# *sfs*: A Parallel File System for the CM-5*

*Susan J. LoVerso* *Marshall Isman* *Andy Nanopoulos*
*William Nesheim* *Ewan D. Milne* *Richard Wheeler*

CM-5 Operating System Group
Thinking Machines Corporation
245 First Street
Cambridge, MA 02142-1264

**Abstract**

This paper describes the creation of a UNIX-compatible file system with highly scalable performance and size. The file system is on the CM-5 backed by a scalable array of disks. Using the UNIX file system (UFS) from the SunOS 4.1.2 kernel as a base and modifying it to support Connection Machine (CM) operations, we have created a new file system, the scalable file system, or *sfs*. We discuss the CM operations we support, such as parallel reads and writes to the processing nodes of the Connection Machine, the use of NFS to support many partitions of processing nodes on the CM, support for very large file sizes (64-bit) and support for odd numbers of disk drives. The tradeoffs and decisions made during the course of this project as well as performance data for varying numbers of disk drives are provided.

## 1 Introduction

The data parallel programming model is the predominant programming paradigm on the CM-5. Typical applications deal with tremendously large data sets. When hundreds or thousands of SPARC processors require access to this data simultaneously, a high bandwidth I/O mechanism is necessary. The CM-5 requires a mass storage system that can support terabytes of data and transfer speeds of hundreds of megabytes per second. With today's technology, no single disk can fulfill those requirements; by using multiple disks and striping the data across all of them, we can meet the data rates required. We have created a highly scalable disk array (SDA) that provides great flexibility in I/O capabilities as well as flexibility in file system organization. These systems with the hardware and software described in this paper are currently running at dozens of customer sites.

CMOST, the operating system that controls the CM-5 derives directly from SunOS 4.1.2. It runs on the control processor (CP), which is a SPARCstation that controls access and use of the Connection Machine processing nodes (PN). The control processor regulates access to I/O devices from the PNs. Since the CP runs a derivative of SunOS, we wanted our file system to have UNIX-compatibility. Using UFS[5, 3] as a starting point, we created our own file system, called *sfs*, which is backed by the SDA. One of the primary functions that *sfs* serves is to provide the CP

*CM-5 is a registered trademark of Thinking Machines Corporation.
UNIX is a registered trademark of Unix System Laboratories.
*NFS*, SunOS, SPARC and SPARCstation are trademarks of Sun Microsystems, Inc.

with the information necessary to allow parallel I/O from the processing nodes to the SDA. UFS was inadequate for our needs in several ways:

1. It does not have support for file system block sizes and fragment sizes that are not a power of 2.
2. It does not support our parallel calls.
3. It does not have support for files larger than 2 gigabytes.
4. For files that are many megabytes in size, it produces highly fragmented files.

Due to the volume of changes required to support these four items, we chose to create a new file system type instead of making the changes in UFS.

In this paper, Section 2 presents a general hardware overview of the CM-5 and Section 3 gives an overview of the SDA architecture. These give the reader an understanding of how the hardware we are dealing with impacts the decisions we made. Then, Section 4 discusses the basic mechanics of parallel I/O and how it interacts with the rest of the system. Then Section 5 continues with a discussion of the support and design changes made to our *sfs* file system and to NFS in Section 6. Section 7 shows performance data showing the capability of this system. Section 8 concludes with a summary and future work.

# 2 CM-5 Architecture Overview

## 2.1 Hardware Architecture

The CM-5 may have tens to thousands of processing nodes. Each node has its own memory and may be executing programs written in either SIMD-style or MIMD-style. Access to the nodes is supervised by the Control Processor.

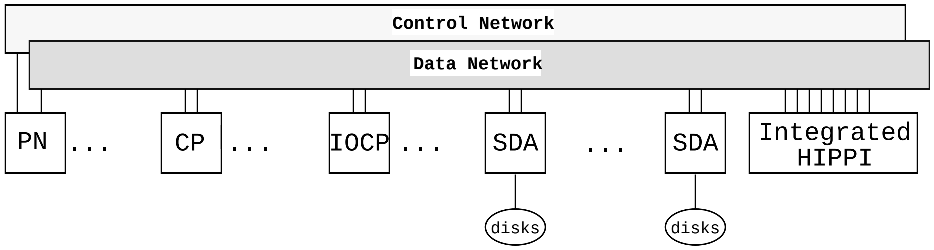


Figure 1: CM-5 Hardware Overview

Two scalable communication networks connect all CPs and PNs as shown in Figure 1. The Control Network handles communications that affect or involve all the nodes, such as broadcasts or synchronization operations. The Data Network handles data traffic – generally point-to-point operations. The SDA is also attached to these networks. The PNs and CPs communicate via the Control Network and all the I/O related activity occurs over the Data Network. The network design provides low latency for transmissions to near neighboring addresses, while preserving a high, predictable bandwidth for more distant communications[10]. The CP on which the SDA file system resides is designated as the IOCP.

High speed external networks, such as HIPPI, are connected directly to the Data Network. Slower external networks, such as Ethernet or FDDI, connect into the CM-5 via the control processor.

In order to provide a scalable high performance disk subsystem, one must spread data across multiple disks. For protection of data against disk failures, the SDA uses a Redundant Array of Inexpensive Disks (RAID) [7, 9] model. The Data Network within the CM-5 carries 20-byte packets for I/O; a 4-byte header precedes 16 bytes of data. A RAID level 3 implementation implies that a small amount of data is written to each disk, and parity data is written for each *disk stripe* of data. With our packet size, it is natural to choose 16 bytes as the amount of data written to each disk before moving on to the next. With each stripe of data, 16 bytes of parity information is stored on an additional parity disk. The parity disk allows reconstruction of any data which is missing due to a disk failure. This RAID-3 implementation is geared toward reads and writes of large amounts of data.

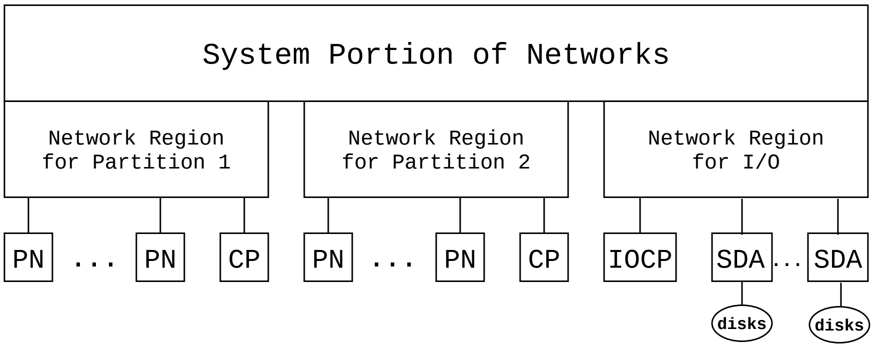


Figure 2: Partitioning the CM-5

The processing nodes of a CM-5 system can be configured into one or more partitions as shown in Figure 2. Each partition is assigned a partition manager – a CP that bears the responsibility for managing the process executing in that partition. The operating system configures the Control Network and Data Network to match the partition structure.

The I/O devices and IOCPs reside in areas of the network address space which are outside any specific partition. Therefore, the I/O devices appear as a global shared resource. There is always additional network capacity for carrying data between partitions and I/O devices. Hence, system-wide data traffic does not interfere with or impede traffic that stays within a partition.

## 2.2 Software Architecture

A parallel program runs within a partition consisting of a CP and a set of PNs. When performing file system operations, the CP is the client requesting the I/O operation and the IOCP is the file server. The client/server model is well suited to this environment because there can be many partitions sharing the same file system. The control part of the transaction is handled by the NFS protocol. (See Section 6 for more details). The data, however, flows directly between the PNs and the I/O devices over the Data Network. The separation of control and data is done because of the very different latency versus bandwidth tradeoffs required by the two types of data movement. The separation of control and data is similar to the mechanism used on the CM-2 I/O system[2] and the IEEE Mass Storage System Reference Model[1].

A program called the *time sharing daemon* runs on the CP and controls processes requesting

CM-5 resources. It works in conjunction with the UNIX kernel and schedules processes to run on the PNs. In some ways, it can be considered an extension of the kernel. When a CM-5 process requests an I/O operation to the SDA, the time sharing daemon acts on its behalf. The time sharing daemon, using the user's file descriptor, makes a parallel I/O system call to the kernel to access the SDA via the file system. However, the time sharing daemon and the PNs are wholly responsible for the actual I/O operation. This file system transaction will be covered in more detail in Section 4 and Section 5. Now let's take a closer look at the architecture of the SDA.

# 3 SDA Architecture

One or more units known as Disk Storage Nodes (DSN) comprise the CM-5 disk subsystem. As illustrated in Figure 3, each node contains eight 1.2 gigabyte SCSI disk drives, four SCSI channels, an 8 Mb data buffer, a 20 Mb/second Network Interface and a SPARC processor with its own 8 Mb memory. Each DSN runs a controller microkernel that mediates the I/O between the CM-5 and the drives. DSNs are packaged in units of three as backplanes. Therefore, systems have disks in quantities of 24, 48, 72, etc. Typical configurations might use 22 or 46 or 70 data disks, one parity disk, and one spare disk. By adding more DSNs, you add a balanced amount of bandwidth from the disk through the network interface. By increasing the number of Data Network addresses, which happens automatically when you add more DSNs, you increase the overall Data Network bandwidth as well.

As shown in Figure 1 in Section 2, the SDAs are connected to the Data Network. Each DSN occupies enough network address space to guarantee its sustained 20 Mb/second transfer rate to anywhere in the CM-5.

## 3.1 Logical Devices

The file system interacts with an abstract virtual disk implemented on top of a set of disks. This set of disks is termed a *logical device*. A Logical Device (LD) consists of an arbitrary number of physical disks grouped together into a RAID 3 disk system. Data is striped 16 bytes per disk. The LD appears as one large disk to the file system. The system administrator edits an ASCII file to select which physical disks become a member of a logical device. There may currently be up to 255 data disks and an optional parity disk in the LD. The configuration of the logical devices is downloaded to the CMOST kernel and to all the DSN kernels running on each of the DSN controllers.

The configuration information describes each logical device. There are four components of the system that require this information:

1. The device driver in the CMOST kernel.
2. The DSN microkernel.
3. The timesharing daemon.
4. The PN microkernel.

The "master" version of the configuration information resides in the CMOST kernel. All other components make queries to the kernel to retrieve it. We use a timestamp mechanism[4] to maintain consistency of this information across the system. An example of some of the information maintained in the configuration tables includes:

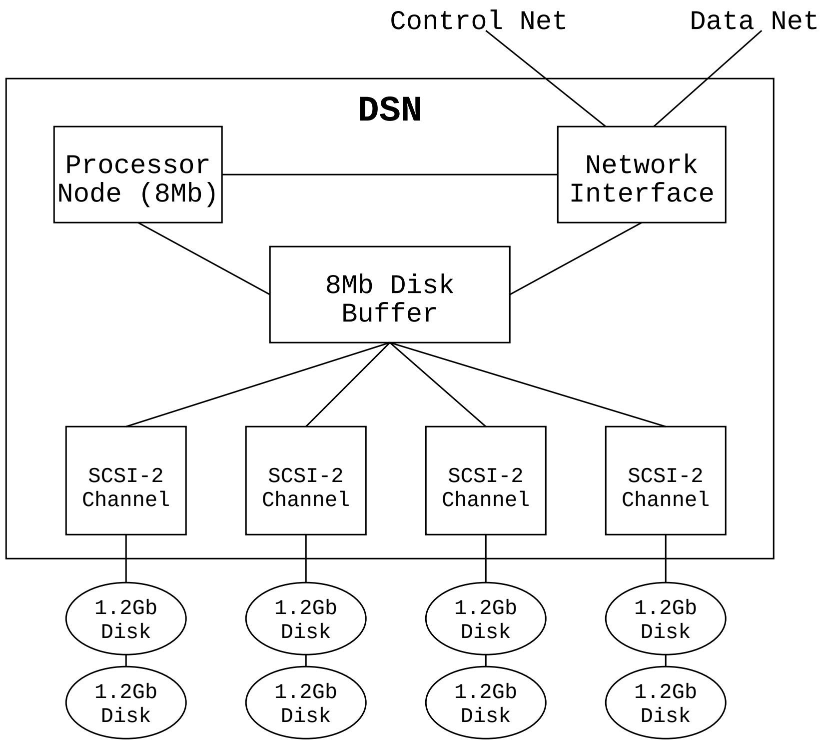


Figure 3: DSN board

- The number of disks in the logical device.
- The network addresses of those disks.
- Whether this logical device has a parity drive.
- The timestamp associated with the logical device.
- A flag indicating the device's state (such as online).
- The network address of the IOCP for this device.
- The `dev_t` representing this device.

Logical devices are global across the CM-5. Lookups of this information can be done using the `dev_t`.

## 3.2 DSN Microkernel

The SPARC processor on the DSN board runs code which serves as the disk controller firmware for the drives connected to that DSN board. The controller microkernel translates I/O requests for a logical device into specific SCSI controller commands. If a logical device spans more than one DSN, then one DSN is the master of the LD and the other DSNs are slaves. An I/O request is directed to the master DSN which forwards the command to the slaves.

For a read operation, the master and slaves transfer their data asynchronously. As the slaves finish, they send a done message to the master. The master sends a done message to the source when all DSNs finish. For a write operation, the master and each slave enable their hardware buffers to receive the data. Each slave sends a message to the master indicating how much data it can receive. The master sums this information and sends the total to the source. The source then transmits the data to each DSN. This process iterates until all the data is transmitted. If any of the DSNs detect a disk error, then the done message sent to the source indicates the logical disk in error and the block where the error occurred.

# 4 Parallel I/O

This section describes parallel I/O in general. The current implementation, described below, uses the time sharing daemon to dispatch a user's parallel I/O request. Scheduling constraints dictated that the time sharing daemon handle parallel I/O. Our eventual goal is to move parallel I/O completely into the kernel and remove some of the latency and overhead. There are several other issues, such as supervisor access to the Control Network, that make this work more than trivial.

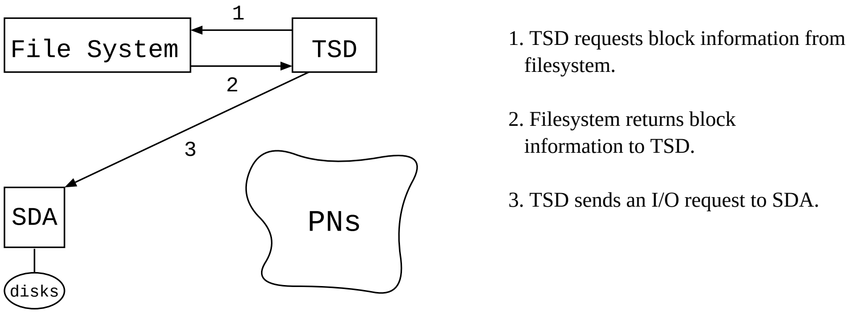


Figure 4: Beginning of Parallel I/O

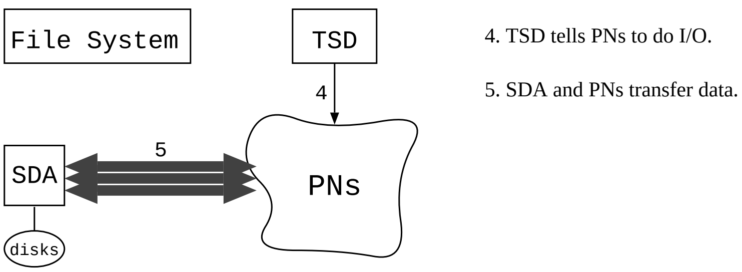


Figure 5: Transferring the Data

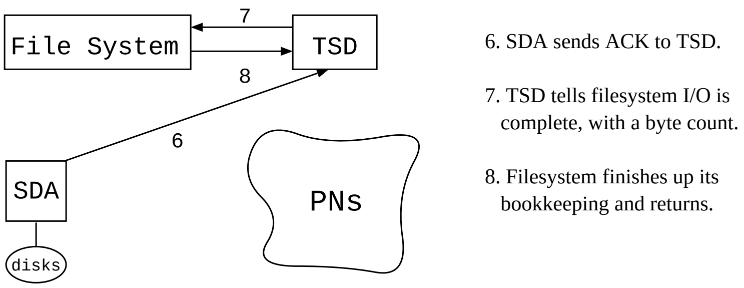


Figure 6: Finishing the I/O

A parallel I/O request really contains three distinct operations. The first, illustrated in Figure 4, shows the time sharing daemon asking the file system for the list of disk blocks relevant to the I/O request. The second portion, shown in Figure 5, indicates that once the I/O is set up, the time sharing daemon tells the PNs to commence the I/O, and a high bandwidth transfer takes place directly between the SDA and the PNs. The final operation, seen in Figure 6, cleans up the remainder of the I/O. The SDA sends an acknowledgment to the time sharing daemon, who then finishes the I/O with the file system. The file system unlocks the inode, increments the file offset and cleans up from the I/O.

The major drawback in this scheme is the latency involved in initiating a parallel I/O operation. This overhead is due to the setup time involved in passing the ts-daemon the user's file descriptor and to the time spent by the ts-daemon retrieving the block information from the kernel and then coordinating the transfer between the SCN's and and the processing nodes. Once the data starts flowing between the SDA and the processing nodes, the disks perform quite reasonably as described in Chapter 7.

# 5 Scalable File System

Since we began with a goal of a UNIX-compatible file system, it seemed logical to begin with the UFS sources and modify them to suit the needs of our system. Basically, four major areas needed support and modification in our new file system:

1. Support for large files.
2. Support for our parallel read and write calls.
3. Handling of file system block sizes that are not a power of two.
4. Slight modifications to block allocation.

Our intent was to preserve the current performance of UFS and not impact the standard read or write paths with our changes. We can boot an *sfs* file system on a Sun workstation on a SCSI disk. Timing tests show that our file system performs the same as UFS, when running to the same disk. While not surprising, it was pleasing to know we had no negative impact on general performance in the file system.

## 5.1 Large Files

Supercomputer applications typically deal with very large files. Data sets of tens of gigabytes are not uncommon. Therefore, file system support for very large files is imperative. While the Berkeley Fast File System itself does nothing to restrict file sizes, higher level file I/O support within SunOS limits file sizes and offsets to sizes representable within 32 bits.

In order to support very large files, several changes were required to SunOS above the file system level. One change was to add a *vfs* flag, VFS_BIGFILES, indicating that the file system supports large files. Other changes relate to maintenance of position within a file, and to representation of the absolute size of a file. We address file offset maintenance first.

In order to keep track of file offsets not representable in 32 bits, we added a new fundamental system data type which we called a `longlong_t`. This structure is a composite type of two 32 bit integers laid out most significant word first, for compatibility with the Gnu C compiler GCC. An additional type, `offset_t`, is defined simply as a `longlong_t` for clarity when dealing with file offsets.

By including an ANSI `long long` within the type, user space applications including kernel header files can be compiled with an ANSI compiler, allowing them to use long long arithmetic directly. We did not feel comfortable switching the compiler used to build the UNIX kernel itself at this point, so kernel operations on `longlong_t` data structures are handled explicitly in the code.

Only two system data structures required changes to support large file offsets. First, the file offset field within the file structure was changed from an `off_t` to an `offset_t`. The *uio_offset* field within *struct uio* was also increased to 64 bits. By adding macros to each modified header file defining the old structure element names to refer to the low order word of the new 64 bit offset fields we could minimize the changes required to the system outside the system call layer and the SDA file system itself.

In addition to extending the data structures, minor changes were required to several routines at the file and vnode layers of the kernel. Any routines above the file system layer which manipulated file offsets needed to be taught about the new 64 bit types. Specific examples of code requiring slight changes include uiomove() and seek(). We also added a new llseek() system call to allow programs to manipulate file position beyond 2 Gigabytes.

Slight changes to the core system were also necessary to handle very large files. In SunOS, file system attribute information is contained in a *struct vattr*, which carries *stat* information in a file system independent format. The file size component of this structure was extended to 64 bits. As with file offsets, we took care here to allow existing kernel code to continue to access the low order portion of the file size field with the same name, eliminating the need for extensive changes within non-*sfs* file system code.

Special enhancements for *sfs* were required in performing serial I/O when the current seek position was not representable in 32 bits. SunOS performs serial I/O operations through the VM system. The high level file system entrypoints (the *VOP_RDWR* routines) perform any necessary block allocation, and then copy data between user space and a kernel virtual memory segment which is backed by the file system. For each page within this segment, the VM system tracks the inode and file offset mapped by this page. Unfortunately, this file offset is stored as a 32 bit quantity. Extending this field to 64 bits required much more extensive surgery on the SunOS VM system than we cared to undertake. Instead, we chose to handle serial file I/O differently for offsets requiring 64 bit representation.

The *sfs* serial I/O routine (SFS_rwip) handles serial I/O for file offsets up to 2Gb exactly as is done within the SunOS UFS file system. By following this path we can support mapped files and text paging within the *sfs* file system just as within UFS, as long as the file offset remains representable within 32 bits. For large files a different path is taken. I/O is done through the buffer cache rather than through the VM system. The buffer cache tracks blocks by their block number in DEV_BSIZE (512 byte) units. By maintaining the cache by block number we effectively get 8 additional bits of file offset maintenance in the cache, allowing *sfs* to support read and write operations on file offsets up to 2**39-1. Mmap operations on files at offsets not representable in 32 bits are not supported.

## 5.2 Parallel I/O

One of the key features of the SDA File System is its ability to support data parallel I/O operations. Parallel I/O is implemented as a combination of code within the time sharing daemon, and file system support code. The file system provides the mapping between file name and block lists, and handles all serial I/O. The timesharing daemon actually performs all parallel I/O operations.

Two new system calls were added to support this function. CM_read_raw() and CM_write_raw() are very similar in that both calls return lists of blocks to the time sharing daemon, with only CM_write_raw() performing block allocation. Each of these calls is executable by the time sharing daemon process only.

The time sharing daemon performs parallel I/O operations on behalf of user programs. Using a file descriptor passed from the application, the time sharing daemon calls the *sfs* through the CM_read_raw() and CM_write_raw() system calls to obtain a list of blocks involved in the I/O operation.

The first call to CM_read_raw() or CM_write_raw() places a lock on the inode involved. This lock is held for the duration of the entire parallel I/O operation, being released only when the timesharing daemon finishes the parallel I/O by calling CM_read_raw() or CM_write_raw() again.

A parallel I/O operation generally involves a large amount of data. As it uses the existing UFS on-disk structure, the *sfs* represents files as lists of individual blocks. But unlike standard UFS disk drivers, the SDA is optimized for transferring large contiguous block ranges, not for individual block reads or writes. The CM_read_raw() and CM_write_raw() system calls support this by coalescing the blocks returned by the file system block lookup routines into lists. The block lists returned to the time sharing daemon are thus effectively a set of extents to be used in performing the I/O operations[6].

## 5.3 Odd File System Block Sizes

We use the term *block stripe* to refer to one 512 byte block on each data disk. The term *disk stripe* is used to describe a logical chunk of data spread evenly with 16 bytes of data on each disk in the array. Therefore, the block stripe size of a 46 data drive logical device is 23 Kbytes, and the disk stripe size of such a configuration is 736 bytes.

UFS was designed for disk systems where the file system logical blocksize is some power of two multiple of the fundamental disk block size. With the *sfs* we were faced with the situation where the fundamental disk block size could be substantially larger than the desired logical file system blocksize, and could under many conditions bear no reasonable mathematical relationship to it whatsoever. A number of changes were required to support this.

Adding support for odd blocksizes involved changes in two areas, file system macros and serial read and write processing. First all the file system macros which converted bytes to blocks, blocks to fragments, etc required modification to use division and modulus instead of shifts and masks. These changes were fairly straightforward and didn't involve major changes to the file system code itself. In benchmarking the system on a Sun 4/300 with a SCSI disk we found that this had little or no impact on file system performance.

Enhancing the serial I/O routines to handle odd block sizes required more effort. As mentioned earlier, SunOS performs serial I/O operations through the VM system. The high level file system entrypoints (the *VOP_RDWR* routines) perform any necessary block allocation, and then copy data between user space and a kernel virtual memory segment which is backed by the file system. The copy causes a fault, resulting in the file system page handling routines ( *VOP_GETPAGE* or *VOP_PUTPAGE*) being called to perform the actual I/O. But by this time the I/O operation has been broken up into some sequence of PAGESIZE units, regardless of any effort by the user application to do I/O in file system blocksize units!

Pages do not map nicely onto file system blocks which are not a power-of-two number of bytes in size. The challenge in supporting serial I/O efficiently with odd blocksizes is to group page I/O's together to form some integral number of full block I/O's. The *sfs* getpage and putpage routines attempt to do disk I/O in chunks of the least common multiple of the machine pagesize and the file system blocksize. A second problem is that a single page can span several logical blocks within a file. A single page may require two separate disk I/O's to complete. By limiting the minimum file system blocksize to be at least as large as the system pagesize we avoid the situation where three I/O's may be required to handle a page.

### 5.4 Block Allocation

The *sfs* uses the UFS on disk structure and allocation policies. When rotational delay is zero, the UFS policies reduce approximately to a cylinder group based extent allocator. The file system will attempt to allocate new blocks immediately following the last block in an existing file. The UFS policies generally allocate blocks for different files from different cylinder groups, with the exception of files within the same directory, which are allocated together.

This policy, in combination with extent coalescing in the parallel I/O routines, suits *sfs* fairly well. Problems arise, however, when simultaneous file extensions are done to files being allocated within the same cylinder group. Under this situation the files are not allocated as extents but instead have their blocks intimately mingled together. For parallel I/O operations this results in a single I/O requiring many extents, greatly reducing performance.

Although work in this area is not yet complete, an approach involving applying an allocation lock on a cylinder group basis for the duration of a file extension operation has shown some potential for increasing the average size of extents allocated under these conditions.

## 6 Network File System

The CM-5 has several SPARCstations acting as partition managers. Therefore, it seemed natural to have partition managers mount *sfs* file systems over NFS[8]. Several reasons motivated this choice over implementing our own mechanism for remotely accessing the file system. Foremost, NFS is a widely used and well understood method. The extensions necessary to support our parallel calls would be minimal. Also, since *sfs* is UNIX-compatible, users on any workstation could remotely mount and access the SDA.

At mount time, the partition manager, as a client, determines whether the remote file system is a CM-5 file system. This information is maintained in a bit in the *mntinfo* structure. If the bit is set, the parallel I/O system calls are allowed. Otherwise an error condition is returned to the user.

Two new RPC calls were added to support the CM_read_raw() and CM_write_raw() parallel I/O system calls from the time sharing daemon. Since the parallel I/O calls split into a "begin" and an "end" half, the new RPCs, *RFS_PARIOB* and *RFS_PARIOE*, are functionally split along those same lines. The first returns the disk block information to the time sharing daemon, and the second completes the operation. The same locking issues that are present in *sfs* apply here as well for the rnode. The rnode is locked when the "begin" call completes, and it does not become unlocked until the "end" call.

Additional RPC calls were necessary to handle file sizes greater than 2 gigabytes due to increased sizes in some of the message fields. We needed to support 64-bit size fields while maintaining compatibility with existing NFS protocols. Five calls required these changes:

1. Getting attributes, *RFS_GETATTRL*.
2. Setting attributes, *RFS_SETATTRL*.
3. Reading, *RFS_READL*.
4. Writing, *RFS_WRITEL*.
5. Lookup operations, *RFS_LOOKUPL*.

We could not change the existing versions of those calls, because then our NFS would not be compatible with the rest of the world for standard UNIX calls. So, we added new 64-bit versions of those calls. The need for 64-bit version of the first four calls is straightforward. Lookups needed to support 64-bit files because the result of a lookup operation returns attributes about the file it looked up, including the file size. As mentioned in Section 5, each mount point may have a flag set indicating it supports "big" files. If that flag is set, then the client uses the 64-bit versions of those calls where appropriate.

# 7 Performance

Our goal was to provide a scalable file system capable of running on top of the SDA within a CM-5 system. The performance goal for the first release of the *sfs* software was to obtain a rate of 1.5 Mb/second per disk on read operations and a rate of 1 Mb/second per disk on write operations across a wide range of disk configurations. We have achieved these performance goals on file systems spanning from 16 to 118 disk drives. Although a system with 118 data drives was the largest we had available to run the tests, the hardware and software systems can support much larger configurations.

Our test measured the amount of time it took to execute the parallel read and write system calls. Each performance test was run ten times for each data point. The highest and lowest performance result were discarded and the other eight trials were averaged and plotted here. All the performance numbers presented in this paper were measured reading and writing regular files, not using the raw device. Each test was run on a partition with 128 nodes. The system was up and running in multi-user mode, although our program was the only parallel process running.

A logical device generally contains data drives and one parity drive and one spare drive to use in case of a disk failure. Therefore one to five backplanes of SDAs typically contain 22, 46, 70, 94 and 118 data disks respectively. We also show performance numbers for power of two data disk configurations containing 16 and 32 data disks. The disks are SCSI based IBM Model 0663E15, which are able to sustain raw transfers at approximately 2 Mb/second on reads and about 1.8 Mb/second on writes.

Figure 7 shows how the effective transfer rate for read operations varies according to the size of the transfer for file systems utilizing various disk configurations. The graph shows that read performance improves as we continue to add disks drives over a range from 16 to 118 drives. The highest read transfer rate achieved was over 185 Mb/second when using 118 data drives. Each curve periodically shows a slight drop in performance at transfer sizes that are somewhere between 64 and 256 Mb. This slight decrease in performance at those specific values is due to the additional overhead involved in switching to a different indirect block. File systems with a power of 2 number

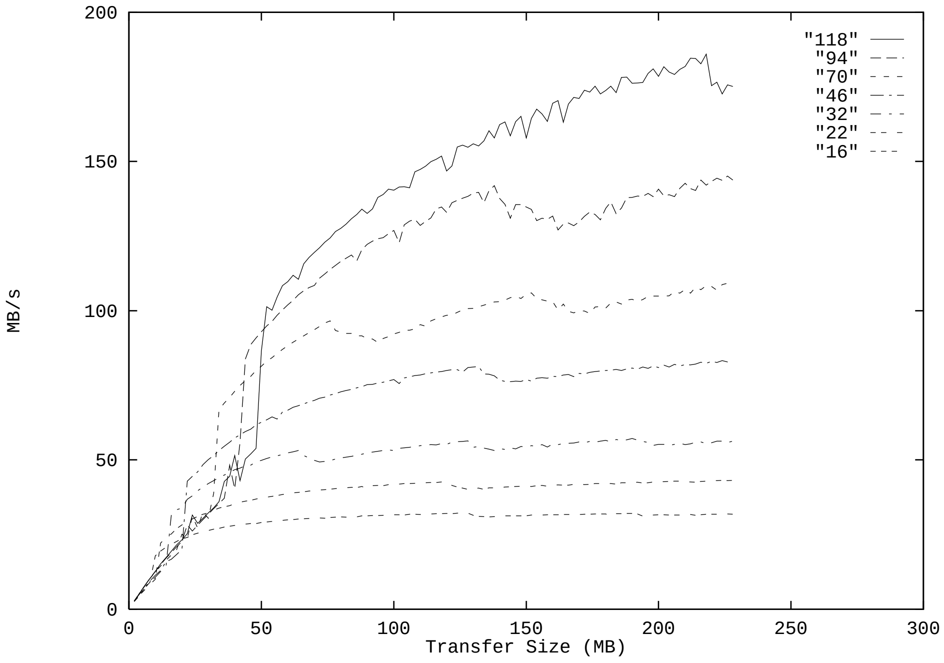


Figure 7: Read performance with 128 nodes

of disks have a file system block size of 16Kb. This allows for 4096 block addresses in each indirect block. Each indirect block can thus address 16Kb $\times$ 4096 = 64Mb of data. This is most clearly shown in the line for 32 disk drives. For the file systems with a non power of two number of data disks, the file system block size is somewhere between 16Kb and 32Kb depending on the number of disks. Notice that for small transfers, larger disk configurations perform more slowly. This is shown by the crossing lines of the graphs near the origin. This behavior reflects additional latency in the larger configurations, due to their having additional DSN controllers and hence, needing to send more messages. The next release of the system software will decrease this latency with the goal of eliminating any differences in performance at the small transfer sizes.

Figure 8 shows how the effective transfer rate for write operations varies according to the size of the transfer for file systems utilizing various disk configurations. Results are presented for the same disk configurations as reads. The shapes of the curves are similar between reads and writes with the write performance topping out around 1 Mb/second per disk while the read performance tops out around 1.5 Mb/second per disk. Write performance is less than read performance for a number of reasons, including the fact that the physical disks perform reads faster than writes and also the buffer size limitations on the DSN boards. Again notice the slight dips due to indirect block access. Latency for large disk configurations causes the lines to cross near the origin for writes as well.

Figure 9 shows how the maximum effective transfer rate **per disk** for read and write operations varies according to the number of disks used for file systems with a varying number of drives. The amount of data transferred per disk is held constant at 2 Mb for this graph. (I.e. 32Mb for 16 disks, 44Mb for 22 disks, etc.) An optimum line for this graph would be a perfectly flat horizontal line. Our data shows nearly optimum scaling. This graph clearly shows that we have achieved scalable file system performance across almost an order of magnitude change in the number of disk drives.

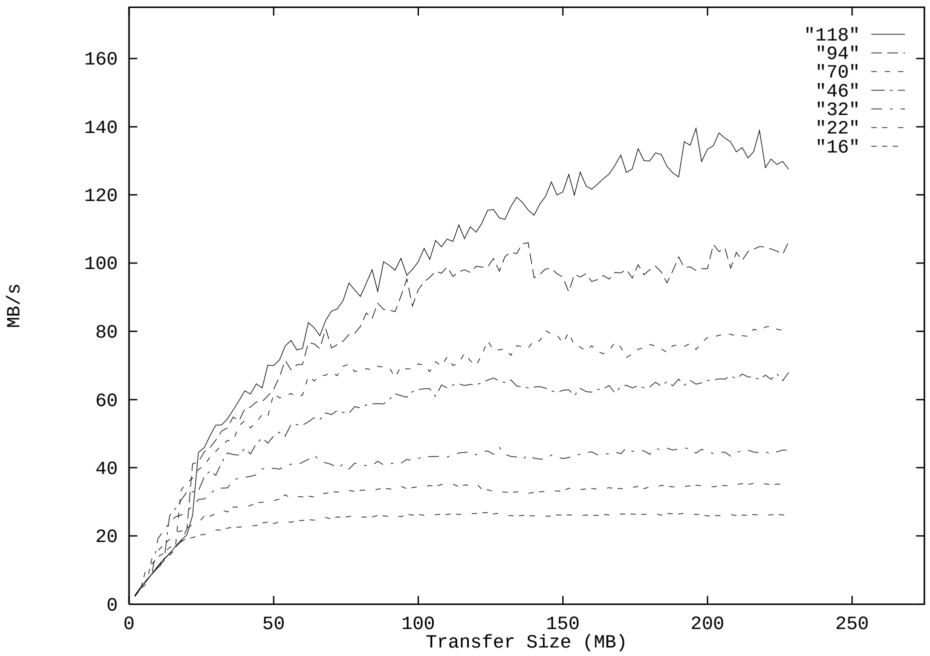


Figure 8: Write performance with 128 nodes

The performance of the SDA file system is determined by several factors, including the time required to execute the file system code on the CP and the time required to process the request on the DSN controllers. These times remain relatively constant regardless of the partition size and disk configuration. The number of disks used and the number of PNs in the partition determine the effective transfer rate, which allows the performance of the file system to scale according to the size of the machine. By allocating blocks contiguously we are able to sustain transfer rates which are a significant percentage of the raw disk transfer rate.

# 8 Conclusions and Future Work

We have shown that you can build a UNIX-compatible file system which is truly scalable in both size and performance. For large data transfers 118 disks have been used in parallel to achieve extremely high transfer rates on a single I/O. Modifying the disk block allocation algorithms to give us large contiguous extents gave us excellent performance results while not adversely affecting the standard read and write paths. We have shown that we have met our performance goals for reads and writes.

We recognize some limitations and latency issues in our implementation. We would like to move the parallel I/O system calls into the kernel and remove the overhead of having the time sharing daemon coordinate the I/O. We would like to address the latencies involved in coordinating many DSN boards. Obviously, we are also investigating new hardware, faster and bigger disk drives to increase the performance of the system.

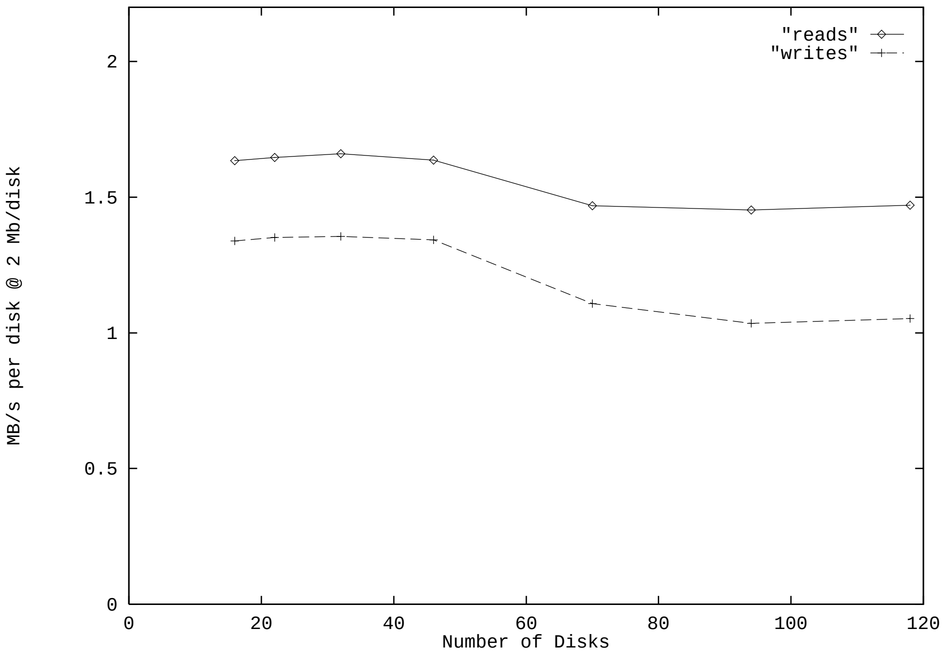


Figure 9: Per-disk read and write performance with 128 nodes

# 9 Acknowledgments

We would like to thank John Smith, Tom Moser, Eric Rowe, Soroush Shakib, Eric Sharakan, Andre Vignos, Roger Lee, Mark Bromley and David Taylor. Their help and contributions to this project were key to its success.

# References

[1] S. Coleman and editors S. Miller. Mass Storage System Reference Model: Version 4. IEEE Technical Committee on Mass Storage System and Technology, May 1990.

[2] B. Kahle, W. Nesheim, and M. Isman. Unix and the Connection Machine Operating System. In *Workshop Proceedings, USENIX Workshop on Unix and Supercomputers*, pages 93–107, Pittsburgh, PA, 1988. USENIX Association.

[3] S. Leffler, M. K. McKusick, M. Karels, and J. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System.* Addison-Wesley, 1989.

[4] S. LoVerso, N. Paciorek, A. Langerman, and G. Feinberg. The OSF/1 Unix Filesystem (UFS). In *Conference Proceedings, 1991 Winter USENIX Technical Conference*, pages 207–218, El Toro, CA, 1991. USENIX Association.

[5] M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry. A Fast File System for UNIX. *ACM Transactions on Computer Systems*, 2:181–197, August 1984.

[6] L. W. McVoy and S. R. Kleiman. Extent-like Performance from a UNIX File System. In *Conference Proceedings, 1991 Winter USENIX Technical Conference*, pages 33–43, El Toro, CA, 1991. USENIX Association.

[7] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Conference Proceedings, 1988 SIGMOD Conference*, pages 109–116. Association of Computing Machinery (ACM), 1988.

[8] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and R. Lyon. Design and Implementation of the Sun Network Filesystem. In *Conference Proceedings, 1985 Summer USENIX Technical Conference*, pages 119–130, Berkeley, CA, 1985. USENIX Association.

[9] M. Schulze. Considerations in the Design of a RAID Prototype. In *Master's Report*, Berkeley, CA, 1988. University of California at Berkeley.

[10] Thinking Machines Corporation, Cambridge, Massachusetts 02142-1264. *The Connection Machine CM-5 Techincal Summary*, 1991.

# 10 Author Information

**Sue LoVerso** is a senior software engineer at Thinking Machines Corporation working on the CM-5 Operating System. Prior to joining Thinking Machines, she was employed at Encore Computer Corporation for several years as a member of the Mach Operating System group. She received her Master's degree in Computer Science from the State University of New York at Buffalo. Sue is a member of the IEEE Computer Society and the Society of Women Engineers. She can be contacted at *sue@think.com*.

**Marshall Isman** has been with Thinking Machines Corporation since 1986 and is a manager of Operating Systems. He has been involved in the architecture, design and implementation of the CM2 and CM5 I/O systems and Operating Systems. Prior to joining Thinking Machines Corporation, Marshall worked at Computer Consoles and Bell Telephone Laboratories. Marshall holds a BS from State University of New York at Albany and an MS from Harvard University in Computer Science. He can be contacted at *marshall@think.com*.

**Andy Nanopoulos** is a senior software engineer at Thinking Machines Corporation and was responsible for the design and implementation of the DSN software on the SDA. He has a BS in Computer Engineering from Boston University. Before coming to TMC he was president of Yorfrenz Development Corporation. He can be contacted at *andyn@think.com*.

**Bill Nesheim** holds a BS from Cornell University. He has been with Thinking Machines since 1986, where he is currently manager of I/O system software. Prior to joining Thinking Machines he was a research programmer at the Cornell University Computer Science department. He can be contacted at *nesheim@think.com*.

**Ewan Milne** is an OS Engineer working on CMOST development. He is currently working on the SDA project. Prior to joining Thinking Machines in October of 1992, he worked for 8 years at Prime Computer Inc., where he was a Principal Engineer in the OS group. Ewan studied Computer Science at Rensselaer Polytechnic Institute and is currently completing his Bachelor's Degree at Boston University. He can be contacted at *milne@think.com*.

**Rick Wheeler** has a BA from Brandeis University and a MSc from Hebrew University. Before joining Thinking Machines in 1990, he was a member of the staff at the Israel Air Force Institute and Hebrew University's Computer Science Department. He can be contacted at *ric@think.com*.