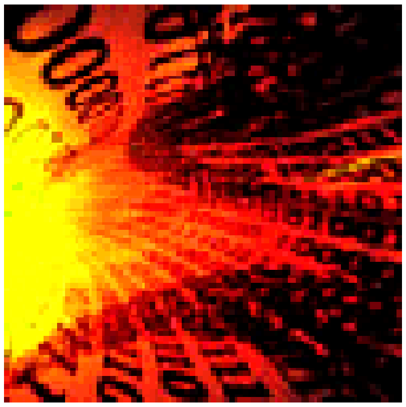
IEEE DISTRIBUTED SYSTEMS ONLINE 1541-4922 © 2004 Published by the IEEE Computer Society
Vol. 5, No. 2; February 2004

Storage System Support for Continuous Media Applications, Part 2: Multiple Disks, Memory, and Integration

Pål Halvorsen, *University of Oslo*
Carsten Griwodz, *University of Oslo*
Vera Goebel, *University of Oslo*
Ketil Lund, *University of Oslo*
Thomas Plagemann , *University of Oslo*
Jonathan Walpole, *Oregon Health & Science University*



To meet multimedia servers' increasing performance and scalability requirements, researchers must design multimedia storage system components that adapt to various workloads and requirements. They must also develop strategies for combining components into a single high-performance system.

Storage systems have long been a major bottleneck in multimedia systems because their performance increases have not kept pace with those of processors and networks. Additionally, new multimedia applications impose a much higher load on such systems. As the number of concurrent users downloading or streaming data from media-on-demand servers increases, so too does the challenge for a multimedia server's storage system.

In the first of this two-part overview of multimedia storage systems, we discussed streaming-system requirements and single-disk issues such as disk scheduling and data placement. Here, we address key issues for multidisk systems and high-layer components, such as buffer management. We've posted a more comprehensive list of references on multimedia storage system issues at www.ifi.uio.no/~paalh/publications/dsonline2004.

MULTIPLE DISKS: ISSUES AND CHALLENGES

Given the high bandwidth requirements of continuous media data playouts, a single disk's bandwidth capacity strongly limits the number of concurrent users in a multimedia scenario. One approach to overcoming the bandwidth bottleneck is to scatter different file segments across multiple disks using data striping or interleaving. Another approach is to use replication to distribute several copies of a file to different disks.

Striping and interleaving

For many years, storage systems have often used *striping*—also called wide or full striping—to read multimedia data from disks. Striping spreads data blocks over all storage system devices. During I/O operations, the system accesses all devices in parallel, increasing the effective transfer rate. RAID (redundant array of independent disks) technology addresses striping performance and reliability issues. Streaming RAID uses a grouping approach, arranging stripes in consecutive segments to achieve good performance.

Recently, however, disk performance has improved considerably (although it's still slow compared to memory and processor performance), and a single disk can now support several continuous media streams concurrently. Traditional striping is thus unnecessary. Developers have proposed several refinements schemes whereby storage systems can serve a request without involving the whole array disk, enabling service of multiple, simultaneous requests.

Data interleaving (also called *compound striping*) stores media files across a set of disks. The simplest version stores successive blocks in a round-robin manner. Other interleaving approaches are *staggered striping*, which stripes data over possibly overlapping disk groups, and *scalable stream pumping*, which stores all data blocks in successive disks and zones.

Two important parameters when implementing a multimedia storage server disk array are the striping unit size and the degree of striping required for optimal resource utilization. Researchers have experimented with a fixed- and variable-sized block placement policy to determine optimal stripe unit size.[1] Another group proposes a scheme to partition the array and stripe data across single-disk partitions to maximize resources.[2] Other work suggests similar policies, because one-disk stripe groups often have better bandwidth utilization.[3] So, as long as using small stripes can service playout rate, it's more efficient with respect to total storage system I/O bandwidth. This is because it requires only one or at most a few disks to pay the overhead of moving the disk head for a particular request, so the system can service concurrent requests in parallel.

Replication

Beyond single-file-retrieval optimization, developers can increase performance of the servers themselves using replication to guarantee availability and increase concurrent access to individual titles. Researchers have proposed several schemes.

Static replication explicitly duplicates content files by storing files on multiple machines and providing various user access points.

Dynamic segment replication dynamically replicates read-only movie segments of equal size according to the prespecified threshold for concurrent read requests. To avoid further I/O overhead, this scheme uses copyback streams and makes replicas when it reads a user's disk blocks.

Threshold-based dynamic replication replicates whole continuous media files and deletes replicas on the basis of workload thresholds. During the replication process, this scheme considers all system disks and the probability of new requests to determine whether to replicate a movie or delete a replica. Researchers have proposed several replication variations here, including injected sequential, piggybacked sequential, injected parallel, piggybacked parallel, and piggybacked and injected parallel.

Partial replication replicates popular file parts to distribute load among different devices and reduce damages if the disk fails. This scheme is based on the observation that if a video receives several consecutive requests and if the server copies the blocks read in by the first request to another disk, the server can switch subsequent requests to the partial replica it just created.

Finally, *random replication* creates and deletes replicas randomly and—in a basic application—doesn't account for differences in access frequency to the different files.

—

Replication addresses requirements such as availability and latency and, when used appropriately, is an important way to increase performance. Because several devices have copies of data elements, the system is more resilient to failures and can access a replica to avoid long waiting times for an overloaded device.

Many Internet systems use static replication. To balance loads among file servers, researchers have proposed automatic replication of the relatively small and frequently accessed read-only system files.[4] Other work uses estimated load information for video-object placement as the basis of a static placement policy.[5] Such a policy complements automatic replication because it reduces dynamic imbalances (although it cannot eliminate them).

Dynamic segment replication, threshold-based dynamic replication, and partial replication have several similar properties. Researchers have studied them in multimedia (continuous media) scenarios and shown that by dynamically creating and deleting replicas on the basis of user accesses, they can improve resource use, the rate of admitted clients, and load balance. Furthermore, while researchers have long viewed the random policy as inappropriate—at least in multimedia scenarios—the unpredictability of user behavior has led to several studies of random placement and replication. Such studies have shown that randomly placing replicas on randomly chosen disks gives performance equal to that of conventional striping schemes in multimedia systems.[6]

Which mechanism to choose depends on several factors, including how frequently file popularity changes, and file and object size. Also, in a highly loaded system, you must consider the replication mechanism's overhead; creating a replica and migrating a user to it in midstream, for example, can be complicated. However, creating and deleting replicas dynamically based on the current load seems efficient and can reduce sensitivity to workload changes and system modifications. [7] Nevertheless, the replication mechanism you apply should account for all possible parameters, because making a copy of a multimedia data element on another device is expensive.

Load balancing

Data interleaving, which accesses only a few disks for each I/O operation, can cause load imbalance (resulting in low overall performance and high latencies). This is because all concurrent requests might come to the same group of disks, leaving others unused. Replication, striping, and interleaving all deal with load imbalance, but researchers have also proposed several other approaches.

Some projects, for example, address the problem of assigning media streams to disks to achieve a balanced disk array load.[2,5] Additionally, to determine imbalance across partitions, one of these projects presents a model to determine which partition sizes best utilize resources.[2] The direct-access storage device (DASD) dancing load-balancing policy determines whether the most frequently accessed files can be played out from memory, how to best assign and replicate such files to striped disk groups, and how to shift existing streams to another disk group's replica in case of an overload. The generalized staggered distributed cyclic layout (G-SDCL) policy tries to avoid hot spots in the disk array (or in the storage node pool) by supporting arbitrary playout modes in various speeds. This policy interleaves data in a round-robin manner but staggers the cyclic layout for each round—that is, it stores the first data segment in each round on a different disk. The prime round-robin (PRR) placement policy is similar to G-SDCL. Both policies try to avoid load imbalance by introducing a rounding distance based on a prime number to evenly distribute disk accesses at any retrieval speed.

BUFFER MANAGEMENT

Given multimedia applications' high data consumption rate, they often replace data before it's reused. Therefore, a complex, computationally expensive caching or page replacement algorithm can be wasteful with respect to required CPU cycles versus reduction of disk I/O operations. It thus can make sense to apply a traditional, low-overhead algorithm such as least recently used (LRU). In some multimedia scenarios, however, data can be reused and proper replacement algorithms can increase performance significantly.

For multimedia scenarios, we place buffering algorithms into two classes. The first is *block-based buffer management*. These caching and replacement algorithms evaluate each block independently for replacement and are typically implemented using an LRU variant. In the multimedia systems context, researchers propose algorithms such as least/most relevant for presentation (L/MRP), which considers parameters such as presentation mode and presentation point. Other L/MRP variations include Q-L/MRP, which offers QoS support, and MPEG-L/MRP.

The second class is *stream-based buffer management*. These policies try to minimize disk accesses on the basis of the observation that if there are many requests for the same video, they're likely to retrieve the same data elements within a short period of time. The system therefore performs caching by keeping in memory data for a stream that follows temporarily close to another stream of the same object. One example of this type of policy is (generalized) interval caching.

Although a system might replace data before it can be reused, caching often offers the benefit of reduced disk operations and should therefore be applied. On the server side, a complex, block-based algorithm is often too CPU intensive, but stream-oriented algorithms that make cache decisions based on the distance between clients might be appropriate. Additionally, as heterogeneous devices with different capabilities increase, support for strided access as a generalization of the stream-oriented approach might be an interesting research direction. Such support is useful, for example, in systems that have multiprotocol support and scalable content.

Another buffer-management consideration is the timeliness of data retrieval from the storage devices. Demand paging is typically inappropriate in high-data-rate, timed applications such as multimedia streaming systems. Prefetching data from disk to memory is therefore better suited to support continuous playback of time-dependent data types. Prefetching preloads data from slow, high-latency storage devices such as disks, to fast, low-latency storage such as main memory. This approach dramatically reduces the data read requests' response time. It also increases the disk I/O bandwidth, because an operation usually retrieves more data by batching I/O requests for the same file—at least when the system stores the requested data contiguously. Obviously, you can use knowledge (or estimations) of application behavior for both replacement and prefetching, and many file systems optimize disk accesses using a read-ahead mechanism if the system reads data sequentially.

With multimedia presentations, prefetching mechanisms can often take advantage of the sequential access pattern, and researchers have proposed several mechanisms to that end. For example, L/MRP calculates relevance values such that the system prefetches, and does not replace, data elements for a given interval in front of the current playout position. A similar read-ahead mechanism retrieves data before it's requested if the system determines that the accesses are sequential.[8] Another system determines the data needed in the next period on the basis of a trade-off between the maximum concurrent streams and the initial delay.[9] This system prefetches this data into a shared buffer (assuming a linear playout of the continuous data stream). Other research describes models for preloading data according to the loading and consuming rate and its available buffers.[10,11] Of course, prefetching data in a multimedia scenario again depends on access patterns. So, a simple read-ahead-like mechanism can often be sufficient in a pure playout application such as news on demand, video on demand, and learning on demand. The amount of data to prefetch should be a trade-off between the data rate, the retrieval rate, and buffer availability.

FILE SYSTEM METADATA STRUCTURES

File systems use a data structure to hold metadata, such as file name, size, owner, and permissions, and data pointers to a file's storage blocks. Traditionally, Unix-like systems have used i-nodes (with direct, indirect, double indirect, and triple indirect block pointers), and Windows systems have used file allocation tables (with a unidirectional linked list).[12] However, these systems require a metadata structure lookup for each accessed block. Systems supporting extent-based allocation modify this metadata structure. Instead of having a pointer to each block, the metadata structure points to the extent. So, for example, systems such as SGI's extended file system (XFS) and Minorca have a pointer to the first block of a contiguous section and a counter holding the extent's length. Similarly, the Windows NT file system (NTFS) has a master file table with a record for each file that defines block runs and extents—that is, a start address and the number of adjacent blocks—in the same way. Thus, metadata structures that use extent (rather than block) pointers reduce the number and complexity of lookups. Additionally, many file systems provide metadata for file- or application-specific tasks.

There are three basic approaches to implementing and using file system metadata. The first, *application metadata*, is information relevant only for the application, such as the format type, data rates, extended access rights, and copyright issues. Application (or media) metadata is important because applications sometimes need it to provide required services. Multimedia file systems often implement this metadata, and some standard file systems (such as Macintosh's Hierarchical File System) also support it.

The second approach, *operational metadata*, changes how the system processes API calls (particularly standard ones). This metadata is similar to application metadata, but the kernel interprets the metadata as well and behaves accordingly. In the system context, operational metadata might be more important than application metadata, because it's sometimes used to select better performance-optimization mechanisms. Symphony, for example, uses data type information to select appropriate modules, which, among other things, changes its interpretation of block sizes. Tiger Shark retrieves a file's default data rate from its metadata and uses it for data prefetching.

The third approach, *integral metadata*, is information integrated into the file system structure itself—the equivalent of the Unix virtual file system i-node and block structures. Integral metadata structures can give applications timeliness support by enabling straightforward access to temporal data units.[13] The Video File Server, for example, stores information for substream synchronization in integral structures.

PUTTING IT ALL TOGETHER

Although many of the techniques we've presented here were developed independently, to evaluate a storage system's overall performance you must integrate the appropriate techniques into a complete storage system. Doing so in a way that increases performance and flexibility for a mixed interactive workload is, in our view, the main research challenge.

Types of file systems

Developers have previously created storage systems that integrate several subcomponents. Although such systems have been labeled inconsistently—file system, file server, and storage server, for example—applications always access them through file system abstractions. We can classify multimedia application file systems as *general file systems*, *multimedia file systems*, and *high-performance file systems*.

General file systems. These systems support all applications rather than a specific application area. Examples include file allocation table (FAT), NTFS, second extended file system (Ext2), and fast file system (FFS). Although general file systems try to support all application classes, they're not optimized for multimedia applications. So, in a multimedia system context, general file systems typically provide performance benchmarks only.

Multimedia file systems. These systems address multimedia requirements such as high throughput, low delay, and synchronization (we described these in Part 1 of this survey). Examples include Video File Server, Shark, Everest, continuous media file system (CMFS), Fellini, Symphony, Tiger Shark, Minorca, embedded real-time file system (ERTFS), XFS, parallel multimedia file system (PMFS), and multimedia integrated parallel file system (MiPFS). Among these, Minorca and XFS are general file systems with multimedia support, while most of the others support only multimedia applications.

Multimedia file systems address multimedia applications' distinguishing feature: they have soft real-time constraints. Many file systems focus on scheduling to address this demand. Additionally, the degree of multimedia support varies. Developers have created systems (and system components) exclusively for streaming (such as ERTFS, Tiger Shark, Shark, CMFS, Video File Server, and Everest), for combining with a second, non-real-time class to accommodate mixed workloads (such as Fellini), and for serving several application classes (such as Symphony and the adaptive disk scheduler for mixed-media workloads (APEX).

High-performance file systems. These systems are primarily for applications that must read and write large data amounts in a very short time. Examples include General Parallel File System (GPFS), clustered extended file system (CXFS), Frangipani, global file system (GFS), parallel portable file system (PPFS), Exemplar, and extensible file system (ELFS).

Unlike multimedia file systems, high-performance file systems do not need timeliness guarantees, but their overall performance must be maximized (for experiments in physics and large-scale simulations, for example). Newly developed high-performance file systems concentrate on this task, while high-performance file systems such as GPFS and CXFS—which are derived from multimedia file systems—have gained scalability while continuing to support resource reservation. However, because of their high performance for large data amounts, even independently developed file systems can be used for multimedia applications and compete with multimedia file systems. Furthermore, as multimedia applications become more interactive and include less predictable read—and write—operations, we should consider high-performance file system features. Many high-performance file systems support operations such as efficient strided reading and writing and other noncontiguous operations by providing a separate API or detecting access patterns. These abilities will be increasingly useful for multimedia applications that offer editing tasks, protocol-dependent packaging, scalable streaming media, and nonlinear multimedia.

Storage system evolution

Prashant Shenoy's analysis of new requirements—which includes the need for integrated file systems that support a variety of applications[14]—offers a comprehensive approach to modernizing multimedia file systems. He considers the development of server-independent, self-healing, self-managing networked file systems a new goal and predicts that more functions will be off-loaded to disk processors. Although storage area networks (SAN) and network-attached storage (NAS) products already claim such abilities, Frank Schmuck and Roger Haskin's recognition of current system limitations led to GPFS's integrated approach.[15]

Integration

Typically, newer storage systems address a breadth of workloads and take many subcomponents into account, whereas older ones often focus on a single component. In the latter case, developers must integrate suitable subcomponents into one "optimal" system for a particular workload, which is a significant challenge. As we mentioned earlier, decisions for each subcomponent often depend on the expected access patterns, but developers must also consider the properties of the mechanisms chosen for the other components. So, developers must make appropriate choices for all components or mechanisms if they're to work well together.

For example, data placement on the disk and disk scheduling are tightly coupled because the disk arm movement depends on the platter location of each data block. Because many disk scheduling algorithms give shortest access time to data blocks near the disk's center, the most popular data could be placed according to this property as in skewed organ-pipe placement.

On the other hand, if developers apply a buffer-management algorithm that tries to store in memory the most frequently accessed data—as in generalized interval caching or L/MRP —they'll often end up with a counterproductive data placement strategy. That's because the requested data probably resides in memory, so the disk isn't accessed during I/O operations that access popular data items, which is the basis of disk block placement optimization. (That is, the disk serves a request distribution that is different from that of the application.)

Furthermore, reorganizing blocks based on popularity is expensive and typically should be minimized. However, dynamic block reorganization can be useful. For example, developers can increase scalability by dynamically reorganizing and replicating a block granularity spanning several disks. This prevents a single disk from being overloaded by frequent access to individual blocks. It also reduces partial file locking in interactive applications that include write operations, such as online gaming.

Improving performance

Although some mechanisms have contradicting properties, you can combine several mechanisms in different ways to achieve good storage system performance. One interesting example would be to use a hierarchical mixed-media scheduler that

- Supports different service classes with heterogeneous requirements
- Removes device idle time and increases overall performance (device utilization) through work conservation
- Improves disk efficiency by sorting requests in a round according to their platter placement.

Because video and audio streams can benefit greatly from sequential prefetching (read-ahead), you should use some kind of adjacent block placement to minimize intrarequest seeks. Pure continuous placement results in large "file sized" seeks as the system retrieves several files in parallel. That is, it might be best to use a type of extent-based placement with an extent size that's large enough to accommodate expected read sizes during one round, avoiding wasteful intrarequest seeks.

As we discussed earlier, caching can improve performance and possibly reduce startup latency for new requests. However, the data rates in multimedia scenarios are high, and the benefits might be small because data might be replaced before it's reused. Given this, you shouldn't use a complex, block-based, CPU-intensive algorithm such as L/MRP in a server. Stream-oriented algorithms that make cache decisions based on the average time between requests might be appropriate.

With index structures, you can reduce the number of accesses required to find the requested data blocks' address using some kind of "extent info" that points to several continuous blocks in a structure entry (such as the location and length of the extent's first block).[16,17] In contrast, adapting application-specific index structures[13] to new application requirements is difficult and should be avoided for future integrated file systems.

To provide multimedia-specific information to the file system, applications access the storage system through the operating system's system call interface. You can solve this by either using an entirely proprietary API or extending the operating system's standard file system API. Because existing applications are often implemented on commodity operating systems, using extended standard APIs is typically the easiest approach.

To meet a multimedia server's demand for storage space, reliability, fault tolerance, and I/O bandwidth, it is often best to use multiple parallel disks. However, because single disks today support several concurrent streams (for example in MPEG-II DVD quality), striping units must be as small as possible—while still meeting playout rates—to best utilize overall disk bandwidth. So, you don't need full striping to achieve high-enough bandwidths, but you can use mechanisms like staggered striping, along with replication, to distribute the workload on several devices. The "Existing Systems" sidebar describes some current systems and how they address multimedia storage challenges.

RESEARCH DIRECTIONS

Future applications will integrate both time-dependent multimedia data types, such as audio and video, and time-independent data types, such as text, graphics, and images. Because the storage systems found in current commercial operating systems were designed for best-effort applications, they often cannot store and retrieve such data efficiently.

It is not clear how future systems will better support multimedia applications, but storage system design should focus on application requirements, access patterns, performance characteristics, and the behavior of underlying storage devices. This task is complex; all these factors change over time, and storage system designs often lag behind the changes.

For example, early storage systems optimized disk scheduling and placement on the basis of issues such as rotational delay. Now, most disks do whole-track reads and cache the track in the internal buffer. So, careful data placement within a track is a waste of time, as is disk scheduling at too fine a granularity.

Also, issues such as bad-block replacement can affect timing-sensitive storage systems. Disks will often transparently replace a damaged block with another one that might be located far away. Because disks present a virtual, rather than physical, view of storage, contiguous reading can involve more seeking than expected.

In addition, multimedia storage systems will probably use complex delivery systems—such as NAS, Internet small computer system interface (iSCSI), and Virtual Shared Disks—that are hidden behind block device interfaces. Such systems require new storage system approaches because they reduce control over block placement and can introduce contention for bandwidth and jitter.

Generally, giving disks some freedom to reorder requests (a common feature in newer disks that might receive multiple requests) will improve throughput. Most multimedia applications use buffers (typically on the client side) to provide tolerance for small delays and smooth jitter before presentation. So, for many such applications, we can relax strict timing requirements in the storage system to better utilize the inherent access pattern predictability and thus support prefetching and focus on periodical services.

Furthermore, there is an emerging trend toward letting systems support multiple application classes with heterogeneous performance requirements. Adding such support in a middleware layer increases runtime overhead and reduces application isolation.[18] However, experiments show that an integrated server typically outperforms an integration layer on a partitioned server.[19] Implementing such a file system is more complex, but the system is easier to administrate and eases the integration of new service classes.

As the number of application classes increases, storage systems must be able to manage various loads and heterogeneous requirements. To fully support different service classes, all system components must be appropriately designed and implemented. Storage system research should thus focus on systems that can automatically adapt to particular workloads by adapting their data layout, scheduling strategy, and buffer management approaches. Finally, although researchers have proposed different storage system mechanisms, policies, and options for multimedia (or mixed-media) support, much work remains on how to integrate suitable mechanisms and analyze their combined performance.

References

1. H.M. Vin , S.S. Rao, and P. Goyal , "Optimizing the Placement of Multimedia Objects on Disk Arrays," *Proc. 1995 Int'l Conf. Multimedia Computing and Systems* (ICMCS 95), IEEE CS Press, 1995, pp. 158-165.

2. P.J. Shenoy and H.M. Vin , "Efficient Striping Techniques for Multimedia File Servers," *Proc. 7th Int'l Workshop Network and OS Support for Digital Audio and Video* (NOSSDAV 97), ACM Press, 1997, pp. 25-36.

3. S. Ghandeharizadeh and S.H. Kim , "Striping in Multidisk Video Servers,"*Proc. High-Density Data Recording and Retrieval Technologies Conf.,* SPIE-The Int'l Soc. for Optical Eng., 1996, pp. 88-102.

4. M. Satyanarayanan , et al., "CODA: A Highly Available System for a Distributed Workstation Environment,"*IEEE Trans. Computers* , vol. 39, no. 4, Apr. 1990, pp. 441-459; http://csdl.computer.org/comp/trans/tc/1990/04/t0447abs.htm.

5. A. Dan and D. Sitaram , "An Online Video Placement Policy Based on the Bandwidth to Space Ratio (BSR)," *Proc. 1995 ACM SIGMOD Int'l Conf. Management of Data* (SIGMOD 95), ACM Press, 1995, pp. 376-385.

6. J.R. Santos , R.R. Muntz, and A.R.N. Berthier , "Comparing Random Data Allocation and Data Striping in Multimedia Servers,"*Proc. 2000 ACM SIGMETRICS Int'l Conf. Measurement and Modeling Computer Systems* (SIGMETRICS 00), ACM Press, 2000, pp. 44-55.

7. C.F. Chou , L. Golubchik, and J.C.S. Lui , "A Performance Study of Dynamic Replication Techniques in Continuous Media Servers,"*Proc. 8th Int'l Symp. Modeling, Analysis, and Simulation of Computer and Telecommunication Systems* (MASCOTS 00), IEEE CS Press, 2000, pp. 256-266;
http://csdl.computer.org/comp/proceedings/mascots/2000/0728/00/07280256abs.htm.

8. D.C. Anderson , et al., "Cheating the I/O Bottleneck: Network Storage with Trapeze/Myrinet,"*Proc. 1998 Usenix Ann. Tech. Conf.,* Usenix, 1998, pp. 143-154.

9. H. Tezuka and T. Nakajima , "Simple Continuous Media Storage Server on Real-Time Mach,"*Proc. 1996 Usenix Ann. Tech. Conf.,* Usenix, 1996, pp. 87-98.

10. P.V. Rangan and H.M. Vin , "Efficient Storage Techniques for Digital Continuous Multimedia,"*IEEE Trans. Knowledge and Data Eng.* , vol. 5, no. 4, Aug. 1993,pp. 564-573; http://csdl.computer.org/comp/trans/tk/1993/04/k0564abs.htm.

11. A. Zhang and S. Gollapudi , *"QoS Management in Educational Digital Library Environments,"* tech. report CS-TR-95-53, Dept. of Computer Science, State Univ. of New York at Buffalo, 1995.

12. A.S. Tanenbaum , *Modern Operating Systems,* Prentice Hall, 2001.

13. P.V. Rangan and H.M. Vin , "Designing File Systems for Digital Video and Audio,"*Proc. 13th ACM Symp. OS Principles* (SOSP 91), ACM Press, 1991, pp. 81-94.

14. P.J. Shenoy , "The Case for Reexamining Integrated File System Design,"*Proc. 10th Int'l Workshop Network and OS Support for Digital Audio and Video* (NOSSDAV 00), ACM Press, 2000, pp. 51-54.

15. F. Schmuck and R. Haskin , "GPFS: A Shared-disk File System for Large Computing Clusters,"*Proc. 1st Conf. File and Storage Technologies* (FAST02), Usenix Assoc., 2002, pp. 231-244.

16. A. Sweeney , et al., "Scalability in the XFS File System,"*Proc. 1996 Usenix Tech. Conf.,* Usenix, 1996, pp. 1-14.

17. C. Wang , V. Goebel, and T. Plagemann , "Techniques to Increase Disk Access Locality in the Minorca Multimedia File System,"*Proc. 7th ACM Int'l Multimedia Conf. (Part 2)* (MM 99), ACM Press, 1999,pp. 147-151.

18. P.J. Shenoy , et al., "Middleware versus Native OS Support: Architectural Considerations for Supporting Multimedia Applications,"*Proc. 8th IEEE Real-time Technology and Applications Symp.* (RTAS 02), IEEE CS Press, 2002, pp. 23-34.

19. P.J. Shenoy , P. Goyal, and H.M. Vin , "Architectural Considerations for Next Generation File Systems,"*Proc. 7th ACM Int'l Multimedia Conf. (Part 1)* (MM 99), ACM Press, 1999, pp. 457-467.

Pål Halvorsen is an associate professor in the University of Oslo's Department of Informatics. His research activities focus mostly on distributed-multimedia-system support, including operating systems, storage and retrieval, communication, and distribution. He received his doctoral degree in computer science from the University of Oslo. Contact him at the Dept. of Informatics, Univ. of Oslo, PO Box 1080 Blindern, N-0316 Oslo, Norway; paalh@ifi.uio.no.

Carsten Griwodz is an associate professor in the University of Oslo's Department of Informatics. His research interests include improved mechanisms and algorithms for media servers and distribution systems. He received his doctoral degree from the Darmstadt University of Technology. Contact him at the Dept. of Informatics, Univ. of Oslo, PO Box 1080 Blindern, N-0316 Oslo, Norway; griff@ifi.uio.no.

Vera Goebel is a professor in the University of Oslo's Department of Informatics. Her research interests include multimedia database systems and operating systems, multimedia middleware, quality of service, and interactive distributed multimedia applications. She received her PhD in computer science from the University of Zurich. Contact her at the Dept. of Informatics, Univ. of Oslo, P.O. Box 1080 Blindern, N-0316 Oslo, Norway; goebel@ifi.uio.no.

Ketil Lund works as a post-doctoral researcher at the UniK-University Graduate Center. His research interests include multimedia database systems, disk scheduling, quality of service, and video on demand. He received his PhD in computer science from the University of Oslo, where his doctoral work focused on disk scheduling for multimedia database management systems. Contact him at UniK-Univ. Graduate Center, PO Box 70, N-2027 Kjeller, Norway; ketillu@unik.no.

Thomas Plagemann is a professor in the University of Oslo's Department of Informatics. His research interests include multimedia middleware, protocol architectures, quality of service, operating system support for distributed multimedia systems, content distribution infrastructures, and interactive distance learning. He received his Doctor of Technical Science from the Swiss

Federal Institute of Technology (ETH) Zurich. Contact him at the Dept. of Informatics, Univ. of Oslo, P.O. Box 1080 Blindern, N-0316 Oslo, Norway; plageman@ifi.uio.no.

Jonathan Walpole is a full professor in the Computer Science and Engineering Department and is the director of the Systems Software Laboratory at the OGI School of Science and Engineering at Oregon Health & Science University. His research focuses on adaptive systems software and its application in distributed, mobile, and multimedia computing environments. He received his PhD in computer science from Lancaster University, UK. Contact him at the Dept. of Computer Science and Eng., Oregon Graduate Inst. of Science and Technology, 20000 N.W. Walker Rd., Beaverton, OR 97291-1000; walpole@cse.ogi.edu.

Existing Systems: A Sampling

Research systems have typically focused on enhancing individual storage system components to accommodate multimedia applications. Commercial products that deliver complete systems usually combine several mechanisms targeted at individual storage system elements; following is a survey of some working systems that address most of the issues we have covered in this survey.

IBM's General Parallel File System (GPFS) integrates many elements,[1] but stops at a block device interpretation of the disk itself. The block device might offer access to a RAID (redundant array of independent disks) system or Virtual Shared Disks. GPFS handles data placement on single and multiple block devices using parameters defined by the system administrator. It uses block distribution and replication to balance load and improve throughput and fault tolerance. Although GPFS knows only a fixed, large block size per file system, it uses subblocks to efficiently store small files. GPFS provides applications with standard APIs and extended APIs that, for example, provide hints for specific access patterns. The system's buffer management supports prefetching and access pattern prediction to reduce access time. Its directory handling is optimized for numerous entries, and it supports application metadata.

SGI's clustered extended file system (CXFS) integrates elements similar to those in GPFS but is based on the XVM volume manager. XVM provides striping and replication for load balancing and fault tolerance. The system administrator manages the type and degree of XVM striping and replication; CXFS does not influence placement in any way. For data placement, CXFS's block size is determined when the file system is created, and its extents consist of an arbitrary number of consecutive blocks.

Spiffi, a parallel file system for Intel's Paragon architecture, is implemented on top of its individual nodes' file systems. Spiffi stripes files over the nodes's disks in a round-robin manner and determines striping granularity at file creation time. The system has a fixed block size, but, if possible, it stores blocks belonging to the same file contiguously on each node. Spiffi prefetches to memory, delays write at each node, and runs parallel transfers between its nodes and client nodes. One case study added video-on-demand system enhancements, including a buffer-management policy that keeps requested buffers in memory, even though the least-recently-used (LRU) algorithm would remove them; priority Scan (PScan) disk scheduling; and prefetching based on priority within a maximum prefetch time window. [2]

The University of Southern California's Integrated Media Systems Center developed Yima, a real-time streaming architecture derived from the Mitra server. Yima's storage system is distributed over standard PCs. Yima can place data in a round-robin or pseudorandom manner; in the latter case, a known seed-value determines the placement per file to reduce metadata overhead. Yima offers replication to improve throughput, and uses disk merging to implement parity-based redundancy for fault tolerance and to hide device heterogeneity. In Yima, buffering and prefetching are left to the client applications, which can retrieve data concurrently from all server nodes.

Lucent Technologies' CineBlitz is a commercial product based on the Fellini storage system. Fellini is built for single-node servers and focuses on buffer management. Its file system uses fixed block sizes, and developers can place data using different policies. Fellini typically stores media files contiguously, but it can also use a striped disk array. The system's buffer cache is organized in pages that are managed per file; if possible LRU replaces the pages within the buffers assigned to that file. The system determines the number of buffers for each file on the basis of admission control and the number of clients that use the file concurrently. Fellini does not provide a standard file system API, and it has no hierarchical file system structure. It does, however, let applications specify the required data rate.

Each of these systems considers key storage system topics for continuous-media applications. As you would expect, they consider buffer management important, and only the experimental Yima system leaves it to the client. Also, only the older Fellini storage system exclusively uses a proprietary file system API; the other systems come with standard APIs. Finally, only GPFS makes online decisions for striping and replication to adapt to varying access patterns and throughput demands.

References

1. F. Schmuck and R. Haskin , "GPFS: A Shared-disk File System for Large Computing Clusters,"*Proc. 1st Conf. File and Storage Technologies* (FAST02), Usenix Assoc., 2002, pp. 231-244.
2. C.S. Freedman and D.J. DeWitt , "The Spiffi Scalable Video-on-Demand System,"*Proc. ACM Int'l Conf. Management of Data* (ACM SIGMOD), ACM Press, 1995, pp. 352-363.