the shared mount table.

The filesystem itself was made up of a small Raid 1 primary partition and a large Raid 5 secondary partition. Both partitions were allocated with a sector size of 32K. The goal of this filesystem configuration was to use the primary partition strictly for directory and filesystem meta-data allocations while the actual data took advantage of the large sector size of the secondary partition.

Immediate Findings

We set out in our initial testing to see where SFS may help us. It seemed logical to try things that a typical user might try. Most users do a great deal of changing directories, looking at directories, copying files, and searching for strings in files. We ran a set of tests consisting of *ls*, *find*, *grep,* and *cpio* commands in both shared and non-shared modes. It was quickly ascertained that SFS was not the medium to perform this typical filesystem I/O. This type of filesystem I/O is the worst case for both the ND14 and SFS so the combination was exponentially poor. The tests ran in 102 seconds in non-shared mode and in 4000

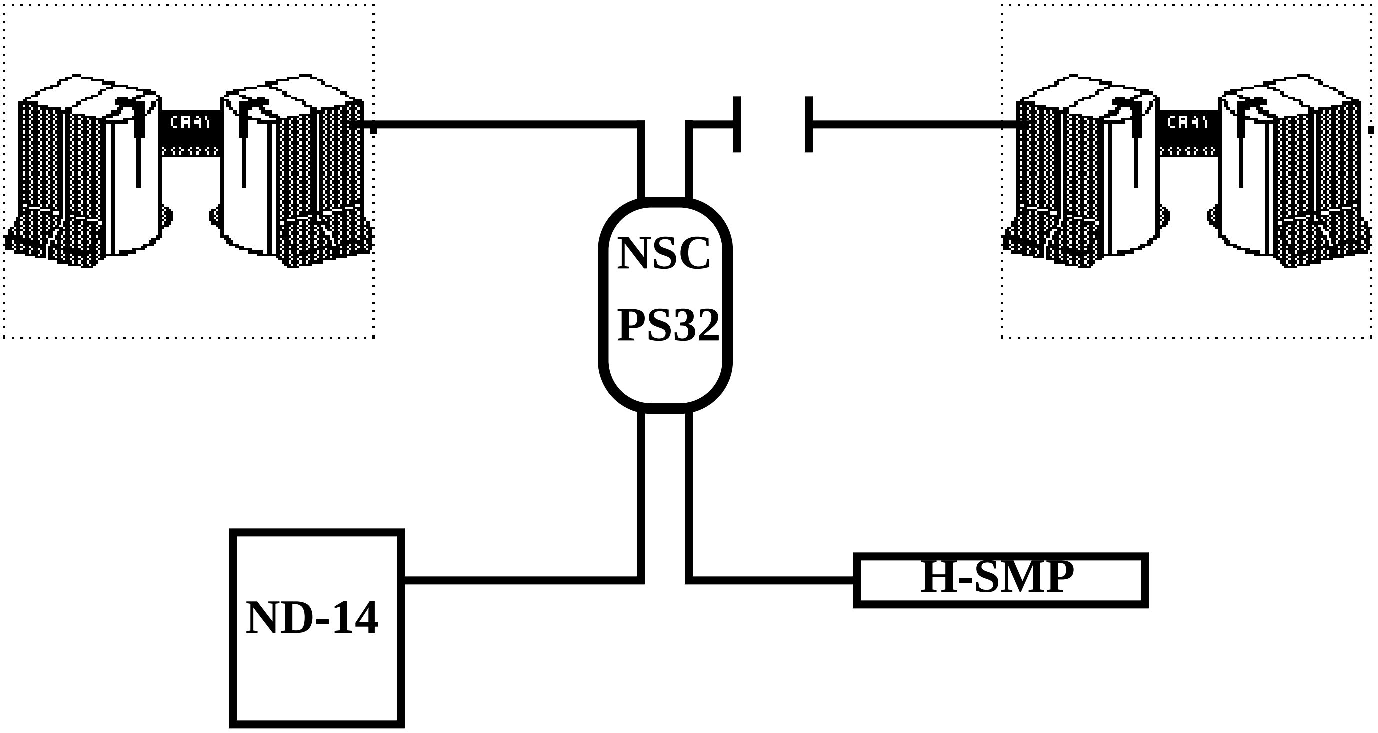

Figure 1.

seconds in shared mode. It was obvious to us that the benefits of SFS are not great enough to outweigh such a substantial performance penalty. With this information in mind, we decided to narrow the scope of our testing. We changed our focus to test large block I/O which might be able to use more bandwidth of the HiPPI channel.

Tests and Results

As we first looked into large block I/O, the initial thrust was to see what kind of performance we could get and what size buffer we would need to get that performance. We also wanted to compare our SFS I/O rates to those of the non-SFS case. We then looked to see if small changes in the program could reap significant benefits in performance.

The program itself was a basic I/O program which would double the buffer size on each iteration starting at 4K until it reached 128Mbytes. It was also flexible enough to allow us to pick and choose flags to be used on the file open and execute different I/O strategies. This gave us the ability to try to find the combination which yielded the greatest performance.

We began by changing from buffered I/O to raw I/O. This allowed us to bypass both library and systems buffers which resulted in a dramatic improvement in performance. Notation (A) in Figure 2 illustrates this benefit.

We also looked into the various locking mechanisms available to us. Read locks, write locks, and the exclusive open flag would permit the SFS code to make certain assumptions about the status of the file and any current users of the file. It also would allow the host to reduce the inode overhead associated with each read or write system call. This becomes important in the SFS environment as very little buffering of filesystem meta-data and inodes is done.

In our testing, the use of the different types of locks showed a noticeable increase in performance. We saw an increase of as much as 19 Mbyte/sec. Notation (B) in Figure 2 shows the overall concept of using the locking mechanisms. We noticed an interesting phenomena while reading a data file without a read lock and no write permission. In this case, the SFS code adds the read lock on the user’s behalf. This allowed the performance to be improved with no changes to the actual application.

As the block size was increased and the file locking mechanism was used, it did appear that the performance would be extremely close to the device speed seen in a non-SFS environment. (See Figure 3). Once the buffer size reached 512 Kbytes, the transfer rate reached about 9.5 Mbytes/sec. This really was the minimum rate that would be acceptable to our user community. The performance continued to increase until leveling off at around 47 Mbytes/sec with a buffer size of 64 Mbytes. The actual graph of both the SFS case and the non-SFS case shows very little differences throughout all the changes in buffer size.

Now satisfied that the I/O rates for large block SFS transfer were acceptable, we tried to come up with a testing scenario which would reflect how our user community might actually use the SFS filesystem. Standard procedure for our users is to have one application writing a file and multiple processes reading a file. It seemed to us that a producer/consumer relationship fit the

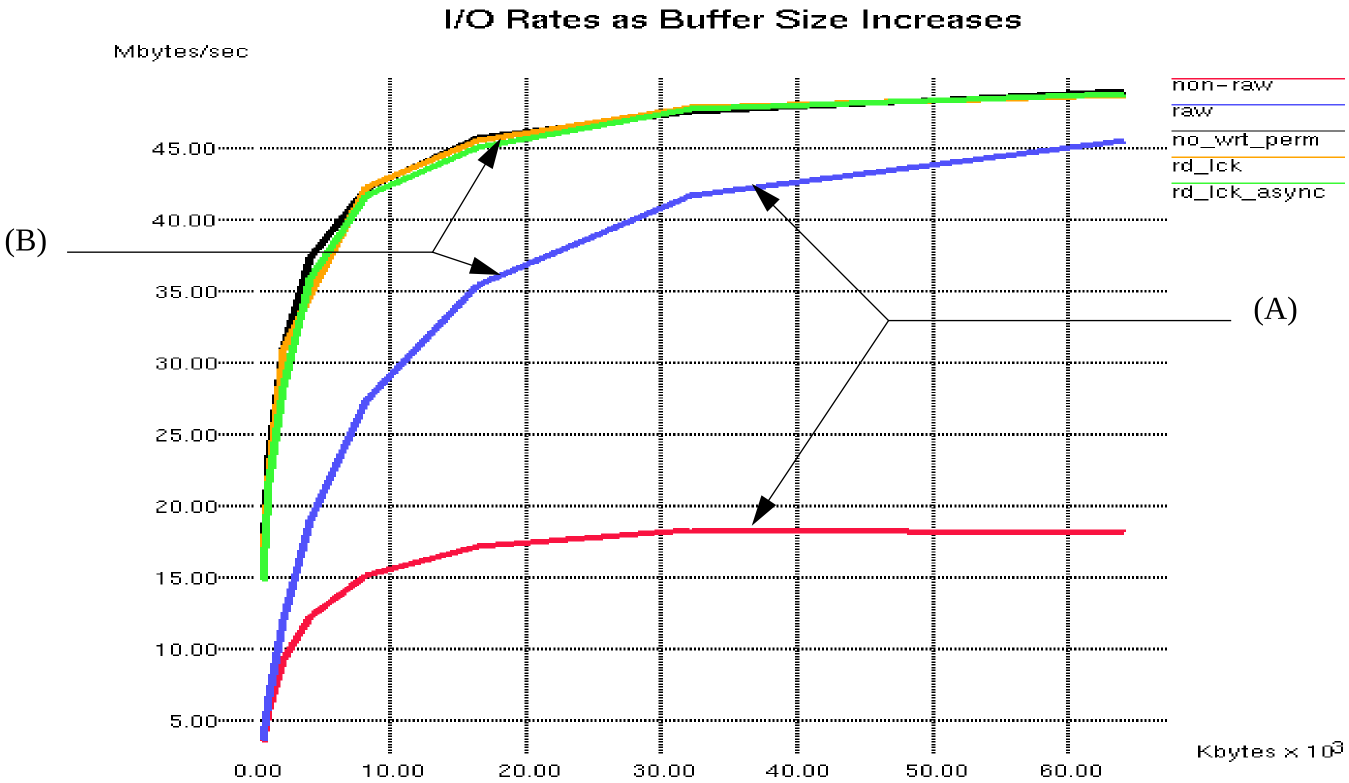

Figure 2:

bill. One process would produce, or write, the file and multiple consumers would consume, or read, the file. This would allow true sharing of the data file itself.

All of the producer/consumer tests we ran used a buffer size of 32Mbytes and a file size of 640Mbytes. Each producer or consumer would use the appropriate lock mechanism and also use raw asynchronous I/O.

We began with the simplest case. We had one producer and one consumer that ran on the same host. The test yielded a producer rate of 33.69 Mbytes/sec. and a consumer rate of 47.6 Mbytes/sec. The consumer rate reached the device speed we had observed in previous testing.

Next, we expanded the scenario to include two additional consumers from the same host. Each of the consumers reached a performance rate of about 17 Mbytes/sec. This seemed to illustrate that the total bandwidth of the path to the device was being shared. We then changed the scenario slightly to have two consumers from two separate hosts. Each of the consumers ran at about 24 Mbytes/sec. When we increased the number of producers to two on both hosts, we saw a combined rate of 36 Mbytes/sec for each host. In the cases tested, it appeared that the bandwidth was being shared equally.

As we continued to increase the number of consumers from each host, we did see points where there was significant competition for both the HiPPI channel and other SFS resources. One consumer "gets control" of the HiPPI channel, gets its request out for the inode semaphore then starts its I/O. The other consumers are then competing for both the channel and the inode semaphore before starting their I/O. We modified the test program slightly to allow all the consumers to get the read lock and then wait until the other consumers can also get the read lock. We believe that the competition for the HiPPI channel was now only for I/O instead of semaphore activity. The actual performance we saw was about 6.5 Mbytes/sec. which is not as high as we might have hoped but we have to live with the constraints of system resources.

These results forced us to re-evaluate the scope of our testing. We acknowledged that our testing may not be an actual representation of the "real life" usage of the SFS. It does seem unlikely that once a file has been produced that multiple consumers will try to use the same file at exactly the same instant. Hence, the competition for the same file should be less significant.

Opportunities

SFS at Unicos 8.0.4 has proven to be a useful product. In order to get to this point, we have seen our share of problems. We currently have one outstanding problem which we have been able to work around. This problem can occur when a host starting SFS fails to detect the live members of the SFS cluster. It then makes the assumption that it is the only member of the cluster and clears most of the SFS infrastructure. This clear can result in a panic of the actual live systems in the cluster. The

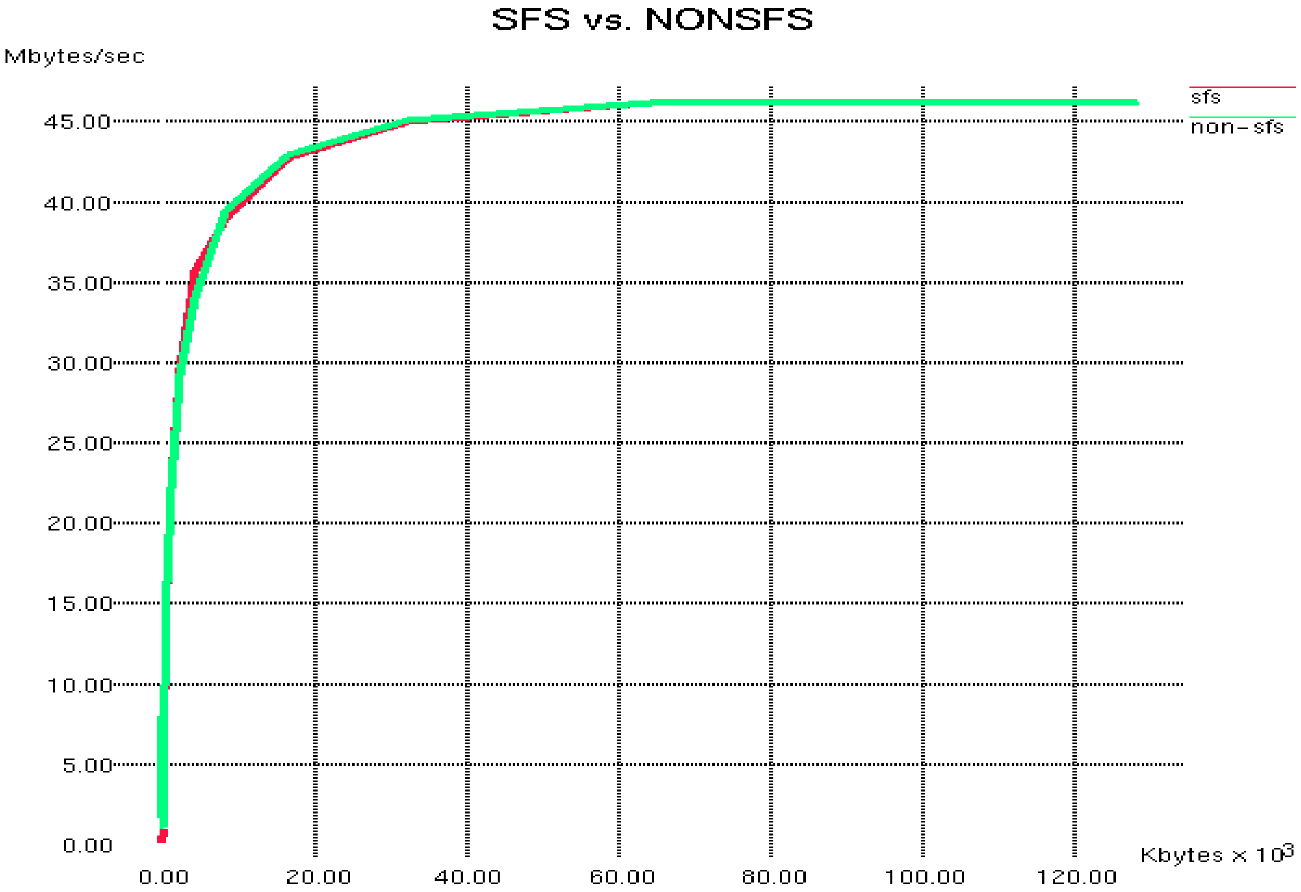

Figure 3:

temporary fix is to prevent the SFS daemon, *sfsd*, from being started with the clear flag.

Another problem we encountered was actually an intended feature of the system. When the hosts could not reach the H-SMP, they would panic. This reaction was not acceptable at our site—we prefer not to have machines crashing just because the network is down. This, and other problems, have been resolved through a set of mods to Unicos 8.0.4

Conclusion

Overall, the system users have been pleased with the performance and flexibility that SFS does provide. We realize it has limits and cannot be used to solve all of the challenges in a shared data computing environment. Yet, for applications that read and write large data blocks and require access to common data sets, SFS can provide the capability to reach the data set and allow a more efficient and economical use of disk space.