Tiger Shark - a scalable file system for multimedia

Roger L. Haskin

IBM Almaden Research Center
650 Harry Road
San Jose, CA. 95120

Keywords: Video server, MPEG, interactive television (ITV), video on demand (VOD), file system, cluster, parallel, real-time, fault-tolerant

Abstract

Tiger Shark is a scalable, parallel file system designed to support interactive multimedia applications, particularly large-scale ones such as interactive television (ITV). Tiger Shark runs under the IBM AIX® operating system, on machines ranging from RS/6000™ desktop workstations to the SP2® parallel supercomputer. In addition to supporting continuous-time data, Tiger Shark provides scalability, high availability, and on-line system management, all of which are crucial in large-scale video servers. These latter features also enable Tiger Shark to support non-multimedia applications such as scientific computing, data mining, digital library, and scalable network file servers. Tiger Shark has been employed in a number of customer ITV trials. Based on experience obtained from these trials, Tiger Shark has recently been released in several IBM video server products. This paper describes the architecture and implementation of Tiger Shark, discusses the experience gained from trials, and compares Tiger Shark to other scalable video servers.

Trademarks: AIX, RS/6000, Microchannel, and SP2 are trademarks of IBM Corp. NFS is a trademark of Sun Microsystems, Inc. UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd. XFS is a trademark of Silicon Graphics, Inc.

DRAFT of a paper published in the IBM Journal of Research and Development, Volume 42, Number 2, March 1998, pp. 185-197.

Copyright 1998 by International Business Machines Corporation. Copying in printed form for private use is permitted without payment of royalty provided that (1) each reproduction is done without alteration and (2) the *Journal* reference and IBM copyright notice are included on the first page. The title and abstract, but no other portions of this paper, may be copied or distributed royalty free without further permission by computer-based and other information-service systems. Permission to *republish* any other portion of this paper must be obtained from the Editor of the Journal.

Introduction

To date, most multimedia applications run on stand-alone personal computers, with digitized video and audio coming from local hard disks and CD-ROMs. Increasingly, there has been a demand for file servers that support multimedia data. Some of the reasons for this are the same ones that motivate the use of file servers for conventional data: sharing, security, and centralized administration.

It is difficult for a conventional file server to handle multimedia data. When a conventional file server becomes overloaded, all users see lower throughput and higher response time. The file server must deliver digitized video or audio data at a rate that allows it to be presented to the user in a smooth, continuous stream (this is called *continuous-time* presentation). Any nontrivial delay by the server results in *stream starvation*, which appears to the user as an annoying interruption in the presentation. Stream starvation can be avoided in an ad-hoc manner by buffering data and/or under-loading the file server, but either of these alternatives can increase cost prohibitively. A video server differs from a conventional file server by incorporating an *admission control* mechanism to prevent overloading, and a *scheduling* mechanism to ensure that data is supplied in a continuous manner.

In large-scale multimedia applications like video-on-demand (VOD), interactive television (ITV)[1], and Worldwide Web, the difficulty of providing continuous-time access is exacerbated by the sheer magnitude of the systems. A single video stream requires between 1.5 and 6 Mb/s of bandwidth. At the 6 Mb/s typically required for ITV, even the 100-stream servers that have been deployed in a number of ITV trials must support a throughput of 75 MB/s. Compare this to most conventional network file servers, which are limited to well under 10 MB/sec. A 1000-stream server, which is at the entry level of what has been considered for production ITV systems, requires at least a 38 node SP2[2], with all nodes accessing the same video data simultaneously, making the need for scalability obvious.

The need for high availability and manageability in a large-scale VOD or ITV server is obvious as well. Failure of a 1000-stream ITV system presenting 2-hour movies at $5 each costs $2500/hour. Failure of a digital-video broadcast (DVB) server can put a cable TV head-end, broadcast station, or a broadcast network off the air. Ordinary component failures must not take down the server or even unduly interrupt viewers, and it must be possible to repair and reconfigure the server while it remains operational.

The remainder of this paper will discuss the architecture of the Tiger Shark file system, with emphasis on the features that allow it to function as a video server, will describe its use in a variety of IBM ITV trials and video server products, and will conclude by discussing the lessons learned to date and possible future directions.

The Tiger Shark file system

Broadly speaking, a video server consists of three components: a *control* component that responds to client requests, a *communication* component that moves data through the network from the server to the client, and a *file system* component that manages the storage and retrieval of data from disk. To enable the use of the RS/6000 and SP2 computers as video servers, we developed the Tiger Shark file system, which incorporates a number of features that permit its use in a video server:

- Tiger Shark is designed to handle continuous-time data by making use of the real-time features of the AIX operating system and by scheduling disk I/O to assure that data is read and written on time.
- Tiger Shark supports a high degree of scalability, both in the amount of data it can store and the data throughput (bandwidth) it supports.
- Tiger Shark is designed for high availability.

[1] Video-on-demand includes relatively non-interactive applications such as playback of movies. Interactive television includes applications such as home shopping, education, and training, in which the user interacts with the system.

[2] The Microchannel on each SP2 node, which has a hardware throughput limit of 40 MB/s, determines the number of nodes.

When properly configured, it remains operational in the face of any single disk or node failure.

- Tiger Shark is designed to simplify or eliminate routine system-management tasks. Automatic load balancing across disks is inherent in the design. All operator-initiated management functions can be performed while the system remains operational.

In addition to supporting video, Tiger Shark contains a number of features that make it suitable as a general-purpose parallel file system:

- Tiger Shark presents a Posix-compliant programming interface, so application programs can use it with few, if any, modifications.

- Tiger Shark provides high-speed access to a file from a single application or from any number of applications running in parallel.

- Tiger Shark is fully cache-coherent across nodes in the SP2. Cache coherence is implemented using a byte-range locking mechanism that allows parallel access to non-overlapping regions of a file with little or no communication overhead.

In summary, Tiger Shark enables the RS/6000 and the SP2 to efficiently satisfy the data access demands of both multimedia and parallel computing.

Tiger Shark overview

Tiger Shark runs on a cluster of processors (*file-system nodes*) that share a pool of disks. File-system nodes can access the disks directly over a switching network, or via other processors called *storage nodes* to which the disks are physically attached. A single node can serve as both a file-system node and a storage node, but for simplicity, the discussion will treat the two types of nodes as if they were distinct. Each file-system node can read from and write to all disks. A single RS/6000 processor can be considered as a single-node cluster. In the SP2, Tiger Shark file-system nodes use a software component called VSD (Virtual Shared Disk, [1]) to ship disk block read and write requests to storage nodes over the high-speed switch. This is illustrated in Figure 1.

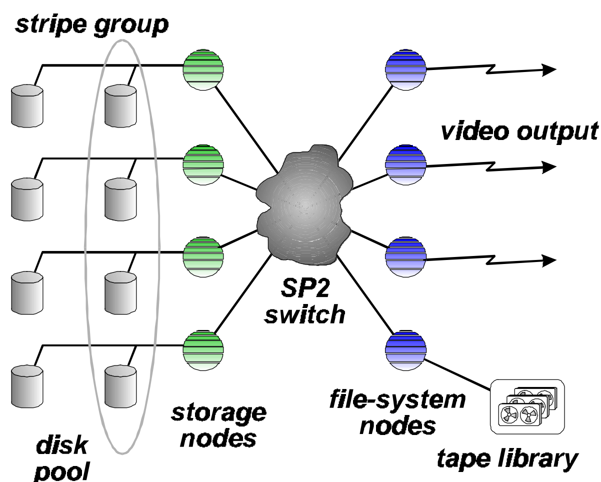

Figure 1. Tiger Shark Architecture

Tiger Shark supports multiple, separately mountable file systems.[3] In Tiger Shark, each mountable file system is striped across a collection of disks called a *stripe group* and can be accessed in parallel from all file-system nodes. Tiger Shark presents a single-system image to its clients — programs on separate file-system nodes see a globally consistent view of each mounted file system.

From Tiger Shark's point of view, video server components that stream video to external users, such as an ATM stream driver or the NFS® (Network File System) dæmon, are simply application programs running on file-system nodes. Other programs can run simultaneously and share data with video server components. For example, a tape archive manager running on one file-system node can begin retrieving a movie from a tape library and, slightly later, an ATM stream driver on another node can start streaming the movie to a viewer.

Tiger Shark makes files available through the AIX VFS (virtual file system, [2]) interface, which makes Tiger Shark compatible with the AIX native file system. Programs don't have to be modified to use Tiger Shark unless they make use of the functions that control its multimedia features.

Tiger Shark architecture

Tiger Shark has a number of architectural elements that allow it to meet its design goals. Following is a brief discussion of the major ones:

[3] The operating system literature uses the term "file system" to refer both to the software that manages file data and to the structure of this data on disk. This paper attempts to make the context sufficient to disambiguate these two meanings.

Continuous-time file access. Tiger Shark uses real-time features of the AIX operating system ([3]) to prevent the kernel and other programs from interfering with continuous-time data delivery. It also implements real-time disk scheduling to execute disk I/O operations in the proper order for achieving an uninterrupted flow of data to clients. Tiger Shark uses deadline scheduling (as opposed to conventional scan or elevator algorithms) to implement both recording and playback of files with arbitrary video rates.

Large disk blocks. To efficiently support multimedia and supercomputing, the file system must maximize throughput from the available disks. Since disk throughput is strongly related to disk block size, Tiger Shark uses a large disk block size, 256 KB being the default. Conventional file systems optimize for space rather than throughput by using small disk blocks; for example, the AIX native file system uses 4 KB blocks. Tiger Shark can store several small files (or partial blocks at the end of large files) together in a single large block, so its disk space utilization is comparable to that of a conventional, small-block file system.

Wide striping. In both supercomputing and multimedia, much or all file system activity is often directed at a single file. For example, many or all clients of an ITV system may be viewing the same "blockbuster" movie simultaneously the first night it becomes available. For the 1000-stream server mentioned in a previous example, the entire 750 MB/s server throughput would be directed at the single file containing this movie. This is higher than the throughput of an individual disk (5-10 MB/s) or even of an individual storage node in the SP2 (40 MB/sec). Achieving higher throughput than this from a single file requires *striping* it across multiple disks and storage nodes, so that successive blocks of the file go on different disks and nodes. Each Tiger Shark file system can be striped across as many as 2^{16} disks. Since each file is striped evenly across all disks in a stripe group, disk load is inherently balanced, regardless of file access skew.

Fault-tolerance. As the number of disks and nodes in the system increases, so does the probability of component failures. If hardware RAID (for example, see [4]) is available, Tiger Shark can use it as a means of protecting disk data. However, hardware RAID subsystems can be more than twice as expensive per byte as conventional disks, RAID controllers are often performance bottlenecks, and, for large systems, the probability of an entire RAID subsystem failing is significant. As an alternative to RAID, Tiger Shark supports block-level replication of both file data and metadata (the bookkeeping information that keeps track of the location of files on disk). Tiger Shark also recovers from file-system node failures, so if a node crashes in the process of modifying the file system (e.g., while recording or while loading content from tape), the file system will be recovered to a consistent state.

On-line system management. Tiger Shark responds to signals generated by the AIX operating system to automatically recover and reconfigure in response to the failure or repair of hardware components. Reconfiguration occurs while the system remains operational. System administration commands (creating and deleting file systems, adding and removing disks from file systems, etc.) are also executed while the system remains operational. Files can be reorganized (e.g., restriped onto newly added disks) on-line, with the reorganization taking place in the background.

File system metadata

Tiger Shark file system metadata (on-disk data structures) roughly correspond to those of a conventional UNIX® file system ([5]), but have been substantially redesigned to work well in a cluster.

The root data structure of a file system is the stripe group descriptor, analogous to the UNIX superblock. It contains overall file-system attributes, pointers to other data structures (for example the inode and allocation map files described below) and information about each disk in the stripe group. The stripe group descriptor is replicated on each disk of the stripe group, and is read and written using a quorum algorithm, which makes the operations of reading and updating the stripe group descriptor tolerant of disk and node crashes. Because reading and writing the stripe group descriptor is done infrequently, the overhead of the quorum algorithm is not significant.

Each disk in a stripe group can be a data disk (which holds file data), a metadata disk (which holds metadata), or both. This allows the use of disks with different physical characteristics for data and metadata.

File system blocks in Tiger Shark (both data and metadata) can be replicated. Each disk address contained in file system metadata is an array, with one element per replica. The disk information in the stripe group descriptor contains system topology information that allows Tiger Shark to allocate block replicas on disks with no common failure point. The degree of replication can be controlled on a per-object basis. A stripe group can have replicated metadata but unreplicated files, or can have different levels of replication for different files; for example, only popular movies might be replicated. This allows making tradeoffs between fault-tolerance and system cost.

Each data file is striped across all data disks in the stripe group. Metadata is similarly striped across all metadata disks. Tiger Shark supports two striping policies: round robin and balanced random. For round robin striping, a single striping order is used for all files. All replicas of the first block of a file are written to randomly chosen disks, subject to the constraint that they have no common failure point. As shown in Table 1, corresponding replicas of successive blocks are written to data disks chosen according to the striping order. For balanced random striping (Table 2), a separate, randomly generated striping order is used for each k blocks of the file, where k is the number of data disks. Again, replicas of a single block are offset by a random amount within this order. Round robin gives higher throughput without missed deadlines, but balanced random files can be restriped (i.e., when disks are added to the stripe group) by moving only 1/k of their blocks.

Because of the random offset in the striping order between replicas of blocks, the replicas of the blocks on a given disk will be more or less evenly distributed across the other disks in the stripe group. After the failure of one disk, the load on the other k-1 disks in the stripe group increases by only 1/k. Contrast this to mirroring, in which the entire load of a failed disk must be borne by its mirror. Thus, Tiger Shark can operate its disks at (k-1)/k of their maximum throughput and still not have them overload after a failure, as opposed to the limit of 1/2 that would be the case with mirroring.

Tiger Shark can use logical disks (for example, AIX Logical Volumes) interchangeably with physical disks. Most hardware RAID subsystems make a parity group, or *rank,* of physical disks appear to the operating system as a single logical disk. In such a configuration, the RAID subsystem stripes individual file blocks across the physical disks in a parity group, and Tiger Shark stripes *successive* file blocks across successive RAID parity groups according to the striping permutation. Tiger Shark optimizes

Disk Number

	Block 0	Block 1	Block 2	Block 3
Replica 1	3	4	1	2
Replica 2	4	1	2	3
Replica 3	1	2	3	4

Table 1. Example of file-system block replication with round-robin striping. The entry is the number of the disk on which the block is written. Striping order is 1-2-3-4.

Disk Number

	Permutation 1				Permutation 2			
	Block 0	Block 1	Block 2	Block 3	Block 4	Block 5	Block 6	Block 7
Replica 1	3	1	4	2	1	3	4	2
Replica 2	4	2	3	1	2	1	3	4
Replica 3	1	4	2	3	3	4	2	1

Table 2. File-system block replication with balanced-random striping. The entry is the disk on which the block is written. The striping orders are 3-1-4-2 and 1-3-4-2

performance for specific RAID subsystems by proper choice of the size and alignment of data blocks. For example, the IBM RAIDiant ([4]), performs best for 256 KB reads and writes aligned on 256 KB boundaries, which it executes as 64 KB operations in parallel on each disk in the rank.

As in UNIX, each file consists of an *inode*, some number of data blocks, and possibly one or more *indirect blocks*. The inode is a fixed-size data structure that contains the file size, creation and modification times, access control information, and pointers to the file's data blocks. If the inode is too small to contain pointers to all data blocks, the inode instead contains pointers to *indirect blocks,* which in turn point to the data blocks.

The inodes of all files in a stripe group are stored in a special file called the inode file. Each inode is referred to by its *inode number*, which is its ordinal position in the inode file. The stripe group descriptor contains a pointer to the first block of the inode file; the first inode in the file (inode 0) is that of the inode file.

Another special file, the allocation map file, contains the allocated/unallocated state of disk space in the stripe group. File system blocks are subdivided into 32 *fragments* (fractional blocks); the allocation map contains the state of each fragment. Fragments are used to efficiently store multiple metadata blocks, small files, or partial blocks at the end of large files, together in a single file system block. The allocation map is divided into *n* segments, each of which describes *1/n* of the space on each disk in the stripe group. The segmented allocation map allows *n* separate nodes to allocate disk space simultaneously without contention. The number of segments is specified when the file system is created according to the number of nodes expected to be writing at the same time.

Each stripe group also contains a number of log files. Each node has its own log file in which it maintains its recovery log. Recovery logging is described in detail below.

Directories, which associate human-readable file names with inode numbers, are also stored in files. Directories are structured as extensible hash tables rather than as the more familiar linear list or B-tree structures. Extensible hashing [6] allows effectively uniform lookup time regardless of the number of files in a directory. Each hash bucket is stored in an 8 KB block in the directory file. When a hash bucket overflows, the hash table is extended and the full bucket is split into two partially full buckets stored in separate 8 KB blocks.

Software structure

Figure 2 illustrates the Tiger Shark software structure. Tiger Shark is implemented as a multithreaded dæmon process (labeled "Tiger Shark Dæmon" in the figure) and a dynamically loaded kernel extension. The kernel extension implements the Virtual File System (VFS) functions. The application program calls Tiger Shark through the standard file-system interface (open, close, read, write, etc.). The AIX kernel routes each call to the corresponding VFS function in the kernel extension.

Tiger Shark implements two communication mechanisms: a highly efficient mailbox mechanism for communication between the kernel extension and the dæmon, and a set of remote procedure call (RPC) interfaces for the dæmons to communicate among themselves.

Tiger Shark uses memory shared between the dæmon and kernel extension to store its internal data structures, buffered file data, and buffered metadata. The kernel can obtain locks on objects in shared memory, and can examine or modify them. Simple VFS operations (such as reads and writes of buffered data or directory lookups in cached directory files) are performed in the kernel extension. The kernel extension sends mailbox messages to the dæmon to perform more complicated operations, particularly those that require I/O. For example, if a program issues sequential 4 KB reads, the kernel extension

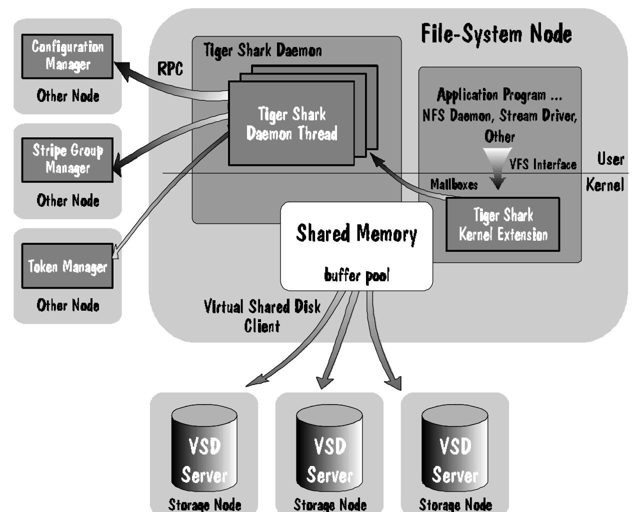

Figure 2. Tiger Shark Software Architecture

sends a message to the dæmon instructing it to read each 256 KB data block (i.e., one message for every 64 read calls), and satisfies the other reads itself from buffered data in shared memory. For real-time playback, Tiger Shark attempts to prefetch data so that a program reading at the playback rate will never be blocked on a read call.

The RPC interfaces are implemented by means of TCP sockets. When the Tiger Shark dæmon receives an RPC, a dæmon thread is dispatched to perform the requested operation. Two of these RPC interfaces, the c*onfiguration manager* and s*tripe group manager*, require discussion.

One file-system node is chosen configuration manager when the system starts up, and thereafter whenever the previous configuration manager terminates. The configuration manager keeps track of what stripe groups are mounted, and chooses the stripe group manager for each active stripe group.

When a file-system node mounts a file system, it sends an RPC to the configuration manager to request the identity of the stripe group manager for the associated stripe group. If no stripe group manager exists, the configuration manager appoints the requesting node as stripe group manager. If a stripe group manager fails, the configuration manager chooses another node as its successor. The stripe group manager maintains the list of nodes that have the stripe group (or more accurately the file system contained in the stripe group) mounted, processes mount requests, and allocates log files to nodes. System management commands that affect the stripe group (e.g., adding or removing a disk, restriping) can be initiated on any node but are performed by the stripe group manager. The stripe group manager also manages other global state operations, e.g., changes to user disk space quotas, assignment of allocation segments to nodes, etc.

Tiger Shark is a general-purpose file system that allows any number of nodes to read and write files simultaneously while giving all nodes a coherent view of the file system. Since file-system nodes buffer (or cache) data to improve performance, nodes must be informed when other nodes perform operations that affect their cached data. Tiger Shark uses a modified version of the token manager in the Calypso file system ([**7**]) for distributed locking. Buffer pool coherency locking is functionally distinct from program controlled (*fcntl*) locking, although the latter is also implemented via the token manager.

In common with many modern file systems, Tiger Shark implements recovery by means of *journalling*, or write-ahead logging ([8]). Operations that modify metadata (with the exceptions noted below) are logged, and the log records are written to disk before the modified metadata is written. Within a stripe group, each node has a separate log file, which is replicated. After a node failure, the stripe group manager reads the failed node's log file, applies the changes described in its log records, and then instructs the token manager to release any tokens that might have been held by the failed node. This recovery processing takes on the order of seconds. In the interest of performance, updates to inodes and indirect blocks are not logged. Instead, Tiger Shark observes the following order of operations when writing to a file:

- Data blocks and log records describing allocations of new data and/or indirect blocks are written.
- Indirect blocks are written.
- The inode itself is written.

This ordering prevents file metadata from pointing to unallocated or uninitialized data blocks when writing to a file. A similar ordering is required when truncating or deleting a file:

- A list of all blocks to be deallocated is built in memory.
- The modified inode and/or indirect blocks that used to refer to the deallocated disk block are written.
- The blocks in the aforementioned list are deallocated.

This protocol ensures that a disk block is never deallocated as long as a reference to it exists on disk. When a file is deleted, its inode is updated to show that it has been deleted before the inode and indirect blocks are traversed to build the deferred deallocation list.

The write ordering described above maintains metadata consistency after file-system node failures with less I/O than journalling, but a node crash can result in a limited number of unused blocks being marked as allocated in the allocation map. To reclaim this space, an online version of the familiar UNIX *fsck* command was implemented to traverse

file system metadata and reclaim any allocated but unused blocks. This online *fsck* is done in the background while the system is running, and does not interfere with real-time I/O. The more familiar offline *fsck* is also used in rare failure situations such as the loss of a log file or file system corruption due to a software bug. Reloading a multi-terabyte file system from tape takes so long that even these rare failures must be guarded against. The Tiger Shark *fsck* command restores a corrupted file system to a usable state and saves as much file data as possible. In a video server, with large blocks and relatively few files, the time required to execute offline *fsck* is acceptable even for large file systems.

Tiger Shark employs the real-time programming features of the AIX kernel to enable it to implement continuous-time access to multimedia data. When used in a video server, all Tiger Shark code and data buffers are pinned in memory, in order to prevent page faults from interfering with real-time performance. Furthermore, the Tiger Shark dæmon runs at real-time priority. This prevents other processes (including the AIX process scheduler) from interfering with Tiger Shark and disables time slicing for the dæmon. Finally, the AIX real-time clock is used to implement deadline scheduling.

Background activities such as content loading require Tiger Shark to perform non-realtime I/O to the file system while real-time streams are being played and recorded. Tiger Shark allows all disk-system bandwidth not consumed by real-time streams to be used for non-realtime I/O in a manner similar to that described in [9]. During file system operation, Tiger Shark keeps track of the fraction $\boldsymbol{s}$ of this bandwidth actually used by currently playing streams. After executing a non-realtime I/O taking time t, Tiger Shark waits for time $t(1-\boldsymbol{s})/\boldsymbol{s}$ before starting the next non-realtime I/O.

Tiger Shark uses an EDF (earliest deadline first, [10]) policy to schedule real-time I/O. The deadline (completion time) of the ith block in a stream is that of the $i-1$st block plus the ith block's playback time (block size / data rate). Real-time I/O to each disk is executed in EDF order. In single-node configurations, Tiger Shark calls the AIX disk device driver to perform I/O operations. To restrain disk drivers, adapters, and devices from re-ordering I/O to optimize throughput, Tiger Shark limits the number of I/O operations on each disk at any time to at most two, one executing and the other queued. When the executing operation completes and the queued operation begins executing, Tiger Shark queues the subsequent operation to the disk.

As was mentioned above, Tiger Shark uses VSD (Virtual Shared Disk) to perform I/O to shared disks on storage nodes. VSD was modified to allow a deadline in each disk I/O request. Tiger Shark computes the deadline for each real-time I/O request, and VSD executes requests to each disk in global deadline order. VSD contains logic similar to that described above to prevent AIX and the hardware from re-ordering I/O operations.

Global deadline scheduling requires file-system node clocks to be synchronized within approximately the duration of an I/O operation. Tiger Shark generates deadlines using the SP2 switch clock, which is synchronized on all nodes to within a microsecond. Since the duration of a 256 KB disk I/O is approximately 50 ms., if the switch clock were not available, the CPU time-of-day clock could be used in conjunction with a standard clock-synchronization algorithm ([11], [12]) to generate sufficiently precise deadlines.

The Tiger Shark kernel extension implements a function (*tsfattr*) that allows programs to control the continuous-time reading and writing of an open file. Programs call *tsfattr* to reserve bandwidth for reading or writing the file, specify the read/write data rate, and set the prefetch depth (the number of read-ahead buffers).

The kernel extension also implements a *zero-copy* interface that allows stream drivers to directly access the pinned Tiger Shark data buffers. The standard UNIX read and write functions copy data between file system buffers and program buffers. This adds substantially to the CPU overhead required to stream video from the server. The zero-copy functions, called from kernel code (including interrupt handlers), allow the inner loop of a stream driver to be implemented very efficiently. Because the zero-copy functions *must* be called from the kernel, file-system data buffers are protected against access by unprivileged programs. The zero-copy functions include *prefetch*, which starts real-time prefetching of data blocks, *getbuf*, which locks a prefetched buffer so its data can be accessed, and *putbuf*, which returns a locked buffer to Tiger Shark after its data has been consumed. A simple zero-copy stream driver opens the file and calls *tsfattr* to reserve bandwidth and set the data rate and prefetch depth. It next calls *prefetch* to read the number of prefetch buffers specified by *tsfattr*, calls *getbuf* to lock the buffer containing the first data block, and starts

sending data to the output device (e.g., an ATM adapter). Each time the device interrupts subsequently, the interrupt handler checks whether the data buffer is empty. If so, it calls *putbuf* to return the empty buffer to Tiger Shark and calls *getbuf* in non-blocking mode to lock the next prefetched buffer. Normally (i.e., if deadlines are met), this buffer will have been read and *getbuf* will return successfully with the lock. If by some chance the buffer has not yet been read, *getbuf* returns an error to the interrupt handler. In this case, the interrupt handler signals a recovery process in the stream driver (e.g., via an event), which in turn calls *getbuf* in blocking mode to wait for the data and resumes sending data to the device. The use of non-blocking *getbuf* and the recovery process bounds the interrupt latency (as required by AIX) and, not incidentally, allows the Tiger Shark dæmon to complete reading the data block. Zero-copy not only avoids the data copy; it also avoids the process switch into the stream driver and the pin/unpin of the device output buffer. Depending on the output device, this can save up to 40% of total instruction path-length.

The standard AIX NFS dæmon can use Tiger Shark to allow NFS clients to play audio and video over a LAN. Because it was considered impractical to modify NFS, Tiger Shark was modified slightly to provide continuous-time playback through NFS. Tiger Shark provides a command to permanently assign a default playback rate to a file. This rate can be directly specified by a user, or can be generated by an analysis tool that examines the video/audio content of the file to determine its playback rate. When an NFS client reads a file sequentially, and the file has a default playback rate, Tiger Shark assumes that the client is performing continuous-time playback. It then reserves bandwidth for the client and uses deadline scheduling to read the file at the default rate.

Experience and Applications

Video on demand and ITV trials

Tiger Shark has been used in a number of video on demand and interactive television trials. The first was the Bell Atlantic Field Trial ([13], [14]), one of the earliest video on demand trials. This trial featured Shark ([15]), a predecessor of Tiger Shark, to provide 50 simultaneous video streams from a single RS/6000 Model 970. Video was sent over standard telephone lines at 1.544 Mb/s (T1 rate) to the customer's set-top decoder. Customers ordered movies over the telephone via an automated menu system, which sent requests to play movies to the video server. This trial extended from April 1993 through September 1994.

A later trial at Hong Kong Telecom ([16], [17]), that provided 150 simultaneous 1.5 Mb/s streams, ran successfully from March through September 1995. The server throughput was too low to justify the use of an SP2, so the server consisted of two independent RS/6000 Model 980 processors, each with its own IBM 7135 RAIDiant disk subsystems. A single-node version of Tiger Shark was used on each RS/6000 to provide real-time access to its local data. Video content was replicated on each server. As in the Bell Atlantic trial, video was sent over telephone lines at 1.544 Mb/s. However, the Hong Kong Telecom trial was truly interactive. The set-top decoder contained an X.25 port for exchanging control commands with the server. The user interface included menus with image overlays and video fly-ins. Other interactive features, such as direct seek to a position in the video specified by a slider bar, were also provided. Tiger Shark itself was able to start streams in under a second in this system, although other delays in the set-top, X.25, and application program contributed to a response time as seen by the user of over a second. In the initial architecture for the Hong Kong Telecom server, the disk storage on each RS/6000 was divided between permanent file-storage space and a temporary file cache. Each video file had a permanent home on one of the RS/6000s; when the file was needed on the other RS/6000, it was copied over an FDDI ring to that machine's cache. In practice, the CPU overhead of copying files over the FDDI ring was so high that even a limited number of cache misses overloaded the server. It was eventually decided to store the entire 200 hours of video content on each RS/6000.

The Tokyo Metropolitan Government ITV trial ([18]), which began in May 1996, is the first to use Tiger Shark running on an SP2. This server stores 200 hours of six Mb/s video and supports 100 streams, with video delivery via hybrid fiber/coax (HFC). The SP2 is configured with five storage nodes and seven file-system nodes. Each storage node controls 32 4.5GB IBM 7133 (Serial Storage Adapter) disks, a total of 720 GB of video data. Each file-system node contains four MPEG multiplexor cards, each of which combines four six Mb/s MPEG-2 streams into a single 20 Mb/s MPEG transport

stream. The transport stream is modulated onto an analog TV channel and distributed via HFC. The number of nodes is determined by the number of Microchannel slots required for disk adapters and MPEG multiplexors. The 720 GB of disk storage is more than adequate to store 200 hours of 6 Mb/s video but not adequate to fully replicate it. To provide some measure of protection against disk failures, 67 hours of the most popular video is replicated on one 80-disk stripe group (half the disks). The remaining video is stored on a number of smaller, unreplicated stripe groups. If a disk failure occurs in an unreplicated stripe group, all files in it must be reloaded from tape. The size of the smaller stripe groups is chosen to be large enough to provide sufficient throughput to handle its expected content while being small enough to reload in an acceptable period. The decision to use unreplicated data was dictated by cost constraints and, as one might expect, was done against the advice of the developers!

Argonne Labs, one of the U. S. national supercomputing centers, collaborated with IBM to develop an experimental WAN-based video server that employs Tiger Shark on a 28-node SP2. This system uses an implementation of RTP (Real-Time Protocol) to transmit video over the M-Bone (multicast backbone) network interconnecting the supercomputing centers. The system was demonstrated at the Supercomputing '95 conference in San Diego (Dec. 3-6, 1995) with video sent to an IBM location in New York.

IBM products using Tiger Shark

Positive results from the aforementioned customer trials have led to Tiger Shark being introduced in three IBM video server products. At present, these products feature Tiger Shark in a single-node configuration to provide real-time delivery, wide striping, and high throughput. The fact that these products are not yet offered on the SP2 reflects the current demands of the marketplace. As customer needs grow to require an SP2, the technology is available to quickly fill this demand.

IBM Multimedia Server for AIX uses Tiger Shark to stream video to personal computers and workstations over video-capable LANs (e.g., ATM, switched Ethernet, or FDDI) using the standard NFS protocol. Multimedia Server supports up to 75 Mb/s (60 streams) from a single RS/6000 server. Tiger Shark uses the streaming heuristic described above to support real-time playback through the stateless NFS protocol. The LAN used to transport video must have sufficiently high throughput, low latency, and low error rate to transport video smoothly. Although conventional LANs have no means to reserve bandwidth or schedule delivery, new LAN technologies such as ATM and switched Ethernet can transport video acceptably to a reasonably large number of clients. In practice, more video "glitches" result from background activity of the client operating system (e.g., paging) than from network delay or errors.

IBM Videocharger Server provides video playback to Internet clients through a Web browser. It supports low bit rate (28.8 kb/s) video over wide-area networks, and high bit rate (e.g., MPEG) video over campus or corporate Intranets. It uses Tiger Shark to store and play back video, and uses the RTP protocol to transmit video over the network. For low bit-rate video, a file block size of 32KB is used, since devoting 256 KB buffers to each low-speed stream wastes RAM.

IBM Media Streamer supports video capture and playback from locally attached video boards (SCSI-attached MPEG to NTSC decoders) and over dedicated-bandwidth ATM connections. It can be used in small to mid-size video on demand applications (hotels, cable TV head ends, digital video broadcast, etc.). It can also replace digital or analog videotape recorders in applications such as ad insertion at a broadcast station, video editing, etc. Media Streamer provides up to 120 Mb/s on an RS/6000 server.

Discussion

Scalability

The Tiger Shark system used in the TMG trial has been tested with 112 6 Mb/s streams, limited by the number of available MPEG multiplexor adapters. This size is modest in comparison to that anticipated for a large ITV system. However, even the TMG server represents several hundred thousand dollars of hardware, which illustrates why scalability must be analyzed by modeling rather than empirically.

Extensive modeling results of the Tiger Shark architecture are presented in [19]. This study modeled the performance of a server with up to 500 4 Mb/s streams. It studied the effects of a number of configuration parameters, including switch

bandwidth, disk block size, read-ahead buffer size, I/O scheduling policy, and number of nodes. To summarize the pertinent results of this modeling as they apply to Tiger Shark:

The model was used to predict the probability of stream starvation due to variations in queuing delay at the switch ports. The model indicates that the stream capacity of the server is proportional to the number of nodes until the number of ports supported by the switch is reached. The highest switch port bandwidth modeled in [19] was 60 MB/s, at which stream starvation rate was .02/hr/stream for the "two tier" architecture used in the TMG trial (separate file-system and storage nodes). The SP2 used in the TMG trial differs from the model in that it has a switch port bandwidth of 80 MB/s (vs. 60) and a CPU utilization of 37% (vs. 70% in the model). Both of these differences would indicate the stream starvation rate in the actual Tiger Shark system would be lower than the .02/hr/stream rate of the model. The model assumed a nonblocking switch, whereas the SP2 switch can block at high loads. Numerous analyses of switch blockage as a function of load have been published (see, for example, [20]), showing that for a port utilization of below 50%, the probability of packet loss due to blocking is negligible. In the TMG system, the switch port utilization is 20% in the storage nodes and 15% in the file-system nodes.

The model evaluated optimum block size as a function of disk bandwidth utilization. The model in [19] indicates that for up to 90% disk bandwidth utilization, the 256 KB block size used in the TMG configuration is within 5% of optimal. The TMG system had 140 disks, each capable of reading 256 KB blocks at 6 MB/s. The modeling results indicate that this number of disks could provide 784 6 Mb/s streams at 70% utilization with a starvation rate of .01/stream/hr. For this result, the model assumed first-come first-served scheduling and random striping of data blocks across disks. The TMG system used EDF scheduling and round robin striping, both of which would result in an even lower stream loss rate than that of the model.

The model also evaluated stream starvation as a function of read-ahead buffer size. In the TMG system, each stream has two full-block read-ahead buffers in Tiger Shark, a buffer in the HFC card device driver, and a buffer on the card itself. The model shows that for 80% disk utilization, the starvation probability for a system with four buffers is negligible (0.0).

The model shows the possibility of stream starvation for video clips too short to be striped across all the disks. To maintain a constant starvation rate of .01/hr/stream when playing short files, the model indicates that the stream capacity for each node must drop from 20 4 Mb/s streams for a single-node server to 18 for a 128-node server. Tiger Shark avoids even this minimal degradation by allowing short files to be replicated to whatever degree is necessary to stripe their blocks across all disks in the stripe group.

Disk Scheduling

The modeling results of [19] indicate that disk utilization over 80% can be achieved with essentially no stream starvation using EDF scheduling. A number of algorithms have been proposed ([21], [22], [23]) that provide higher disk throughput by grouping read requests for a number of streams, and scheduling each group to maximize throughput using a conventional scan algorithm. Modeling results presented in [24] indicate that for the large block sizes used by Tiger Shark (256KB and up), the throughput increase is negligible. For example, Table 3 shows the effect of grouping for a typical SCSI-2 disk drive. The minimal throughput gain indicated by the simulation does not warrant the increase in stream startup latency caused by grouping.

Other Scalable Multimedia File Systems

Although a number of scalable video servers have been developed and commercially deployed, relatively little detailed description of them exists in the technical literature, and little of that discusses the file systems. The closest comparable published work is Tiger ([25]), and XFS™ ([26]).

Streams/Group	Throughput (MB/sec)	Latency (ms.)
1	4.03	62.08
2	4.20	119.16
4	4.34	230.32
8	4.44	449.97
16	4.51	887.39

Table 3. Effect of Grouping on Performance for IBM Ultrastar 2ES SCSI-2 disk drive, 4.5 GB capacity, 5600 RPM, 56 KB/track (avg.), seek times 1.1/8.5/15 ms. min/avg./max.

XFS is a high-performance file system for Silicon Graphics systems. Like Tiger Shark, it provides real-time delivery ("guaranteed rate I/O"), wide striping, and a large data-block size. In addition, like Tiger Shark, XFS can replicate data, uses journalling for recovery, allows very large files and file systems, supports sparse files, and has online system management. There are a number of major differences between XFS and Tiger Shark. XFS directories use a B-tree rather than extensible hashing. B-tree lookup performance is O(log n) in the number of files, whereas that of extensible hashing is O(1). XFS replicates data by mirroring logical disks, compared with the Tiger Shark block-level replication. Consequently, when an XFS disk fails, its entire load is taken up by its mirror rather than being spread evenly over the remaining disks, as it is in Tiger Shark. XFS has a "direct I/O" mechanism to read data from disk directly into the user buffer. Like the Tiger Shark zero-copy feature, direct I/O avoids copying data from the file system buffer pool to user space (by eliminating the file-system buffer rather than by giving the user access to it). However, direct I/O prevents the buffer pool from providing any benefit. This can affect the stream startup latency for frequently accessed files (e.g., video fly-ins used in top-level menus). Finally, the ultimate scalability of XFS is limited by servers on which it presently runs, the largest at present being the Challenge XL shared memory multiprocessor with 32 processors and 1.2 GB/sec I/O throughput. The SP2 switch-connected cluster can expand to 512 processors, with an I/O throughput of up to 20 GB/s.

Tiger is a special-purpose file system developed by Microsoft for use in video servers. It consists of a number of nodes (called "Cubs") acting under the direction of a central controller node. Files are striped across the Cubs, which are analogous to the Tiger Shark storage nodes. The Cubs transport video to set-top boxes via an ATM network. To play a video stream, a "multipoint-to-point" ATM switched virtual circuit is established between every Cub and the set-top box (presumably this is done once, when the viewer starts using the server). The stream is assigned a free time slot in a global schedule. Time slots are arranged cyclically, so if block i of a stream comes from disk j during time slot t, block i+1 will come from disk j+1(modulo the number of disks) in slot t+1. Each block is sent to the set-top in the time slot following the one during which it was read. If the clocks in the Cubs are synchronized, there are no collisions along the multipoint-to-point connection. The principal advantage of this architecture is that it avoids the expense of having both storage and front-end file-system nodes (and the additional level of switching to connect the two). However, this advantage also limits Tiger's flexibility. The slotted schedule requires all streams to have the same data rate. Tiger Shark, by comparison, can support arbitrary video rates, which can provide significant cost and space savings for content (e.g., news) that does not require the full six Mb/s data rate used in [25]. The slotted architecture makes it difficult or impossible to expand the system (i.e., add Cubs and/or disks) online. Because adding disks or Cubs would require all data to be restriped, the amount of downtime required for such reorganization in a large server could be prohibitive. Finally, VOD and ITV systems now and for the foreseeable future will have to support ADSL and hybrid fiber/coax for video distribution. Neither ADSL nor HFC support multipoint-to-point. To use Tiger ADSL or HFC would require front-end nodes to receive ATM packets from the Cubs, combine them into streams, and send them to the ADSL or HFC adapters. The resultant configuration of Cubs, ATM switches, and front-end nodes closely corresponds to the storage nodes, switch, and file-system nodes in Tiger Shark, eliminating the purported cost advantage of the decentralized Cub architecture.

Technical and commercial applications

Tiger Shark's attributes of scalability, high throughput, availability, manageability, and a standard programming interface are important in many technical and commercial applications such as simulation, seismic processing, and data mining. Many such applications require high-speed sequential read and write access to large files, which is precisely for what Tiger Shark was designed. This has resulted in considerable interest in Tiger Shark as a general-purpose parallel file system for use in technical computing and scalable servers.

To date, Tiger Shark has been successfully used in the Scalable Web Server prototype developed at IBM Watson Research Lab ([27]). Tiger Shark has also been used in a study with an IBM customer retrieve seismic data for processing. In this study, Tiger Shark offered a 15X speedup over NFS and a 5X speedup over the existing PIOFS parallel file system for SP2 ([28]), which is designed for this type of parallel technical computing.

Tiger Shark is presently being developed as a product for the SP2. As part of this product development effort, several extensions to Tiger Shark are under way to allow it to better support general purpose parallel computing. Two of the most important of these are dynamic prefetch and fine-grained write sharing. Dynamic prefetch senses when an application program is trying to read a file at a high rate and automatically increases the number of prefetch buffers in order to maximize I/O parallelism. Fine-grained write sharing extends the token manager and local lock manager to allow multiple nodes to work on non-intersecting portions of a file with minimal global locking traffic. A large simulation, for example, can involve many nodes simultaneously making updates to small pieces of a file on which they are all working.

Summary

Multimedia applications place demands on the file system beyond the ability to support real-time access. Tiger Shark not only supports real-time access, but also provides a scalable, robust, and manageable system adequate for large ITV systems. Tiger Shark has proven itself in a number of real-world customer trials, which have been sufficiently successful to justify its being used in three IBM video server products. At present, development is under way for a general-purpose parallel computing file system based on Tiger Shark. The early customer feedback on this product is promising.

Acknowledgements

The author acknowledges the contributions of the many people that participated in the development of Tiger Shark. The members of the Tiger Shark project at Almaden Research Center include Jim Wyllie, Frank Schmuck, Daniel McNabb, Michael Roberts, Carol Hartman, Thomas Engelsiepen, and Marc Eshel. Research colleagues at the T. J. Watson Lab responsible for the Calypso token manager and VSD components used by Tiger Shark include Murthy Devarakonda, Daniel Dias, and Rajat Mukherjee. The Video on Demand project team at the IBM Bethesda Laboratory under Jack Bottomley was instrumental in the success of the Bell Atlantic and Hong Kong Telecom trials. Special thanks go to Frank Stein, Leonard Degollado, James Tani, and Hatem Ghafir. The On Demand Systems group in IBM Japan under Y. Satoh included T. Sanuki, Y. Asakawa, and H. Yamasaki, all of whom contributed heavily to the success of the Tokyo Metropolitan Government Trial. Thanks goes to the development group in IBM's Austin Lab under Daniel Bandera, who built the Multimedia Server, Videocharger, and Media Streamer products around Tiger Shark. David Craft, Scott Porter, Eugene Johnson, Brian Dixon, Partha Narayanan, and Damon Permizel deserve special mention for their tireless work. Finally, the work of the parallel file system groups in Poughkeepsie (particularly Lyle Gayne, Robert Curran, Radha Kandadai, and Andrew Zlotek, with support from JefferyLucash) and Haifa (Zvi Yehudai, Boaz Shmueli, Benny Mandler, John Marberg, Sybille Schaller, and particularly Itai Nahshon) must be recognized.

References

1. IBM Corp., *Virtual Shared Disk user's guide*, Order Number GC23-3849-00, IBM Corp, 1994.
2. IBM Corp., AIX Version 4 Kernel Extensions and Device Support Programming Concepts, Order Number SC23-2611, IBM Corp., 1995.
3. IBM Corp., *AIX Version 3.1 RISC System/6000 as a real-time system*, Order Number GG24-3633-00, IBM Corp., 1991.
4. IBM Corp., *A practical guide to the IBM 7135 RAID Array*, Order Number SG24-2565-00, IBM Corp., August 1985.
5. Maurice J. Bach, *The design of the UNIX® operating system,* Prentiss-Hall, Inc., Englewood Cliffs, New Jersey, 1986.
6. Ronald Fagin, Jürg Nievergelt, Nicholas Pippenger, H. Raymond Strong*, Extendible hashing - a fast access method for dynamic files*, ACM Transactions on Database Systems, New York, NY, Volume 4 Number 3, 1979, pages 315-344.
7. Ajay Mohindra and Murthy Devarakonda, *Distributed token management in the Calypso file system*, Proceedings of the IEEE Symposium on Parallel and Distributed Processing, New York, 1994.
8. J. N. Gray, *Notes on database operating systems*, in O*perating systems, an advanced course*, edited by R. Bayer et. al., Springer-Verlag, Berlin, Germany, 1979, pages 393-400.

9. Narasimha Reddy and James C. Wyllie, *I/O issues in a multimedia system,* IEEE Computer, 27(3):69-74, 1994.

10. C. L. Liu and J. W. Leyland, Scheduling algorithms for multiprogramming in a hard real-time environment, Journal of the ACM, 1973, pages 46-61.

11. Joseph Y. Halpern, Barbara Simons, H. Raymond Strong, Danny Dolev*, Fault-tolerant clock synchronization*. Proceedings of the Third Annual ACM Symposium on Princiles of Distributed Computing, Vancouver, B. C., Canada, August , 1984. ACM, ISBN 0-89791-143-1, pages 89-102.

12. Flaviu Cristian, *Probabilistic clock synchronization*. Distributed Computing, Volume 3, Number 3, Springer-Verlag, Berlin, Germany, 1989, pages 146-158.

13. Larry Plumb, Bell Atlantic demonstrates video on demand over existing telephone network, Bell Atlantic press release, Jun. 14, 1993.

14. Bell Atlantic, IBM announce agreement for video on demand server, World News Today, Jan 8, 1993.

15. Roger L. Haskin, *The Shark continuous media file server*, Proceedings of IEEE 1993 Spring COMPCON, San Francisco, CA, Feb. 1993, pages 12-17.

16. Alan Patterson, *Hong Kong Telecom, IBM map video effort*, Electronic Engineering Times, Aug. 1, 1994, page 20.

17. Roger L. Haskin and Frank B. Stein, *A system for the delivery of interactive television programming*, Proceedings of IEEE 1995 Spring COMPCON, San Francisco, CA, Mar. 1995, pages 209-216.

18. Toshiyuki Sanuki and Yasuo Asakawa, Design of a video-server complex for interactive television, this issue.

19. Renu Tewari, Daniel Dias, Rajat Mukherjee and Harrick Vin*, Design and performance tradeoffs in clustered multimedia servers*, Proceedings of the 1996 International Conference on Multimedia Computing and Systems, IEEE Computer Society, Tokyo, June 1996.

20. Achille Pattavina, *Asynchronous time-division switching,* in Communications Handbook, Jerry Gibson, ed., CRC Press, Boca Raton, 1996, pages 686-700.

21. Narasimha Reddy, and James C. Wyllie, *Disk scheduling in a multimedia I/O system*, Proceedings of the 1st International ACM Conference on Multimedia, Anaheim, CA, August 1-6, 1993, ISBN: 0-89791-597-6.

22. P. S. Yu, M. S. Chen, and D. D. Kandlur, *Grouped sweeping scheduling for DASD-based multimedia storage management,* in ACM Multimedia Systems Journal, 1:99-109, 1993.

23. D. James Gemmel, and Jiawei Han, *Multimedia network file servers: multi-channel delay sensitive data retrieval,* in ACM Multimedia Systems Journal, 1:240:252, 1994.

24. Renu Tewari, Dan Dias, and Rajat Mukherjee, *Real-time issues for a clustered multimedia server,* IBM Research Report RC20020, April 1995.

25. William J. Bolosky, Joseph S. Barrera, III, Richard P. Draves, Robert P. Fitzgerald, Garth A. Gibson, Michael B. Jones, Steven P. Levi, Nathan P. Myhrvold, and Richard F. Rashid, *The tiger video fileserver*, Microsoft Technical Report MSR-TR-96-09, presented at the Sixth International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV 96), Zushi, Japan, April 23-26, 1996.

26. Mike Holton and Raj Das, *XFS: a next generation journalled 64-bit filesystem with guaranteed rate I/O,* Silicon Graphics Inc., http://www.sgi.com/Technology/xfs-whitepaper.html, Silicon Graphics, Inc., Mountain View, CA.

27. D. M. Dias, W. Kish, R. Mukherjee, and R. Tewari, *A scalable and highly-available Web server,* Proceedings of IEEE 1996 Spring COMPCON, Santa Clara, CA, February 1996, pages 85-92.

28. Peter F. Corbett and Dror G. Feitelson, *The Vesta parallel file system*, ACM Transactions on Computer Systems, Volume 14, Number 3, New York, NY, August 1996, pages 225-264.