Performance of the V Storage Server: A Preliminary Report

David R. Cheriton and Paul J. Roy

Computer Science Department
Stanford University

Abstract

Network file access efficiency is a key issue in a distributed system's performance, especially when many of the network nodes are diskless and rely on a shared network file server. We have designed and implemented a file server that uses the network interprocess communication of the V kernel for file access. This paper describes the basic design of the file server with emphasis on the performance-critical areas. We also give its performance under a variety of workloads and compare these measurements with results predicted by other modeling studies.

We conclude that the buffering and disk layout strategies we have used work well under load. Performance results are consistent with a previous modeling study that the file server processor is the most critical resource. However, our experiments with high load were limited by the small amount of buffering on the network interface, i.e. large numbers of packets are dropped at high load giving poorer than predicted performance.

1. Introduction

Shared network file servers are a basic part of most distributed systems. Typically, a file server is a machine with one or more disks connected to the local network and dedicated to providing file service to other nodes on the network. A shared file server may be used to augment the disk storage of individual workstations or to replace it altogether, as with diskless workstations.

Shared file servers have the advantages over local disks of:

- Economy of scale and sharing - one 500 megabyte disk is cheaper than 10 50 megabyte disks. Also, one copy of the file system code executes on the server, not on every client.
- Less environmental impact - file servers can reside in computer rooms so the noise, heat and power problems are away from individual users and workstations.
- Simpler maintenance and backup - maintenance of disks and backup of files can be handled the same as for a timesharing system, i.e. by an operator working in the computer room.

The major disadvantage is, of course, performance.[1] Sharing leads to contention between users and degradation for each user under load. With this in mind, we have been exploring how to build an efficient file server.

We have designed and implemented a software module we called the V *storage server* (because it provides an organized system of "storage" as a service). This server module executes as a *team*[2] of processes on top of the V kernel, using the V message-based interprocess communication facilities both internally as well as for its client interface. The storage server and V kernel are fairly portable. However, our measurements in this paper deal exclusively with this software executing on a SUN workstation-based file server [1]. This structure is illustrated in Figure 1.

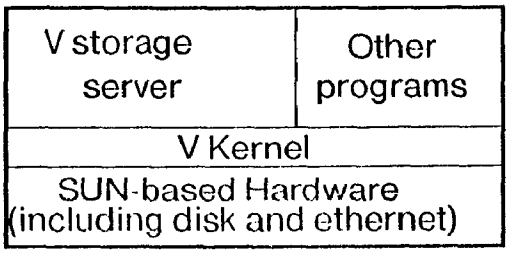

Fig. 1: V SUN-Based File Server Machine

This paper describes the basic design of the storage server module with emphasis on the performance-critical areas. We also give its

This work was sponsored in part by the Defense Advanced Research Projects Agency under contract N00039-83-K-0431.

[1] In some applications, users are also concerned about autonomy and data security, but we do not consider these issues here.

[2] A team is a collection of processes sharing the same address space.

performance under a variety of workloads and compare these measurements with results predicted by other modeling studies.

From these measurements, we have formed some preliminary conclusions that are guiding a refinement to the design. In particular, we conclude that the buffering and disk layout strategies we have used work well under load. We also confirm conclusions drawn in modeling studies that the file server processor is the most critical resource. However, our experiments with high load were limited by the small amount of buffering on the network interface, i.e. large numbers of packets are dropped at high load giving poorer than predicted performance. We discuss future plans for dealing with this problem in the last section.

The next section describes the basic design, including the disk data structures, the multiprocess structure of the storage server and the buffer mechanism. Section 3 gives our performance measurements with interpretation of results. We end with a discussion of our conclusions to date. This report is intended as a preliminary study of the performance of our hardware and software. Further refinement of the implementation and additional work on the V storage server is planned.

2. Design of the V Storage Server

The V storage server is a program that executes on top of the V kernel [4, 5]. It implements a fairly conventional tree-structured file system on one or more raw disk devices using the basic disk support provided by the V kernel. Clients access files using the V I/O protocol [5, 6] as a presentation and session protocol on top of the transport-level V interprocess communication. The V kernel provides transparent message-based interprocess communication, allowing clients to access the storage locally or remotely with no functional difference. Thus, our file server is distinctive in that it is not implemented inside the kernel, or as a standalone program on a dedicated machine, nor does it use a specialized protocol or protocol implementation. Instead, it is simply a multi-process program that can execute on any machine with a disk running the V kernel.

We divide the description of the storage server into three parts: the disk data structures, the multi-process structure, and the buffering. Besides being background for our performance evaluation, we believe the design is of interest in itself.

2.1. Disk Data Structures

The disk layout is almost identical to that of the Thoth file system [3], from which our design and implementation descends. It is also similar to the DEMOS file system [9]. In particular, files are represented as *extents* of contiguous blocks. Each extent is specified as a start block and a count of blocks in the extent. A file may be zero, one or more extents.[3] The block allocation is maintained in a bitmap, allowing the storage server to maximize the contiguity of files in creating, rewriting or growing files.

Files are described by file descriptor records that are stored in a file called the file descriptor file, with the first descriptor describing the file descriptor file itself. This is similar to the Unix i-node scheme [10, 12], except that the file descriptor area can grow and be accessed just as any other file (since it is a file - not a reserved area of disk).

This disk layout is given in Figure 2.

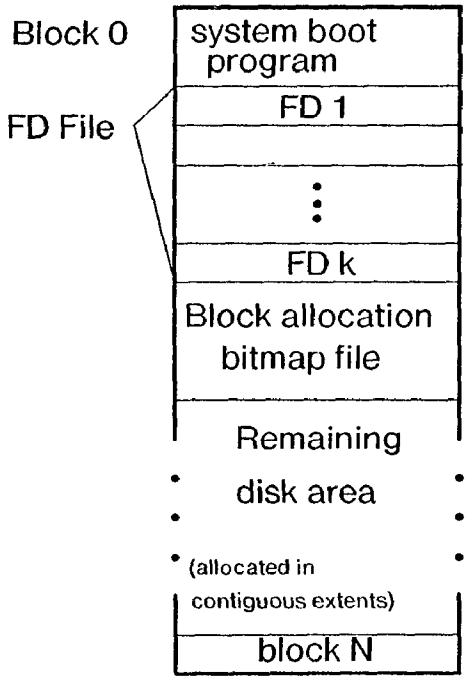

Fig. 2: Disk Layout

2.2. Multiprocess Structure

The storage server takes advantage of the message-based interprocess communication and inexpensive processes of the V kernel to use multiple concurrent processes internally in its implementation. The overall multi-process structure [3] is illustrated in Figure 3.

At the highest level, there are one or more main *storage server* processes that handle opening files, changing file size, and various directory operations. Once a file is opened, read and write requests to the file are handled by a *read/write(r/w)* server process. There are multiple r/w servers. The main servers distribute the file access load over the r/w servers. (We are planning to experiment with dynamic creation of new r/w servers but this paper only deals with a configuration-specified (static) number.)

[3] There is a limit in our current implementation of 9 extents.

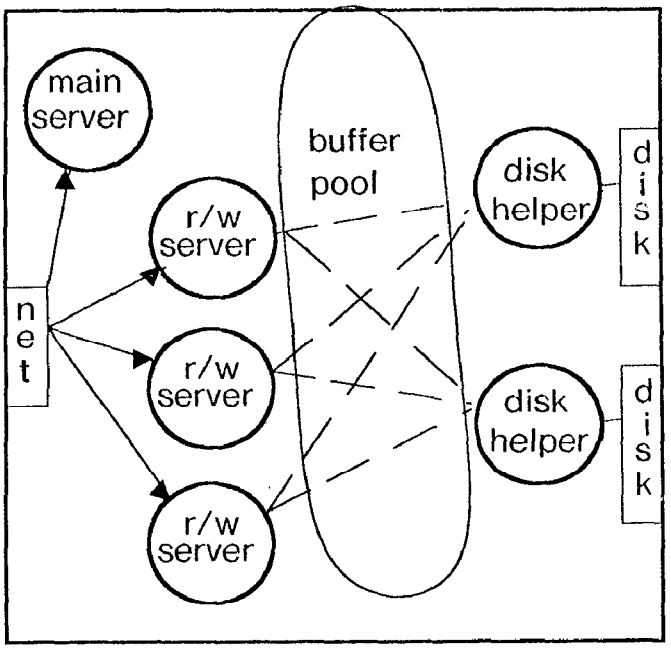

Fig. 3: Storage Server Multi-process Structure

Client read and write requests are sent directly to the r/w servers. On a read request, a r/w server searches the buffer pool in an attempt to locate the data in memory. If it succeeds, it replies to the request with the data immediately. Otherwise, it makes a record of the request, allocates one or more buffers, and then enqueues a request for the required data with one of the *disk helper* processes. On a write request, so-called write-behind is implemented by enqueuing the written data to a disk helper and replying immediately to the client.

Each disk helper process services a queue of disk requests. It executes an infinite loop of waiting for a next disk request, performing the disk operation, and (on a read) returning the request to the requesting r/w server. A disk request is presented to a disk helper as a record describing the operation plus a pointer to the buffer in which the data resides (on a write) or is to reside (on a read). Taking advantage of the shared address space between the r/w servers and the helpers, transferring requests and buffers between the r/w servers and helpers is done by enqueuing records in queues at either level and by passing pointers. Thus, this additional level of processes does not introduce copying overhead.

The disk helpers issue read and write operations on the raw disk using the V kernel device server [2]. The device server is implemented directly by the kernel as a pseudo-process (as opposed to a normal process like other system servers) and is accessed using the V I/O protocol.

We claim that the use of a multi-process structure such as this within a single address space allows a cleanly structured concurrent program that does not incur excessive cost in copying or buffering of data. In particular, considerable effort has been made to optimize the data path between client and disk. A key aspect of this is the buffering scheme.

2.3. Buffering Mechanism

The primary purpose of the buffering mechanism is to improve performance by reducing the disk access overhead. If access to disk pages is totally random and in single sectors, buffering cannot provide any performance improvement with a realistic amount of buffer space. For example, the buffer space on our file server is 1/500th of the space on the disks (1.5 megabytes versus 760 megabytes), giving a low expected hit ratio for random access. However, previous studies by Sager [11] indicate a disk access behavior that allows considerable benefit from buffering, and we have designed our buffering mechanism accordingly.

In particular, Sager's measurements indicate:

- Reading predominates over writing: on average 80 percent of disk I/O are reads. Variation has been observed depending on the type of data. For instance, standard system programs and data have 95 percent reads while temporary file space and swap space is accessed roughly 50/50 between reads and writes.
- Sequential file I/O predominates. Thus, the file organization on disk and the buffer management should be optimized for efficiently handling sequential access.
- Disk I/O queues are short. This implies there is little improvement to be gained by disk head scheduling, especially if the disk layout is optimized for sequential organization.

It is important to note that access is sequential to *files* (not to disk), so it is also of considerable benefit to layout files on disk to minimize disk access overhead with sequential file access. Thus, the file system allocates blocks using extents, giving a strong bias to sequential allocation of blocks on the disk. That is, sequential file reading translates into sequential disk reading (for the most part).

We recognize that buffering serves three main functions:

- **Cache** - data in the buffer pool need not be read off disk.
- **Prefetching** - the storage server can heuristically recognize sequential file access and prefetch file pages, reducing latency. Here, the buffer pool provides temporary storage for the file pages pending the client request for them. Similarly (of course), it provides so-called write-behind of pages out to disk.
- **Fragmentation/Reassembly Area** - large disk data transfers must be fragmented into units that fit into network packet sizes. The buffer pool provides large buffers into which one can transfer from disk, and then fragment into smaller network packets. For instance, we can transfer 8 kilobyte units (and larger) directly from the disk in one operation but the Ethernet is limited to 1536-byte packets.

The buffering mechanism is designed not only to provide each of these functions, but to recognize each class of use and apply different heuristics.

Cache behavior is detected as repeated access to the same disk pages. This is assumed to predominate with small read and write requests such as access to a directory or index file. Therefore, the storage server uses small buffers for caching data that have repeated references. This keeps the hit rate high relative to the size of the cache.

Sequential reading and writing is recognized by keeping a one-block history of access for each open file. If the current request is for the page following the previous request, the storage server assumes this file as being accessed sequentially. In this case, buffered data is unlikely to be read again once read or written. Thus, the storage server does not hesitate to reallocate buffers used during sequential access.

On sequential reading, the storage server initiates read-ahead beyond the current client request, thus overlapping disk operation with client processing. Taking advantage of the contiguous allocation of blocks in files, the storage server can read ahead numerous blocks in a single disk operation. For instance, currently the storage server reads ahead in 8 kilobyte units. This is more efficient in disk channel utilization than multiple disk reads with virtual sectoring (or disk interleaving)[4] as used on other systems. The contiguous allocation facilitates these large disk transfer units yet avoids the disk space wastage of large allocation units. For example, if the disk were allocated in 4 kilobyte units, measurements of Unix indicate [7] that more than 45 percent of the disk space would be wasted with internal fragmentation. Our use of a 1 kilobyte allocation unit reduces this waste (by the same measurements) to 11.4 percent.[5]

Finally, clients can issue large read and write requests in the V I/O protocol. For example, a client process can request a 1 megabyte read in a single request to the storage server which replies with the requested megabyte of data. Large writes are useful for checkpointing programs and performing file transfers. Large reads are useful for program loading, file transfers, and fast file access.

Client programs can recognize and realize the efficiency of volume file access operations, thus reducing the number of network packets, disk transfers, and server CPU cycles required per unit of data. The storage server supports this by using extra large buffers when they are available and the request size justifies it.

Our storage server project is now at the stage of having produced a basic working file server matching our design. It is in regular use at Stanford as a file server within our project. At this point, we embarked on some measurements of its performance, both to corroborate the intuition we had used in formulating the original design and to detect problems that would require further attention. It was also to guide future refinements of the software and hardware configuration.

[4]Our disk is actually interleaved at the disk controller level because the bus and memory cannot keep up with the non-interleaved transfer rate of the disk! Thus, we do lose some transfer capacity to interleaving, but less than if the CPU was issuing multiple disk operations for the same amount of data.

[5]This is based on Unix file system measurements. We would expect our file system to do at least as well, if not better because there are no indirect blocks and an equal amount of cost for internal fragmentation.

3. Performance Measurements

The configuration used for measurement was a SUN workstation [1] with a 10 MHz 68010 processor, 2 megabytes of memory, a 3COM Ethernet interface and 2 Fujistu Eagles connected through a Xylogics controller. The client workstations were identical except for being without disks. The Fujitsu Eagle disk rotates at 3961 RPM with 5 milliseconds track-to-track seek time, 35 milliseconds maximum seek time, and 7.5 milliseconds average rotational latency. Disks are formatted with 46 512-byte sectors per track and a 3:1 interleave ratio. The storage server was configured with 10 read/write servers, giving a high degree of concurrency in handling read and write requests.

We identified three different aspects of the file server performance, namely: unloaded elapsed time, maximum throughput, and expected response with different numbers of clients. Each of these is discussed in turn in the following subsections. We first describe how the measurements were made.

3.1. Measurement Methods

Measurements of individual file access operations were performed by executing the operation N times (typically 1000 times), recording the total time required, subtracting loop overhead and other artifact, and then dividing the total time by N. Measurement of total time relied on the software-maintained V kernel time which is accurate plus or minus 10 milliseconds.

Measurement of processor utilization was done using a low-priority "busywork" process on each workstation that repeatedly updates a counter in an infinite loop. All other processor utilization reduces the processor allocation to this process. Thus, the processor time used per operation on a workstation is the total time minus the processor time allocated to the "busywork" process divided by N, the number of operations executed.

Using 1000 trials per operation and time accurate plus or minus 10 milliseconds, our measurements should be accurate to about .02 milliseconds except for the effect of variation in network load.

Measurement of throughput was done running several instantiations of a client program which issues repeated file requests as fast as possible. We simply increased the number of such clients until no further increase in throughput occurred.

3.2. Elapsed Times on an Unloaded Server

The measurements in this section indicate performance when the file server is not loaded with requests from other clients.

The elapsed times for the basic kernel operations have been measured and reported elsewhere [4]. Here we include some kernel overhead times particularly relevant to this section. The basic message transaction time for sending a 32 byte request message and receiving a 32 byte response with a 1024 byte segment is 6.50 milliseconds. Message transactions that include moving 4 or 8 kilobytes over the network take 17.45 and 29.92 milliseconds, respectively.

Table 3-1 gives basic elapsed time read performance for the file server when accessed by remote clients. For operations requiring

disk activity, the seek time is about 10 milliseconds and the average rotational latency is 7.5 milliseconds. Disk transfer time (including interleave effects) for 1k, 4k, and 8k is approximately 1.3, 7.3, and 15.2 milliseconds, respectively.

File Operation	Elapsed Time per Operation
1K buffer read	7.80
1K no-buffer read	35.48
4K buffer read	18.69
4K no-buffer read	52.94
8K buffer read	31.23
8K no-buffer read	73.81
1Meg no-buffer read	5000.95

Table 3-1: File Access Elapsed Times (times in milliseconds)

The measurements designated "no-buffer" mean that the data was first read from disk. "Buffered" means it was in a disk buffer when the request was received. These measurements indicate several points. First, the cost per byte of file access drops with increasing data transfer units. As observed in a previous study [8], increasing the data transfer unit is the key to performance. However, its utility is limited by buffering constraints, decreasing benefit once overheads are well-amortized, and small file sizes (in many environments). For example, a 1 megabyte read can be done by the storage server using 8 128 kilobyte disk transfers and 1034 network packets, as opposed to 1024 1 kilobyte disk transfers and 2048 network packets, as might be the case for simple page-level access of the same amount of data. However, a 1 Mbyte read is fairly uncommon.

Finally, processor time is a significant portion of the cost, as shown by the cost for buffered operations for which no disk activity is required. With the 10 Mb Ethernet, a simple calculation indicates that the network (unloaded in these experiments) transfer time accounts for less than 20 percent of the elapsed time with buffered operations.

3.3. Throughput Measurements

We were also interested in the peak throughput of the file server, i.e. the maximum number of bytes that it can deliver per second. To determine maximum throughput, we wrote a client program that generates a steady stream of 4k read requests. We kept incrementing the number of instantiations of the client program until the throughput stopped increasing. (This occurred at 3-5 clients, depending on the buffer hit ratio.)

Different file request patterns produce different throughput. For instance, highly random read requests can defeat the buffering mechanism and also incur overhead for considerable seeking. In this paper, we use file request behavior that is characterized by the buffer hit ratio it generates.[6] Figure 4 gives the throughput we measured as a function of the buffer hit ratio.

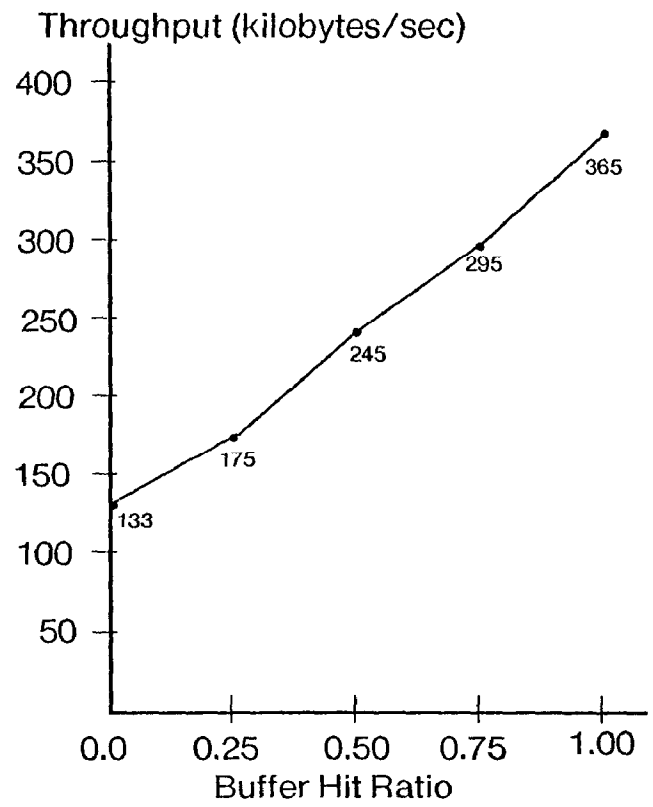

Fig. 4: Throughput as a function of buffer hit ratio

Unfortunately, our measurement for buffer hit ratio of 1 is deficient because the clients suffered from considerable retransmissions, presumably because the server's Ethernet interface receive buffers could not be emptied fast enough to keep up with the rate of requests. So, once this problem is alleviated (possibly with a network interface with more receive buffers), the throughput for buffer hit ratio of 1 should be higher.

At maximum throughput, ideally the storage server should be utilizing whatever is the bottleneck resource at 100 percent capacity. For low hit ratios, the file server throughput is limited by disk overhead. At higher hit ratios, the CPU becomes the major factor. Figure 5 plots the CPU utilization as a function of the buffer hit ratio. The fact that the utilization does not reach 100 percent at a hit ratio of 1 again indicates that our network interface problem mentioned above is restricting throughput.

It is interesting to note the problem we are observing, namely performance being curtailed by lost network packets, lost because of limited buffering on the Ethernet board, was recognized as a factor in our previous study [8].

[6] With read ahead and reasonable seeking behavior, the seek time is not a major factor compared to buffer hit ratio.

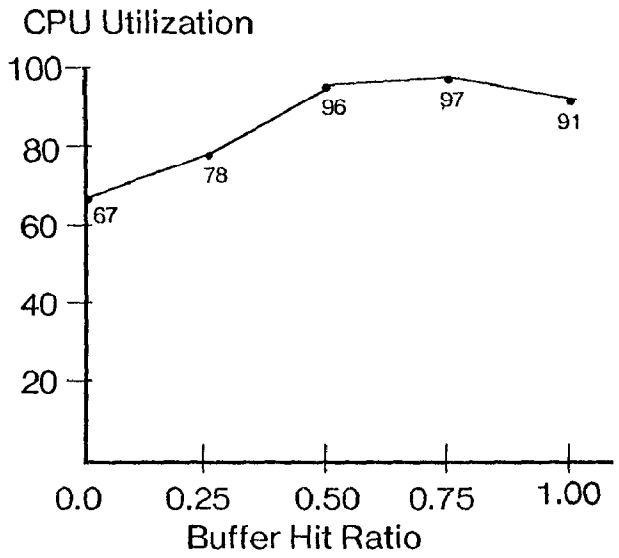

Fig. 5: Server CPU Utilization vs. Buffer Hit Ratio

3.4. Response with Different Numbers of Clients

A previous study [8] used a queuing model to predict average response time for file requests with different numbers of clients. We were anxious to compare the actual performance of our file server with the predicted performance.

Measurement of response with various numbers of "realistic" clients was done using multiple instantiations of a client program that generated a sequence of file access requests with an expected rate. That is, it randomly delayed between requests but averaged the data request rate desired. The average response time per request was calculated by subtracting the total delay and overhead time (determined in a separate timing test) from the total elapsed time and dividing by the number of requests.

With 1 kilobyte read requests and clients averaging one read per second, no measurable degradation was observed at up to the equivalent of 20 workstations. With clients generating 4 kilobyte read requests on average one per client per second and a 50 percent buffer cache hit ratio, the elapsed time rose from 40 milliseconds with one client to 47 milliseconds for 20 clients. Unfortunately, with larger numbers of clients, we started seeing high levels of packet loss, giving figures badly inflated by packet retransmissions. Our Ethernet interface has only 2 receive buffers making the real-time response in handling of incoming packets critical. We believe that improvements in the kernel network handler and (especially) a network interface with more packet buffers would allow the file server to handle more than 30 clients with only modest performance degradation, as predicted by the previous modeling study [8].

4. Conclusions

Good network file access performance is crucial to distributed systems, especially those using diskless workstations. We have presented the design of a file server and some measurements of its performance; both the elapsed time for individual operations as well as its performance under load.

Its behavior closely follows that predicted in some earlier papers on the V kernel [4] as well as some results based on queuing network models [8]. In particular, elapsed times for file access increase under load as the processor, being the most critical resource, saturates. Network utilization and disk utilization are much lower.

In performing these experiments, we ran into a danger raised in the previous modeling study, namely that lack of buffering at the network level can cause packet loss such that some gains from large transfer units are lost. Because our network interface has only 2 receive buffers and one transmit buffer, it is critical to schedule these well to avoid packets being dropped because the receiver buffers are full and to avoid small acknowledgement packets, etc. being excessively delayed at the network interface behind large multi-packet transfers. Note that this problem arises, not from the average arrival rate of network packets, but from the occasional clustering of packets from many clients in a short time interval.

We are endeavoring to improve the kernel handling of the network interface to reduce the amount of packet loss with this particular network interface. We are also acquiring some higher performance network interfaces with more buffering on board to further reduce this effect. At this point, we suspect that such a network interface is absolutely required for the file server to provide its full service potential and make maximum use of the CPU.

In this vein, performance we have measured is valid for file server hardware providing a processor similar to those of the client workstations. If one views this processor as providing the file system processing for K processors which would be distributed if each had a local disk, the saturation of the processor is not surprising. That is, if file system processing takes 5 percent of a processor with a local disk, then one processor should be able to support less than 20 workstations, and saturate at higher loads. Of course, the file server processor has the additional overhead of implementing the network protocol and the clients run slightly faster without this file system processing overhead.

Our results are dependent on the use of a single similar processor for the file server. We plan to extend this work to consider file server performance on multiprocessor machines as soon as the V kernel is available on one.

References

1. A. Bechtolsheim, F. Baskett, V. Pratt. The SUN Workstation Architecture. Computer Science Department, Stanford University, January, 1982.

2. E.J. Berglund, P. Bothner, K.P. Brooks, D.R. Cheriton, S.E. Deering, J.C. Dunwoody, R.S. Finlayson, D.R. Kaelbling, K.A. Lantz, T.P. Mann, R.J. Nagler, W.I. Nowicki, P.J. Roy, M.M. Theimer and W. Zwaenepoel. V-System Reference Manual. Computer Systems Laboratory, Stanford University

3. D.R. Cheriton. *The Thoth System: Multi-process Structuring and Portability.* American Elsevier, 1982.

4. D.R. Cheriton and W. Zwaenepoel. The Distributed V Kernel and its Performance for Diskless Workstations. Proceedings of the 9th Symposium on Operating System Principles, ACM, 1983.

5. D. Cheriton. "The V Kernel: A Software Base for Distributed Systems." *IEEE Software 1*, 2 (1984), 19-43.

6. D.R. Cheriton. A Uniform I/O Interface and Protocol for Distributed Systems. submitted for publication

7. M. McKusick, W.N. Joy, S.J. Leffler and R.S. Fabry. "A Fast File System for Unix." *ACM Transactions on Computer Systems 2*, 3 (1984).

8. E. Lazowska, J. Zahorjan, D.R. Cheriton and W. Zwaenepoel. File Access Performance of Diskless Workstations. Tech. Rept. STAN-CS-84-1010, Computer Science Department, Stanford University, 1984.

9. M. Powell. The DEMOS File System. Proceedings of the 6th Symposium on Operating System Principles, ACM, November, 1977. Published as *Operating Systems Review* 11(5).

10. D. M. Ritchie and K. Thompson. "The UNIX timesharing system." *Comm. ACM 17*, 7 (July 1974), 365-375.

11. G.R. Sager. Unix File Performance Notes. Unpublished Unix file performance measurements

12. K. Thompson. "UNIX Implementation." *Bell System Technical Journal 57*, 6 (July-August 1978).