Chip:
A Temporary, Memory Based Network Filesystem

Benjamin Chambers, Jeff Shneidman, Carl Steinbach
bac@mit.edu, jeffsh@eecs.harvard.edu, cws@mit.edu

December 6, 2001

Abstract

In this paper we present Chip, a network filesystem which uses the memory of participating server machines as the filesystem's store. File data is either striped or replicated across participating servers on a per-file basis. Since data is stored in the memory of server machines this filesystem accomplishes reduced latency by completely avoiding slow reads and writes to disk. Furthermore, individual server failures do not affect the files stored in the system since lost data can be reconstructed using stripe parity information or from redundant data replicas. Our preliminary evaluations of Chip indicated that while the system is faster than disk for certain operations, further work is necessary before it can compete in earnest.

1 Introduction

Many modern computing applications make use of disk based file systems for temporary storage of intermediate results. For example, compilers store object files which they later link to generate program binaries; these files are stored on disk and tend to make compilation an I/O bound task, at least in the linking stage. Similarly, many multimedia applications (image manipulation, audio/video editing, etc.) make use of swap space on disk to store intermediate changes made to a file before the user has saved the changes and committed them to permanent storage. (These intermediate results also allow changes made to the files to be undone before they have been saved).

We present Chip, a system which improves the speed of these applications (and any others requiring a fast, private, short-term file system) by allowing computers on a LAN to organize themselves into a distributed, fast, reliable file system which any other computer on the network running our software can use. Chip distributes data across the memories of multiple computers on a LAN, and ensures continued operation despite server failure by replicating or striping data across the Chip server pool on a per-file basis. In addition, our system dynamically reconfigures the number of computers in the server pool and adjusts the striping or replication configuration accordingly.

We observe as many others have that improvements in the speed of hard disks have not kept pace with those of CPUs, memory, or networking hardware. We also note that while the price of memory continues to fall (resulting in larger memories in computers), there is no shortage of applications whose need for short-term storage exceeds the amount of physical RAM that a typical computer is likely to have. For these applications, the use of Chip will result in considerable speedups over using a local hard drive for swap space.

Chip allows any computer on a LAN to *publish* some amount of its RAM for use by other computers in the system. Any computer on the LAN which requires fast, short-term storage can organize a filesystem for itself and make use of the memory published by the other machines on the LAN. The Chip software allows nodes to join and leave the server pool without compromising the integrity of the data stored in the filesystem. If nodes leave the system in an orderly manner, i.e. are shut down properly, users of the system will not notice any change. If a node leaves the system abruptly, users who have data stored on the missing node may experience a brief drop in performance while the filesystem regenerates the lost data.

Chip provides a private filesystem for each workstation using the system. This means that each workstation's Chip filesystem is only available to that workstation. Taking this approach optimizes for the applications we are interested in (namely, fast short-term storage for applications requiring a great deal of swap space) and avoids the complications inherent in maintaining consistency in a shared filesystem.

2 Related Work

Our system might be classified as a limited RAID [11] setup [3], using distributed memory instead of local disks. To our knowledge, we are the first to consider this modification. Zebra [6] is a network file system that increases throughput by striping file data across multiple servers. However, Zebra doesn't consider the case of rapidly adding and removing servers, since it is based on disk RAID striping setups. There has been much work on data replication to make data highly available in network file systems such as Byzantine NFS [2] and Harp. Harp [7] is a replicated Unix file system that provides highly available and reliable storage for files and guarantees that file operations are executed atomically in spite of failures. We borrow the observation that replication means that with some assumptions, we don't need to write data to disk. Harp is a general purpose file system, whereas Chip's imposed limitations (namely, each user sees their own data) allow us to make some concurrency shortcuts. Coda [8] and Locus [9] use replication to improve both availability and response time. However, all three are geared toward achieving data consistency across multiple users, which is a problem we do not encounter in Chip. In addition, rather than consider how our replication scheme performs in the face of less than n failures, we consider the problems involved when an unbounded number of servers fail, but each fails with specific time properties with respect to the others. There has been considerable work in shared virtual memory systems [5]. Finally, the notion of the "virtual disk" allows an abstraction of the implementation. Network file systems such as NFS [1] and Coda [8] back up the notion of a file with mechanisms that provide some level of access and consistency guarantees across multiple disks in a distributed setting. Local memory filesystems such as md [4] support a filesystem in memory. (There are other ways to implement a virtual disk, such as single-file virtual disks like those created by FreeBSD's `vnconfig` or `mdconfig`.) Chip combines ideas from both network and memory representations, but our access assurances are closer to the network case, and our speed assurances are closer to the memory case.

3 Environment

Chip is intended to run in an environment consisting of a distributed collection of nodes that communicate by means of a network. For Chip to achieve effective performance we require that the roundtrip network packet transit time between a client and any of the remote servers be a fraction of the native filesystem I/O access time on the client machine. Based on contemporary disk and network interface card performance specifications, a network using 100 megabit or gigabit NICs easily satisfies this requirement. Nodes can crash because of power failures, hardware failures, or software errors. Node machines in Chip can function as clients, remote servers, or both.

4 Design

Chip's design can be logically broken up into two different components, a *local server* and some number of *remote servers*. We refer to them hereafter as LS and RS respectively. The LS resides on the workstation that is using the filesystem, and it receives NFS requests from the client on the workstation. The LS then issues requests to the RSs to serve the client requests. Copies of the RS program run on every workstation in the system that wishes to publish memory for the system to use. The RSs are responsible for storing data and returning them on the LS's request. Figure 1 shows a diagram of a workstation running the LS and three other machines running the RS. Note that the workstation itself could also be publishing some if its memory with its own copy of the RS program, and likewise each of the other machines in the system could be running the LS program and using memory from other machines.

4.1 Replication Vs. Striping

When storing data, the LS either replicates or stripes data across the remote server pool in order to ensure filesystem reliability. The user decides whether the LS will replicate or stripe data when he starts the LS program. When operating in replication mode, the LS stores each block of data on two remote servers. Hence, one of the two remote servers can crash and the LS will still be able to read its data. When operating in striping mode, the LS arranges blocks into stripes consisting of up to n blocks each, and computes a parity block by taking the exclusive-OR of the n blocks in the stripe. The n blocks and the parity block are then stored on $n + 1$ distinct remote servers. Consequently, one of the $n + 1$ remote servers can disappear from the network without the LS losing its data. Although striping utilizes space on remote servers more efficiently, replication provides superior I/O performance. Consequently, a user working with small files should probably run the LS in replication mode, whereas a user working with a large dataset should run the LS in striping mode, assuming he is willing to sacrifice some performance.

4.2 Remote Servers

The operation of the RSs is simpler than that of the LS, so we discuss it first. The job of each RS is to store blocks of data sent to it by the LSs running on other nodes. An RS will receive an initial request from an LS indicating that the LS wishes to store some blocks on the RS, and requesting a "lease" from that RS for a given period of time, typically an hour. The RS can notify the LS at some point if it wishes to revoke the lease and leave the system, but otherwise the RS will keep any data sent to it by the LS for at least the amount of time stipulated by the lease. All blocks sent to the RS from a given LS are kept under the same lease, and the LS may renew the lease with the RS at any time. LSs renew their leases with the RSs at regular intervals that are significantly less that the length of the lease itself, such as every 10 minutes. This ensures that, should an LS become separated from an RS, its data will remain on the RS for nearly a full lease period before that RS will reuse any of that memory.

Once a lease has been established on the RS for a given LS, the RS receives requests (create, read, write, delete) for blocks. On create requests, the RS stores the block of data and returns a unique identifier which the LS can use to refer to that block in any future read, write or delete requests. The unique identifier, which we call a *claim slip*, consists of the RS's local time appended to a counter. This ensures that each block is effectively timestamped with its creation time; if an RS reboots without the LS knowing and starts handing out new claim slips, there is zero probability that the RS will issue a claim slip value identical to any it has previously issued. This ensures that an LS will never accidentally retrieve a block other than the one it originally stored.

4.3 Local Server

The LS is responsible for managing the Chip filesystem of the local machine on which it is running. It maintains all of the metadata for that workstation's Chip filesystem and sends block requests to the RSs as necessary to retrieve data for the files in the filesystem. The metadata that the LS must maintain is similar to the metadata in a standard disk-based file system, with a few notable differences. Whereas typical disk based filesystems store the addresses of the pages on disk that contain the filesystem's data, Chip stores the addresses of blocks located on remote servers. Each block address consists of the name and port number of the RS that the block resides on, that block's claim slip value, and a pointer to the stripe to which the block belongs.

Each stripe in the system lists the block addresses of all the data blocks in that stripe, plus the block address of that stripe's parity block. If the system is configured to do replication instead of striping, there is only one entry in each stripe's block list and we refer to it as the *primary block*. Instead of storing a parity block in the stripe, we

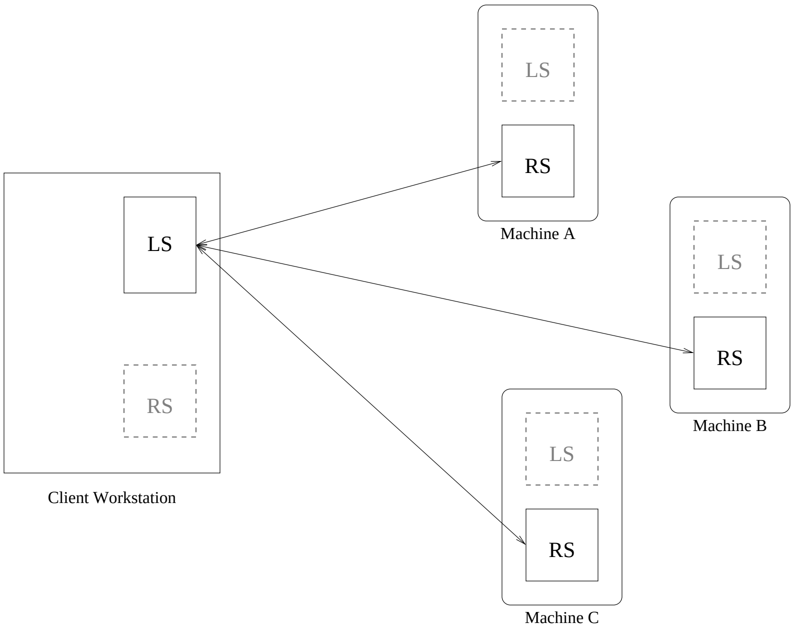

Figure 1: Chip Overview: Local Server and Remote Servers

store a *replication block* whose content is identical to that of the primary block. We maintain a list of these stripes, each of which is given a unique identifier. As explained above, each block contains a pointer to the stripe to which it belongs. In the event that a block becomes unavailable, the LS can look up the stripe, retrieve all the other blocks in the stripe along with that stripe's parity block, and regenerate the missing block. In the case of replication, it is even simpler; the LS merely looks up, retrieves, and returns the replication block, which is equivalent to the unavailable primary block.

Each LS maintains a list of active RSs and randomly selects one from the list when it creates a new block, which serves to load-balance the system across the available RSs. Obviously, in order to maintain availability, all the blocks in a given stripe must reside on RSs on different machines.

4.4 System Reconfiguration and Failures

System reconfigurations occur when an RS leaves the server pool. The process is orchestrated by the local servers, and each LS is responsible for ensuring that all of the files used by its clients make it safely through a system reconfiguration. There are two different cases in which system reconfiguration is necessary. The first is if a user publishing some of their memory (i.e., running a copy of RS) decides to stop publishing it. In that case, the data that their RS is storing must be relocated. The other case is if an RS goes down unexpectedly. In this case, the lost data must first be reconstructed, and then relocated as described above. The final case we consider is what happens if the LS itself crashes or loses power.

4.4.1 Unpublishing Memory

For both replicated and striped systems, handling an RS unpublishing its memory is fairly straightforward. In this situation, the RS notifies all of the LSs that it is storing data for (that is, those LSs for which it has a current lease). Upon being notified, each LS removes that RS from its list of available RSs. Every LS must then read back all of the blocks in its filesystem that are stored on that RS, assign them to a new RS selected from the list, and update the metadata for each block accordingly.

4.4.2 Failure of a Remote Server

If an RS fails unexpectedly, the LSs must handle things slightly differently since they don't have the opportunity to read back the data from the RS that left the server pool. We'll consider the replicated and striped systems separately, discussing the replicated case first.

Replicated System The LS will need to respond when it attempts to either read or write a block residing on an RS which is down. Either the request to the RS will time out, or the RS will already have been removed from the available RS list.

In the case of a read, an LS running in replication mode will need to look up the replicated block corresponding to the unavailable primary block and return that to the client instead. It then needs to select a new RS from the list of available RSs, create a new block on that RS with the correct data, and update the metadata of the file so that it now points to the new RS to find the primary block for that file.

For a write, it is slightly more complicated. If the write spans the full width of the block in question, then the new state of the data in the block doesn't depend on the previous state, and the LS can simply create a new block on an available RS and update the metadata to point to that new block, as well as writing the new data back to the replica block. If the size of the write is less than the block size, then the LS must first read the replica block and modify the data according to the write, and then write back the data both to the replica block and to a newly created block on a new RS.

Striped System The striped version of Chip is similar to the replicated case, but slightly more involved. Recall that each stripe consists of a collection of n blocks from one file, plus 1 parity block, for a total of $n+1$ blocks per stripe, where n is the stripe width. Each block in a stripe is stored on a different RS.

In the event of a read failure on a block, all n other blocks from the stripe must be retrieved and the missing block's data reconstructed from the stripe's other data block and its parity block. The reconstructed result is then returned to the client. In addition, the LS must select another available RS and store the block back to that RS so that future requests can be answered without reconstructing the block each time. If there are no more available RSs in the system, then the LS must create another stripe, divide the blocks between them, and update the parities accordingly. To avoid this, it's best to have the number of RSs in the system be significantly larger than n.

Chip's response to a write failure in striped mode is similar to its handling of the same situation in replicated mode. Depending on whether the write spans the entire block, the LS may or may not need to reconstruct the missing block before performing the write. In either case, after creating the new block on an available RS, the parity block must also be updated. As with a read failure, the LS may need to split up the stripe if there aren't enough available RSs remaining.

4.4.3 Reconstruction

The previous section explains how the LS can continue to service requests in the event of an RS failure. However, simply relocating blocks as needed in a lazy fashion is not enough to ensure the reliability of the system, since another RS could fail in the future. For that reason, once an LS observes one RS failure, it should wait a little while to ensure that the failure is not a temporary network outage, while continuing to service requests and move blocks as explained above. Once the LS has waited long enough that it is unlikely that the RS is going to reappear, the LS must do what it did in the case of unpublishing by an RS. Namely, it must operate in the background and relocate every block that used to reside on the failed RS to a new RS. In the meantime, the LS can continue to service filesystem requests as explained above. This ensures that Chip will not lose user data if RSs fail independently and with some delay between their failures,

4.4.4 Failure of a Local Server

Failures of the LS are catastrophic for the filesystem, since the LS maintains all of the metadata for the filesystem in volatile memory. However, we believe that this is an acceptable tradeoff for the purposes of providing a temporary filesystem; it is likely that if the user's workstation crashes, they will need to restart the computation that was in progress. Most processes do not require files in `/tmp` to survive a reboot of the local machine. In the event that an LS fails, its leases will eventually expire on the RSs that it was utilizing, which will subsequently release the used blocks and free the memory they were using.

5 Implementation Details

Our implementation of Chip is based on the SFS toolkit [10]. In particular, we implemented Chip as an NFS loopback server with the SFS toolkit, building on the sfsusrv program. This means that Chip provides a standard NFS interface to the workstation using it, and to a user of that workstation Chip looks like any other part of the workstation's filesystem.

Figure 2 shows the layout of the unmodified NFS loopback server. In this system, filesystem accesses to files in the `/sfs` directory are passed to the kernel NFS client and then passed up to the user-level SFS client, which communicates via NFS 3 calls to the sfsusrv program on the remote machine (i.e., the machine exporting the filesystem). The sfsusrv program then services requests from its local disk based filesystem. See [10] for details.

Figure 3 shows the layout of Chip's implementation. Like the standard NFS loopback server setup, accesses to files in the `/sfs` directory are passed to the user-level SFS client, which communicates with the sfsusrv program. However, in Chip, the sfsusrv resides on the client workstation. Instead of servicing requests from the disk-based filesystem, sfsusrv makes function calls to the LS module. The LS module manages the metadata in the filesystem and makes RPC calls to the RSs, which store the actual file blocks in RAM.

One side note is that mounting an SFS filesystem on the same machine that is exporting it is usually not allowed by the SFS client, because in the normal case (when the sfsusrv is exporting some part of that machine's disk-based FS) there is risk of filesystem deadlock. Chip does not have this problem since all file data resides in memory and never exists on the local machine at all. However, in order to run Chip, it is necessary to kill sfscd (the SFS client) and restart it with the `-l` option to enable the mounting of an sfs path that is being served by the local machine.

Two different versions of Chip were implemented: a prototype with no replication, and a version which implemented replication. The prototype stores each file block only once and assumes no RSs will fail; it was written as a proof of concept and used to benchmark the speed of our system. The replicated version performs replication as described in our design.

A version of Chip that stripes data instead of replicating it was not implemented due to time constraints. This remains an option for future work. However, it's worth noting that a striped system would almost certainly be slower than a replicated version since on every write it is necessary to retrieve all the blocks in the relevant stripe, recalculate the stripe's new parity, and write the new block and the parity block both back to their respective RSs. By comparison, the replicated version needs only to issue 2 writes, one each for the primary and replication blocks.

Leases and dynamic reconfiguration were also not included in the replicated version that we implemented; currently data will stay on an RS indefinitely (or until the program is shut down). The first time an LS makes a read request to an RS that has failed, there will be a long delay while RPCs time out; once the timeout occurs, the LS will retrieve the replicated block. The LS notes that the RS is down and routes any subsequent requests to the replica automatically, instead of sending them to the failed RS. However, re-replicating the data on a different RS has not yet been implemented, nor has recovery for writes to failed RSs.

The Chip prototype was implemented in roughly 4000 lines of C++ code, including comments. The LS consists of approximately 3400 lines of code, and the RS required around 600 lines of code. The replicated version added approximately 200 additional lines of code.

The LS module implements the metadata management of the filesystem from scratch. It maintains its own tables of tnodes (the Chip equivalent of inodes) that contain file attributes and addresses of blocks of data. The data block addresses specifically consist of the name and port of the RS stor-

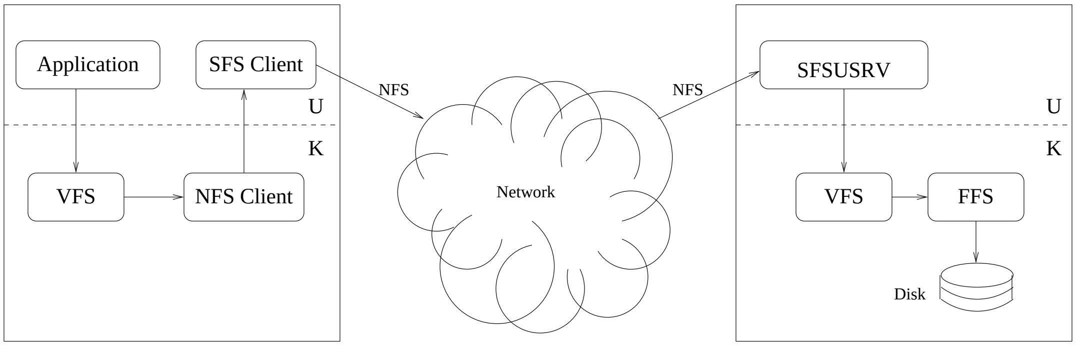

Figure 2: Standard NFS Loopback Server

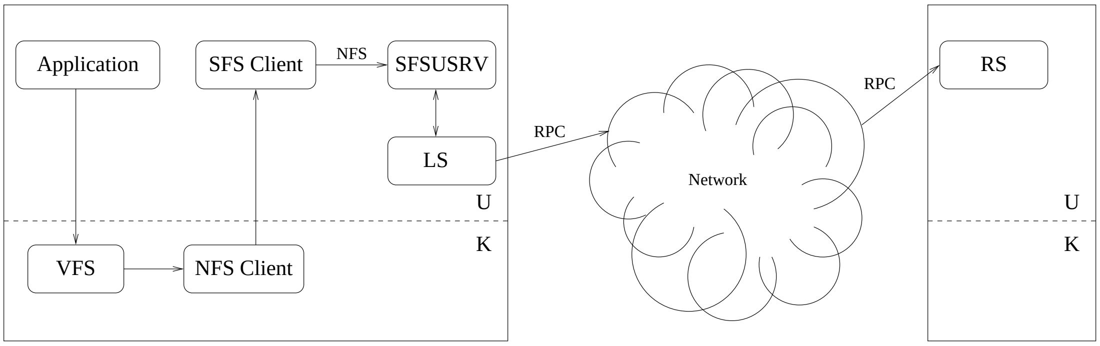

Figure 3: Chip Architecture

ing the block, along with a unique claim slip which was described earlier. Tnodes for directories also contain a list of directory entries, and tnodes for symlinks contain the target string of the symlink. Directory entries consist of a filename, pointer to the file's tnode, and a unique identifier that the LS module passes to sfsusrv to identify that file.

The LS module operates asynchronously, receiving requests from sfsusrv (which were in turn generated as the result of NFS requests from the SFS client) and issuing them to an RPC handler that packages an appropriate RPC call to the correct RS and calls a callback when the RPC call returns. The LS assembles the necessary replies from the RSs and calls the sfsusrv when the request is complete.

The RS was implemented as a user-level application that allocates memory to share to the LSs. Multiple LSs communicate with multiple RSs via a custom RPC interface. Using this RPC interface, clients can create, delete, re-write, read, and renew interest in memory blocks. Like the LS, our implementation of the RS uses the SFS asynchronous library. The RS can optionally pin pages in main memory (not used in our implementation as we noticed that the server never swaps to disk in our tests). In addition, it can sweep expired memory regions which might be created if clients crash before officially deleting memory. (We employ a timeout mechanism to detect this case.) Finally, the RS provides hooks to notify LS clients when the RS is about to terminate. Although we don't use this feature in our LS implementation, this would allow an LS to re-replicate data before an RS goes down.

6 Performance Analysis

In this section of the paper we consider the functionality, usability, and performance of the Chip system.

6.1 Functionality and Usability

A user can access a Chip filesystem using standard UNIX filesystem calls. Users can create and delete directories, define rights, make symbolic links, and monitor the size of their files. All of the methods that come up in normal file system usage are implemented - we used our own system to put together this paper.

6.2 Performance

There are many issues that factor into a disk performance study. In looking at local disk access speeds, we must consider the seek times and bandwidth of the disk, the disk caching policies, the load on the machine at the time of the test, and possibly other factors. In evaluating Chip, we must consider network bandwidth and ping times, network saturation, machine load, the cost of marshalling data into RPC packets of various sizes, the cost of splitting data into TCP packets, the internal block size memory representation, and of course, our own implementation performance (including our decision to implement RS as a user-level client program and the local server within the framework of sfsusrv). With logical reasoning and a few assumptions based on our understanding of various system components, we can reason about Chip's performance and evaluate the strengths and weaknesses of our prototype.

The current implementation of Chip beats sfsusrv running as a locally exported NFS directory (and trounces remote NFS file exports) but is bested by operations on the local disk. While our system meets or beats local disk usage for certain operations (delete, create, and operations that manipulate metadata in memory without requiring network traffic), it is clear that local disk is faster for more realistic workloads in the testbed environment that we have seen so far. We believe that modifications to our implementation and network hardware upgrades might reveal more compelling speedups for normal workloads. In this section, we'll discuss the results that we have compiled so far, and consider what modifications seem justified by our data.

Figure 4 shows two different versions of Chip competing against local disk and locally mounted sfsusrv. The test involved 100 runs of a program which created a file, wrote a specified amount of randomly generated data to the file, flushed the data to disk, and closed the file. We ran the test on Chip twice, varying the block size between 1K and 8K. For small files, the performance of Chip versus the local disk is indistinguishable. However, as the file size increases, the performance gap between local disk and Chip widens, and the version of chip with the larger block size fares better. This data shows that block size matters in terms of maximum performance, and suggests that the disk is winning by being able to write large chunks of data faster than Chip can, and that a version of Chip with a block size greater than 8K might perform better compared with the local disk.

Unfortunately this is not the case. We observed that regardless of the amount of data being read or written by a client application, the NFS/SFS client breaks the read or write down into 8K chunks. This results in a performance loss due to the overhead incurred from additional RPC calls between the SFS daemon and loopback server, and between the LS and RS. We speculate that the overhead in these multiple resulting RPC calls is significant, as each 8k block is a new NFS call. This is fairly meaningful, since the NFS call must complete before the next 8k write can occur. This implies that our current sfsusrv-specific implementation will have trouble beating local disk on large continuous reads and writes because of the added overhead in this data split.

We attempted to measure the overhead that sfsusrv was incurring. To do so we measured the difference in speed between local disk and the disk-based version of sfsusrv. One can conclude that the difference in times between the disk-based sfsusrv and the local disk corresponds to the overhead generated by sfsusrv. However, we cannot simply subtract this overhead from our own results. (The reader might notice that doing so with the data given in Table 1 leads to the conclusion that Chip performs in negative time!) We believe that this inconsistency can be explained by the fact that the disk-based sfsusrv must have additional inefficiencies that make it slow compared with local disk, beyond merely the RPC overhead. Even so, we can conclude that sfsusrv is likely preventing us from realizing our full design performance, and may not have been the best choice for a system on which to base an implementation of Chip that has competitive performance. A different implementation of Chip, perhaps using VFS, or much more detailed

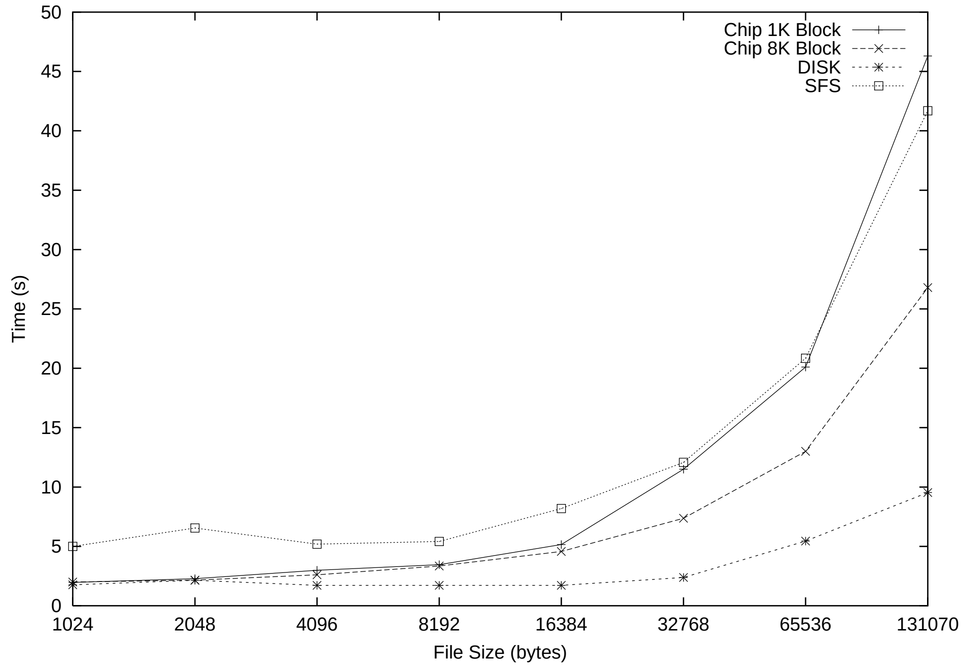

Figure 4:

profiling of our code would help us settle this question.

A final note on large data access: We believe that large files are best served by high bandwidth devices, be it network or disk. Chip, if run on a gigabit Ethernet, would theoretically run faster for large files than local disk access using the fastest SCSI hard drives on the market today. If we extrapolate trends in cost and technology performance improvement, we believe that the speed of physical disks will not grow as rapidly as the speed of networks. It is reasonable to assume that the price of a fast network will drop faster than the price of an equivalently fast mechanical hard drive.

Table 1 shows that for certain operations, Chip is slightly better than local disk, and is in the same order of magnitude as local disk (and an order better than sfsusrv) when it comes to executing random 1k reads on a very large data set. We found that the trends expressed in this table - namely that creates and deletes are always faster, and reads are always slower (with a curve similar to that in Figure 4) - remain valid as the testing parameters of block size, file size, and number of files are varied. One observation is that in theory, the access time across the network is an order of magnitude faster than disk access time. This implies that Chip might win on small sized random reads/writes. However, our tests indicate that there is some room for improvement in our system implementation - the average ping time between our local server and RS when we performed these tests was approximately .0003 seconds, and Table 1 hints that a roundtrip in our system is an order of magnitude slower. Based on this, we believe that a lot of the possible seek time win is being eaten up by overhead in our system. At the moment, we appear to be comparable with (instead of better than) disk.

7 Future Work

There remain a number of interesting issues to follow up on in the design of Chip. As a result of our performance analysis, we believe that the most important next step is to make Chip faster. In particular, we believe that eliminating the 8 Kbyte file transfer limit from the NFS/SFS client side would be a good first step that is likely to significantly improve Chip's performance. In addition, while the NFS loopback setup provides a convenient interface for the filesystem, the fact that the LS resides

Table 1:

	CREATE, WRITE 1000 1-KByte Files	`rm` 1000 Files	Read Random 1K
Unmodified SFSUSRV	71 s	30 s	0.021 s
Local Mounted Disk	26 s	18 s	0.0058 s
Chip	22 s	9 s	0.0096 s

on the local workstation means that the NFS calls between the SFS client and the LS are certainly unnecessary, and it is worthwhile to eliminate them. An implementation that skipped the sfsusrv and issued asynchronous calls directly to the RS is likely to be considerably faster.

Assuming we are successful in speeding up our implementation of Chip, the next thing we would like to work on is finishing a complete implementation of both replication and striping, and compare the performance of each under various file sizes and workloads. We have also considered the idea of a hybrid replicated/striped design which uses replication for small files and then automatically switches to striping for larger files.

8 Conclusion

This paper has described the design and initial implementation of Chip, a memory-based distributed network filesystem. The Chip filesystem provides users with a temporary filesystem that is replicated or striped to provide reliability and availability.

Chip performs extremely well for operations that primarily manipulate metadata, such as file creations, lookups and deletes. This is due to the fact that the metadata is stored entirely in the memory of the local machine.

There are a number of areas in which Chip could benefit from additional work. Most notably, the performance of Chip suffers from higher latencies and lower throughput than we would have expected of a system based only on RAM and the network. We have listed a variety of factors that may have contributed to the less than stellar performance we experienced.

There are a number of lessons that we take away from the experience of implementing Chip. The most significant of these is that the most readily apparent tool available for solving a problem is not always the best choice. Specifically, we believe that implementing chip as an SFS loopback server introduced a number of unnecessary latencies into the system and was not an optimal choice. We would like to see a version of Chip implemented as a direct interface to the local filesystem and compare the performance.

Overall, we believe that the Chip system design holds considerable promise but that our current implementation will require further work before it can compete with local disk-based filesystems.

9 Appendix

Our tests were run on soco.lcs.mit.edu and sweat.lcs.mit.edu. Sweat (which we used to run the RS) is a dual-processor 1194MHz Athlon with 512MB of memory. Sweat has a Intel Pro 10/100B+ Ethernet NIC. Swap space was not used during the test. Soco (used to run sfsusrv and all client operations, including all tests run in the analysis section of the paper) is a 200MHz Pentium Pro with 64 MBs of memory and a Digital 21140 Fast Ethernet card. The hard drive used in the local disk tests is comparable in bandwidth to that of the bandwidth of the Fast Ethernet (100Megabits/sec). The hard drive used is a Quantum Fireball TM2110S supporting a 10MB/sec sustained transfer, with a 12.5ms seek time.

References

[1] Callaghan, B., *NFS Illustrated*, Addison-Wesley, 2000.

[2] Castro, M., Liskov, B. Practical Byzantine Fault Tolerance, In *Proceedings of the Third Symposium on Operating Systems Design and*

Implementation, New Orleans, USA, February 1999

[3] Chen, P., Lee, E., Gibson, G., Katz, R., Patterson, D. RAID: High-Performance, Reliable Secondary Storage. *ACM Computing Surveys* 26(2): 145-185 (1994)

[4] Free-BSD Documentation Project, "FreeBSD Handbook", 1995-2001.

[5] Iftode, L., and Singh, J. Shared Virtual Memory: Progresses and Challenges.

[6] Hartman, J., Ousterhout, J. "The Zebra Striped Network File System", *ACM Transactions on Computer Systems*, Vol. 13, No. 3, Aug. 1995.

[7] Liskov, B., Ghemawat, S., Gruber, R., Johnson, P. and Williams, M., "Replication in the Harp File System", *Proc. Thirteenth ACM Symposium on Operating System Principles*, Oct. 1991, pp 226-238.

[8] Satyanarayanan, M., Kistler, J.J., Siegel, E.H. Coda: A Resilient Distributed File System, *IEEE Workshop on Workstation Operating Systems*, Nov. 1987, Cambridge, MA

[9] Walker, B., Propek, G., et. al. The Locus Distributed Operating System, *Proceedings of the Ninth ACM Symposium on Operating Systems Principles*, October 10-13, 1983, pages. 49-70

[10] Mazieres, David. A toolkit for user-level filesystems.

[11] Patterson, D.A., Gibson, G., Katz, R.H., A Case for Redundant Arrays of Inexpensive Disks (RAID), *Proc. of the 1988 ACM Conf. on Management of Data (SIGMOD)*, June 1988, 109-116.