Symphony: An Integrated Multimedia File System*

Prashant J. Shenoy, Pawan Goyal, Sriram S. Rao, and **Harrick M. Vin**

Distributed Multimedia Computing Laboratory
Department of Computer Sciences, University of Texas at Austin
Taylor Hall 2.124, Austin, Texas 78712-1188, USA
E-mail: {shenoy,pawang,sriram,vin}@cs.utexas.edu, Telephone: (512) 471-9732, Fax: (512) 471-8885
URL: http://www.cs.utexas.edu/users/dmcl

ABSTRACT

An integrated multimedia file system supports the storage and retrieval of multiple data types. In this paper, we first discuss various design methodologies for building integrated file systems and examine their tradeoffs. We argue that, to efficiently support the storage and retrieval of heterogeneous data types, an integrated file system should enable the coexistence of multiple data type specific techniques. We then describe the design of Symphony–an integrated file system that achieves this objective. Some of the novel features of Symphony include: a QoS-aware disk scheduling algorithm; support for data type specific placement, failure recovery, and caching policies; and support for assigning data type specific structure to files. We discuss the prototype implementation of Symphony, and present results of our preliminary experimental evaluation.

Keywords: Multimedia file systems, multimedia operating systems, integrated file systems

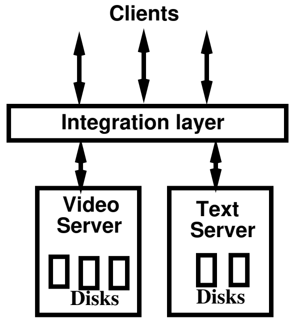
1. INTRODUCTION

Recent advances in compression, storage, and communication technologies have resulted in the proliferation of applications that involve storing and retrieving multiple data types such as text, audio, video, imagery, etc. (collectively referred to as *multimedia*). Realizing such next generation applications requires the development of file systems that can efficiently manage the storage and retrieval of multiple data types (referred to as *integrated multimedia file systems*). Most existing file systems have been optimized for a single data type. For instance, conventional file systems have been optimized for the storage and retrieval of textual data.[1] Since these file systems provide only a best-effort service to user requests, they are ill-suited for meeting the real-time performance requirements of audio and video data (continuous media). On the other hand, continuous media servers exploit the periodic and sequential nature of continuous media accesses to improve server throughput, and employ resource reservation algorithms to provide real-time performance guarantees.[2–6] Most of these servers, however, assume a predominantly read-only environment with substantially less stringent response time requirements, and hence, are unsuitable for textual data. Thus, existing file systems are inadequate for managing heterogeneous data types. The design and implementation of a file system that addresses these limitations types constitutes the subject matter of this paper.

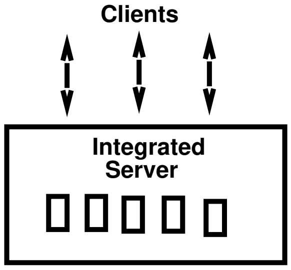
This paper makes the following research contributions. First, we propose two different architectures for designing integrated multimedia file systems, namely physically and logically integrated file systems. We examine the tradeoffs between these architectures and demonstrate that physically integrated file systems yield better utilization of file system resources. Next, we discuss the requirements imposed by various data types on physically integrated file systems. Specifically, we discuss the requirements imposed on the service model, the retrieval architecture, and techniques for placement, fault tolerance, caching, and meta data management employed by the file system. We conclude from these requirements that, to efficiently support heterogeneous data types, an integrated file system should enable the coexistence of multiple data type specific techniques.

Finally, we present the design and implementation of *Symphony*–a physically integrated file system that meets these requirements. Symphony consists of a number of novel features including: (1) a QoS-aware disk scheduler that efficiently supports both real-time and non-real-time requests, (2) a storage manager that supports multiple block sizes and allows control over their placement, thereby enabling diverse placement policies to be implemented in the file system, (3) a fault-tolerance layer that enables data type specific failure recovery, and (4) a two level meta data (inode) structure that enables data type specific structure to be assigned to files while continuing to support the traditional byte stream interface. In addition to these novel features, Symphony synthesizes a number of recent innovations into the file system. These features include: (1) resource reservation (i.e., admission control) algorithms to provide QoS guarantees,[7] (2) support for client-pull and server-push architectures,[8] (3) support for fixed-size and variable-size blocks,[9] and (4) support for data type specific caching techniques.[10,11]

*This research was supported in part by an IBM Faculty Development Award, Intel, the National Science Foundation (Research Initiation Award CCR-9409666 and CAREER award CCR-9624757), NASA, Mitsubishi Electric Research Laboratories (MERL), and Sun Microsystems Inc.



(a) Logically integrated file system



(b) Physically integrated file system

Figure 1. Logically and physically integrated file systems. The logically integrated file system partitions the set of disks among component servers; the physically integrated file server shares all disks among all data types.

Integrating these diverse techniques into the file system is a key challenge, since it can complicate the file system architecture. Symphony addresses this by employing a two layer architecture. The lower layer of Symphony implements a set of data type independent mechanisms (such as disk scheduling, buffer management, storage space management, etc.) that provide core file system functionality. The upper layer uses these mechanisms to implement multiple data type specific policies. It consists of a set of modules, one per data type, each of which implements policies optimized for that data type. Such a two layered architecture allows the file system designer to easily add modules for new data types, or change policies implemented in existing modules without changing the rest of the file system.

We have implemented a prototype of Symphony in Solaris 2.5.1 and have evaluated it in an environment consisting of a cluster of workstations interconnected by an ATM network. We present and analyze our experimental results.

The rest of the paper is organized as follows. In Section 2, we propose and evaluate two different architectures for designing integrated file systems. The requirements imposed by various data types on an integrated file system are outlined in Section 3. Sections 4, 5, and 6 describe the design of Symphony. In Section 7, we experimentally evaluate the Symphony prototype. Section 8 describes related work, and finally, Section 9 summarizes our results.

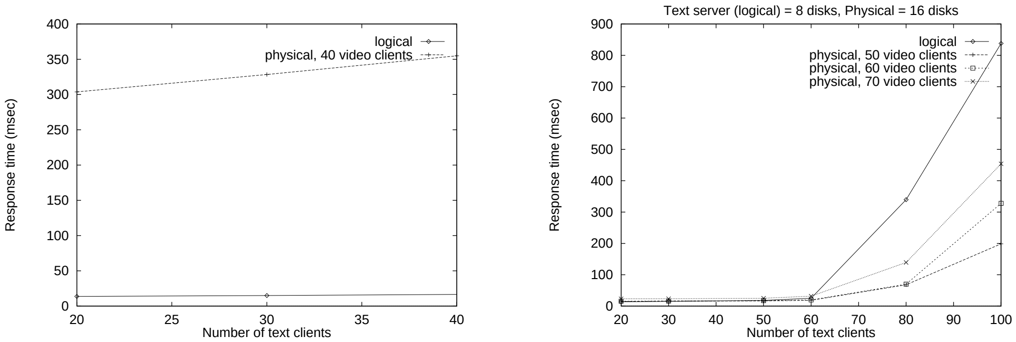
2. DESIGN METHODOLOGIES

There are two methodologies for designing integrated file systems: (1) *Logically integrated* file systems, which consist of multiple file servers, each optimized for a particular data type, glued together by an integration layer that provides a uniform way of accessing files stored on different file servers; and (2) *physically integrated* file systems, which consist of a single file server that stores all data types. Figure 1 illustrates these architectures. Each architecture has its advantages and disadvantages.

Since techniques for building file servers optimized for a single data type are well known,[1,6] logically integrated file systems are easy to implement. In such file systems, the set of disks is statically partitioned among component file servers. This causes requests accessing different component servers to access mutually exclusive set of disks, thereby preventing interference between user requests (e.g., servicing best-effort text requests does not violate deadlines of real-time continuous media requests). However, static partitioning of resources has the following limitations:

- Static partitioning of storage resources is governed by the expected storage space and bandwidth requirements of workloads. If the observed workload deviates significantly from the expected, then repartitioning of disks may be necessary. Repartitioning of disks is tedious and may require the system to be taken offline.[12] An alternative to repartitioning is to add new disks to the server, which causes resources in under-utilized partitions to be wasted.
- The storage space requirements of files stored on a component file server can be significantly different from their bandwidth requirements. In such a scenario, allocation of disks to the component server will be governed by the maximum of the two values. This can lead to under-utilization of either storage space or disk bandwidth on the server.

In contrast, physically integrated file systems share all resources among all data types, and do not suffer from the above limitations. In such servers, static partitioning of resources is not required; storage space and bandwidth are allocated to data types on demand. Such an architecture has several advantages. First, by co-locating a set of files with large storage space but small bandwidth requirements with another set of files with small storage space but large bandwidth requirements, this architecture yields better resource utilization. Second, since resources are allocated on demand, it can easily accommodate dynamic changes in access patterns. Finally, since all the storage resources are shared among all the files, more resources are



(a) SCAN (b) Symphony disk scheduling algorithm

Figure 2. Response times in logically and physically integrated file systems for a 8KB request. Use of the SCAN disk scheduling algorithm in the physically integrated server yields response time that are substantially higher than that in the logically integrated server, whereas use of the Symphony disk scheduling algorithm yields comparable or better performance.

available to service each request, which in turn improves the performance. A disadvantage of physically integrated file systems is that the file system design becomes more complex due to the need for supporting multiple data types. Moreover, since multiple data types share the same set of resources, data types can interfere with each other. For instance, arrival of a large number of text requests can cause deadlines of real-time continuous media requests to be violated. Algorithms that multiplex resources in a physically integrated file system must be designed carefully to prevent such interference.

To quantitatively compare logically and physically integrated file systems, we measured the response times yielded by the two architectures using trace-driven simulations. Each server was assumed to contain sixteen disks. In the logically integrated server, eight disks each were allocated to the continuous media server and the text file server, while in the physically integrated server, all disks were shared by both data types. The text server in the logically integrated case was assumed to use the SCAN disk scheduling algorithm. However, use of SCAN in the physically integrated server yields poor response times for text requests. This is because, SCAN does not differentiate among requests with different requirements and schedules text and video requests merely based on their cylinder numbers. As shown in 2(a), even at a moderate video load, this interleaving causes the response time of the physically integrated server to be substantially higher than that of the logically integrated server. The response time improves significantly if the physically integrated server uses a disk scheduling algorithm that delays real-time video requests until their deadlines and uses the available slack to schedule text requests. By giving priority to text requests over video requests whenever sufficient slack is available, such a scheduler provides low response time to text requests without violating the deadlines of real-time video requests (see Section 5 for a detailed description of the algorithm). Figure 2(b) compares the two architectures when this disk scheduling algorithm is used. The figure shows that the response time of the physically integrated server is comparable to that of the logically integrated server at light text loads. Moreover, at moderate to heavy text loads, the physically integrated server significantly outperforms its counterpart. This is because, at heavy loads, the response time is governed by the queuing delay at a disk (i.e., the time for which a request waits in the disk scheduling queue before it receives service). Since text files are interleaved across all disks, the number of disks servicing text requests, and hence the number of queues, is larger than that in the logically integrated server. This yields a smaller number of requests per queue, and hence, better response times to text requests. At light video loads, the different in the response time for text requests observed in the logically and the physically integrated server is even larger. Thus, by carefully designing the disk scheduling algorithm, a physically integrated server can provide identical or better performance as compared to a logically integrated server.

To summarize, physically integrated file servers obviate the need for static partitioning of resources inherent in the logically integrated servers. The resulting dynamic sharing of resources leads to better performance and higher utilization of resources, albeit at the expense of increased file system complexity. Due to the advantages of physical integration, in this paper, we focus only on physically integrated file systems. Next we discuss the requirements imposed by various data types on physically integrated file systems.

3. REQUIREMENTS FOR A PHYSICALLY INTEGRATED FILE SYSTEM

A physically integrated file systems should achieve efficient utilization of file system resources while meeting the performance requirements of heterogeneous data types and applications. In what follows, we discuss the effect of this objective on the service model and the retrieval architecture supported by the integrated file system, as well as on techniques for efficient placement, fault tolerance, meta data management, and caching.

Service Model

Most conventional file systems provide a single class of best-effort service to all applications regardless of their requirements. A single best-effort service class is inadequate for meeting the diverse QoS requirements of applications (e.g., real-time requirements of continuous media applications). To efficiently support applications with diverse requirements, an integrated file system should support multiple service classes, such as periodic real-time, aperiodic real-time and best-effort. Instantiating multiple service classes will require the file system to employ: (1) a disk scheduling algorithm that isolates service classes from each other and aligns the service provided with the requirements of these classes (e.g., provide low response times to best-effort requests and meet deadlines of real-time continuous media requests), and (2) admission control algorithms to provide performance guarantees to requests within the periodic and aperiodic real-time classes.

Retrieval Architecture

An integrated file system can service user requests using either the *client-pull* or the *server-push* (i.e., *streaming*) architecture. In the former case, the server retrieves data from disks only in response to explicit read requests from clients, while in the latter case, the server periodically retrieves and transmits data to clients without explicit read requests. Typically textual applications are serviced using the client-pull architecture, whereas, due to their periodic and sequential nature, the server-push architecture is better suited for continuous media applications. Adapting continuous media applications to the client-pull architecture is difficult.[8] Also, the server-push architecture is inappropriate for supporting aperiodic textual requests. Hence, to efficiently support multiple data types, an integrated file system will need to support both the client-pull and the server-push architectures.

Placement Techniques

Due to the large storage space and bandwidth requirements of continuous media, integrated file systems will use disk arrays for storing data. To effectively utilize the array bandwidth, the file server must *interleave* (i.e., *stripe*) each file across disks in the array. Placement of files on such striped arrays is governed by two parameters: (1) the *stripe unit size*[†], which is the maximum amount of logically contiguous data stored on a single disk, and (2) the *striping policy*, which maps stripe units to disk locations. Both parameters depend significantly on the characteristics of the data type. For instance, the large bandwidth requirements of real-time continuous media yields a stripe unit size that is an order of magnitude larger than that for text.[13] Use of a single stripe unit size for all data types either degrades performance for data types with large bandwidth requirements (e.g., continuous media), or causes internal fragmentation in small files (e.g., textual files). Similarly, we have shown that a policy that stripes each file across all disks in the array is suitable for textual data, but yields sub-optimal performance for variable bit-rate continuous media.[13] Moreover, since continuous media files can have a multi-resolution nature (e.g, MPEG-2 encoded video), the striping policy optimizes the placement of such files by storing blocks belonging to different resolutions adjacent to each other on disk.[14] Such contiguous placement of blocks substantially reduces seek and rotational latencies incurred during playback. No such optimizations are necessary for "single resolution" textual files. Since placement techniques for different data types differ significantly, to enhance system throughput, an integrated file system should support multiple data type specific stripe unit sizes and striping policies.

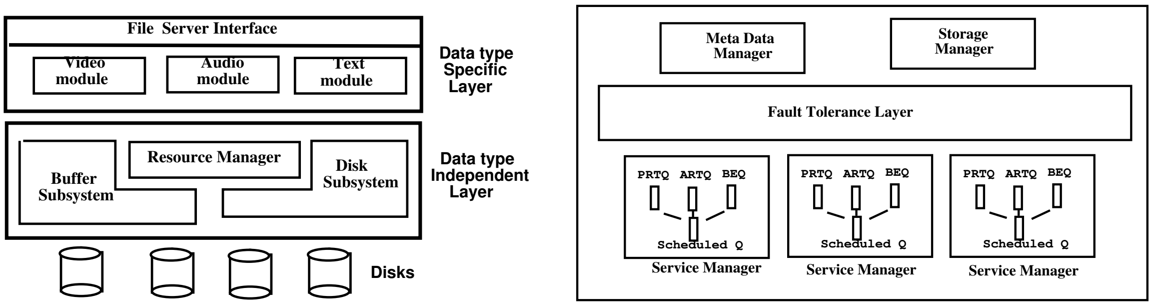
Meta data structures

Typically textual files are accessed as a sequence of bytes. For continuous media files, however, the logical unit of access is a video frame or an audio sample. To access a file as a sequence of logical units, an application must either maintain logical unit indices, or dynamically generate this information at run-time. Developing such applications would be simplified if the file system were to support logical unit access to files. Such support is also required to implement the server-push architecture since the file system needs logical unit sizes (e.g., frame sizes) to determine the amount of data that must be accessed on behalf of clients. Consequently, an integrated file system must maintain meta data structures that allow both logical unit and byte level access to files, thereby enabling any data type specific structure to be assigned to files.

Failure Recovery Techniques

Since disk arrays are highly susceptible to disk failures, a file server must employ failure recovery techniques to provide uninterrupted service to clients even in the presence of failures.[15] Disk failure recovery involves two tasks: *on-line reconstruction*, which involves recovering the data stored on the failed disk in response to an explicit request for that data; and *rebuild*, which involves creating an exact replica of the failed disk on a replacement disk. Conventional textual file systems use mirroring or parity-based techniques for exact on-line reconstruction of data blocks stored on the failed disk.[15,16] Some continuous media applications with stringent quality requirements also require such exact on-line reconstruction of lost data. However, for many continuous media applications, *approximate* on-line reconstruction of lost data may be sufficient. For these applications, several on-line failure recovery techniques that exploit the spatial and temporal redundancies present within continuous media data to reconstruct a reasonable approximation of the data stored on the failed disk have been developed.[17] Unlike mirroring and

[†]A stripe unit is also referred to as a media block. We use these terms interchangeably in this paper



(a) Two level architecture of Symphony (b) Disk Subsystem Architecture

Figure 3. Architecture of Symphony

parity based techniques, these techniques do not require any additional data to be accessed for failure recovery, and thereby significantly reduce the failure recovery overheads. Hence, to enhance system utilization, an integrated file system must support multiple data type specific failure recovery techniques.

Caching Techniques

A caching policy such as LRU improves response times for text requests. It is well known that LRU performs poorly for sequential data accesses,[10] and hence, is inappropriate for continuous media. Policies that are tailored for sequential data accesses, such as Interval Caching,[18] are more desirable for continuous media, but are unsuitable for textual data. Since no single policy is suitable for all data types, an integrated file system will need to support multiple data type specific caching policies.

From the above discussion, we conclude that to efficiently support different data types, an integrated file system *should enable the coexistence of multiple data type specific techniques*. In what follows, we describe the design and implementation of *Symphony*–a physically integrated file system that meets these requirements.

4. ARCHITECTURE OF SYMPHONY

The need for supporting multiple data type specific techniques in a file system is a key challenge. Symphony addresses this by employing a two layer architecture. The lower layer of Symphony (data type independent layer) implements a set of data type independent mechanisms that provide core file system functionality. The upper layer (data type specific layer) consists of a set of modules, one per data type, which use the mechanisms provided by the lower layer to implement data type specific techniques for placement, failure recovery, caching, etc. The layer also exports a file server interface containing methods for reading, writing, and manipulating files. Figure 3(a) depicts this architecture. Such a two layer architecture provides clean separation of the data type independent mechanisms from the data type specific policies. This allows the file system designer to easily add modules for new data types, or modify techniques implemented in existing modules. We now describe each of these layers in detail.

5. THE DATA TYPE INDEPENDENT LAYER

The data type independent layer of Symphony multiplexes storage space, disk bandwidth, and buffer space among different data types. It consists of three components: the *disk subsystem*, the *buffer subsystem*, and the *resource manager* (see Figure 3(a)).

5.1. The Disk Subsystem

The disk subsystem of Symphony is responsible for efficiently multiplexing storage space and disk bandwidth among different data types. It consists of four components (see Figure 3(b)): (1) a *service manager* that supports mechanisms for efficient scheduling of best-effort, aperiodic real-time, and periodic real-time requests; (2) a *storage manager* that supports mechanisms for allocation and deallocation of blocks of different sizes, as well as techniques for controlling their placement on a disk array; (3) a *fault tolerance layer* that enables multiple data type specific failure recovery techniques; and (4) a *meta data manager* that enables data type specific structure to be assigned to files. The key features and the algorithms used to implement these components are described below.

5.1.1. Service Manager

The main objective of this component is to support multiple service classes, and meet the quality of service requirements of requests within each class. The service manager supports three service classes: best-effort, periodic real-time and aperiodic real-time; and two service architectures: client-pull and server-push. Requests in the best-effort class do not have any deadlines, but desire low average response times. Periodic and aperiodic real-time requests have deadlines that must be met by the service manager. Best-effort and aperiodic real-time requests can arrive at arbitrary instants and are serviced using the client-pull architecture. Periodic real-time requests, on the other hand, are serviced in terms of periodic rounds using the server-push architecture. Requests in this class arrive at the beginning of each round and must be serviced by the end of the round (i.e., all requests have the end of the round as their deadline).

To meet the requirements of requests in each class, the service manager employs a QoS-aware disk scheduling algorithm.[19] The disk scheduling algorithm exploits the characteristics of requests within each class while scheduling requests and aligns the service provided with request needs. To illustrate, since meeting request deadlines is more important to real-time clients than response times, the disk scheduler delays the scheduling of real-time requests until their deadlines and uses the available slack to service best-effort requests. Thus, by giving priority to best-effort requests whenever possible, the disk scheduling algorithm provides low response time to best-effort requests without violating deadlines of real-time requests. To isolate service classes from each other, the disk scheduler assigns fractions α_1, α_2, and α_3 ($\alpha_1 + \alpha_2 + \alpha_3 = 1$) to the periodic real-time, aperiodic real-time, and best-effort service classes, respectively,[‡] and allocates disk bandwidth to classes in proportion to these fractions. That is, the disk scheduler ensures that *at least* $\alpha_i \cdot \mathcal{R}$ duration within each round is spent in servicing requests of class i, where $\mathcal{R}$ denotes the duration of a round. If all service classes have pending requests, then each class is allocated $\alpha_i \cdot \mathcal{R}$. However, if some service class has no outstanding requests, then the unused portion of its allocation is redistributed to other classes.

To implement this disk scheduling algorithm, the service manager maintains four queues per disk: three pending queues, one for each service class, and a scheduled queue (see Figure 3(b)). The overall scheduling proceeds in terms of periodic rounds. Newly arriving requests are queued up in the appropriate pending queue. Pending requests are eventually moved to the scheduled queue, where they are guaranteed to be serviced by the end of the current round. To move a request from the pending queue to the scheduled queue, the disk scheduler must: (1) determine the insert position in the scheduled queue, and (2) determine if the service class to which the request belongs has sufficient unused allocation to insert this request.

The insert position in the scheduled queue is determined based on the following principles:

- Pending periodic and aperiodic real-time requests are inserted into the scheduled queue in SCAN-EDF order.[20] That is, real-time requests are maintained in the scheduled queue in order of increasing deadlines, and requests with same deadlines are maintained in CSCAN order. Note that, since all periodic real-time requests have the same deadline (end of the round), they can be efficiently serviced in CSCAN order.
- A best-effort request is inserted into the scheduled queue at the first position where the available slack is at least equal to the time required to service the request; a sequence of best-effort requests is always maintained in CSCAN order. Such a strategy ensures that best-effort requests are given priority over real-time requests whenever possible.

Thus, at any instant, the scheduled queue consists of a sequence of CSCAN schedules, where each schedule contains either real-time requests with the same deadline, or best-effort requests.

A pending request is inserted into the scheduled queue only if its service class has not used up its allocated duration of $\alpha_i \cdot \mathcal{R}$. To precisely formulate this criteria, let $\mathcal{U}_i$ denote the time used up by requests of class i in the current round, and let $\mathcal{I}$ denote the time for which the disk was idle (i.e., the schedule queue was empty) in the current round. To determine if a pending request can be inserted into the scheduled queue, the disk scheduler first estimates the service time of the request (defined as the sum of the seek time, rotational latency, and transfer time). Let us assume that the request is to be inserted between requests A and B in the scheduled queue, and let τ denote the service time of the request. Observe that, inserting the request between A and B will cause the service time of request B to change (since the service time of B was based on seek and rotational latency from A to B, which is no longer true after the insertion). Let τ' be the current service time of request B, and let τ'_{new} be its service time if the request is inserted. Then, the disk scheduler inserts the pending request into the scheduled queue only if: (1) the class containing the pending request has sufficient unused allocation, and (2) the total used allocation does not exceed the round duration. That is,

$$\mathcal{U}_i + \tau \leq \alpha_i \cdot (\mathcal{R} - \mathcal{I}) \tag{1}$$

and

$$\sum_{j=1}^{3} \mathcal{U}_j + \tau + (\tau'_{new} - \tau') \leq \mathcal{R} - \mathcal{I} \tag{2}$$

[‡]The fractions associated with a service class are specified at file system startup time. The service manager can also monitor the workload from each class and adapts these fractions accordingly; techniques for doing so are beyond the scope of this paper.

After inserting the request, $\mathcal{U}_i$ is incremented as $\mathcal{U}_i = \mathcal{U}_i + \tau$. Since the insertion causes the service time of B to change, the used allocation of its class is updated as $\mathcal{U}_j = \mathcal{U}_j + (\tau'_{new} - \tau')$.

Once a service class uses up its allocated duration of $\alpha_i \cdot \mathcal{R}$, then outstanding requests in its pending queue are serviced only if some other class does not use up its allocation. That is, if the disk becomes idle (the scheduled queue is empty) and a service class that has used up its allocation has outstanding requests, then the disk scheduler inserts these requests into the scheduled queue one at a time. Requests are inserted one at a time such that the disk scheduler can revert back to servicing requests based on Equations (1) and (2) as soon as a new request belonging to a class with unused allocation arrives.

5.1.2. The Storage Manager

The main objective of the storage manager is to enable the coexistence of multiple placement policies in the data type specific layer. To achieve this objective, the storage manager supports multiple block sizes and allows control over their placement on the disk array.

To allocate blocks of different sizes, the storage manager requires the minimum block size (also referred to as the *base* block size) and the maximum block size to be specified at file system creation time. These parameters define the smallest and the largest units of allocation supported by the storage manager. The storage manager can then allocate any block size within this range, provided that the requested block size is a multiple of the base block size. The storage manager constructs each requested block by allocating a sequence of contiguous base blocks on disk.

To allow control over the placement of blocks on the array, the storage manager allows *location hints* to be specified with each allocation request. A location hint consists of a (disk number, disk location) pair and denotes the preferred location for that block. The storage manager attempts to allocate a block conforming to the specified hint, but does not guarantee it. If unable to do so, the storage manager allocates a free block that is closest to the specified location. If the disk is full, or if the storage manager is unable to find a contiguous sequence of base blocks to construct the requested block, then the allocation request fails.

The ability to allocate blocks of different sizes and allow control over their placement has the following implications:

- By allowing a location hint to be specified with each allocation request, the storage manager exposes the details of the underlying storage medium (i.e., the presence of multiple disks) to the rest of the file system. This is a fundamental departure from conventional file systems which use mechanisms such as *logical volumes* to hide the presence of multiple disks from the file system. By providing an abstraction of a single large logical disk, a logical volume makes the file system oblivious of the presence of multiple disks.[12] This enables file systems built for single disks to operate without any modifications on a logical volume containing multiple disks. The disadvantage, however, is that the file system has no control over the placement of blocks on disks (since two adjacent logical blocks could be mapped by the volume manager to different locations, possibly on different disks). In contrast, by exposing the presence of multiple disks, the storage manager allows the data type specific layer precise control over the placement of blocks, albeit at the expense of having to explicitly manage multiple disks.
- The mechanisms provided by the storage manager enable any placement policy to be implemented in the data type specific layer. For instance, by appropriately generating location hints, a placement policy can stripe a file across all disks in the array, or only a subset of the disks. Similarly, location hints can be used to cluster blocks of a file on disks, thereby reducing seek and rotational latency overheads incurred in accessing these blocks. The placement policy can also tailor the block size on a per-file basis (depending on the characteristics of the data) and maximize disk throughput. However, allowing a large number of block sizes to coexist can lead to fragmentation. The storage manager attempts to minimize the effects of fragmentation by coalescing adjacent free blocks to construct larger free blocks.[21] However, such coalescing does not completely eliminate fragmentation effects, and hence, the flexibility provided by the storage manager must be used judiciously by placement policies in the data type specific layer. This can be achieved by restricting the block sizes used by these policies to a small set of values.

5.1.3. The Fault Tolerance Layer

The main objectives of the fault tolerance layer are to support data type specific reconstruction of blocks in the event of a disk failure, and to rebuild failed disks onto spare disks. To achieve these objectives, the fault-tolerance layer maintains parity information on the array. To enable data-type specific reconstruction, the fault-tolerance layer supports two mechanisms: (1) a *reliable* read, in which parity information is used to reconstruct blocks stored on the failed disk, and (2) an *unreliable* read, in which parity based reconstruction is disabled, thereby shifting the responsibility of failure recovery to the client.[17] Unlike read requests, parity computation can *not* be disabled while writing blocks, since parity information is required to rebuild failed disks onto spare disks.

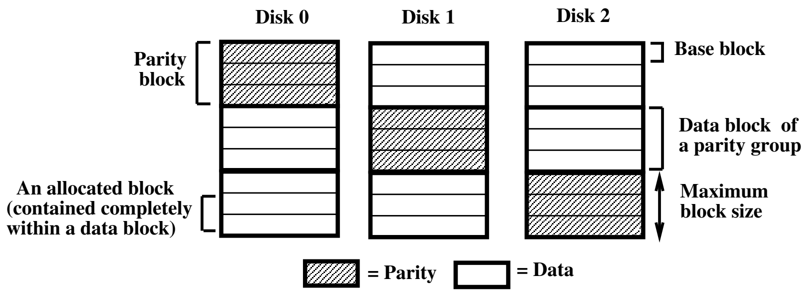


Figure 4. Parity placement in the fault tolerance layer

The key challenge in designing the fault-tolerance layer is to reconcile the presence of parity blocks with data blocks of different sizes. The fault-tolerance layer hides the presence of parity blocks on the array by exporting a *set of logical disks*, each with a smaller capacity than the original disk. The storage manager then constructs a block by allocating a sequence of contiguous base blocks on a logical disk. To minimize seek and rotational latency overheads, we require that this sequence be stored contiguously on the physical disk as well. Since the fault-tolerance layer uniformly distributes parity blocks across disks in the array (analogous to RAID-5[16]), the resulting interleaving of parity and data blocks can cause a sequence of contiguous blocks on a logical disk to be separated by intermediate parity blocks. To avoid this problem, the fault-tolerance layer uses a parity block size that is equal to the maximum block size that can be allocated by the storage manager (see Figure 4). Each data block within a parity group now contains a sequence of base blocks, all of which are contiguous on disk. By ensuring that each allocated block is contained within a data block of a parity group, the storage manager can ensure that the block is stored contiguously on disk.

5.1.4. The Meta Data Manager

The meta data manager is responsible for allocating and deallocating structures that store meta data information, and allows any data type specific structure to be assigned to files. Like in the UNIX file system,[22] meta data structures are of a fixed size and are stored on a reserved portion of the disk. Each meta data structure contains information such as the file owner, file creation time, access protection information, the block size used to store the file, the type of data stored in the file and a two level index. Level one of the index maps logical units (e.g., frames) to byte offsets, whereas level two maps byte offsets to disk block locations. This enables a file to be accessed as a sequence of logical units. Moreover, by using only the second level of the index, byte level access can also be provided to clients. Thus, by appropriately defining the logical unit of access, any data type specific structure can be assigned to a file.

Note that, the information contained in meta data structures is data type specific in nature. Hence, the meta data manager merely allocates and deallocates meta data structures; the actual meta data itself is created, interpreted, and maintained by the data type specific layer.

5.2. The Buffer Subsystem

The main objective of the buffer subsystem is to enable multiple data type specific caching policies to coexist. To achieve this objective, the buffer subsystem partitions the cache among various data types and allows each caching policy to independently manage its partition. To enhance utilization, the buffer subsystem allows the buffer space allocated to cache partitions to vary dynamically depending on the workload.

To implement such a policy, the buffer subsystem maintains two buffer pools: a pool of deallocated buffers, and a pool of cached buffers. The cache pool is further partitioned among various caching policies. Buffers within a cache partition are maintained by the corresponding caching policy in a list ordered by increasing access probabilities; the buffer that is least likely to be accessed is stored at the head of the list. To illustrate, the LRU policy maintains the least recently accessed buffer at the head of the list, while the MRU policy maintains the most recently accessed buffer at the head. For each buffer, the caching policy also computes a *time to reaccess (TTR)* metric, which is an estimate of the next time at which the buffer is likely to be accessed.[23] Since the TTR value is inversely proportional to the access probability, the buffer at the head of each list has the maximum TTR value, and TTR values decrease monotonically within a list.

On receiving a buffer allocation request, the buffer subsystem first checks if the requested block is cached, and if so, returns the requested block. In case of a cache miss, the buffer subsystem allocates a buffer from the pool of deallocated buffers and inserts this buffer into the appropriate cache partition. The caching policy that manages the partition determines the position at which the buffer must be inserted in the ordered list.

Whenever the pool of deallocated buffers falls below a low watermark, buffers are evicted from the cache and returned to the deallocated pool.[§] The buffer subsystem uses TTR values to determine which buffers are to be evicted from the cache. At each step, the buffer subsystem compares the TTR values of buffers at the head of each list and evicts the buffer with the largest TTR value. If the buffer is dirty, then the data is written out to disk before eviction. The process is repeated until the number of buffers in the deallocated pool exceeds a high watermark.

5.3. The Resource Manager

The key objective of the resource manager is to reserve system resources (i.e., disk bandwidth and buffer space) to provide performance guarantees to real-time requests. To achieve this objective, the resource manager uses: (1) a QoS negotiation protocol which allows clients to specify their resource requirements, and (2) admission control algorithms that determines if sufficient resources are available to meet the QoS requirement specified by the client.

The QoS negotiation process between the client and the resource manager uses a two phase protocol. In the first phase, the client specifies the desired QoS parameters to the resource manager. Typical QoS parameters specified are the amount of data accessed by a request, the deadline of a request and duration between successive requests. Depending on the service class of the client (i.e., periodic real-time or aperiodic real-time), the resource manager then invokes the appropriate admission control algorithm to determine if it has sufficient disk bandwidth and buffer space to service the client. The admission control algorithm returns a set of QoS parameters, which indicate the resources that are available to service the client at the current time. Depending on the available resources, the QoS parameters returned by the admission control algorithm can be less than or equal to the requested QoS. The resource manager tentatively reserves these resources for the client and returns the results of the admission control algorithm to the client. In the second phase, depending on whether the QoS parameters are acceptable to the client, it either confirms or rejects these parameters. In the former case, the tentatively reserved resources are committed for the client. In the latter case, resources are freed and the negotiation process must be restarted, either with a reduced QoS requirement, or at a later time. If the resource manager does not receive either a confirmation or rejection from the client within a specified time interval, it releases the resources that were reserved for the client. This prevents malicious or crashed clients from holding up unused resources. Once QoS negotiation is complete, the client can begin reading or writing the file. The reserved resources are freed when the client closes the file.

Depending upon the nature of the guarantees provided, admission control algorithms proposed in the literature can be classified as either deterministic or statistical.[4,7,24] Deterministic admission control algorithms make worst case assumptions about resources required to service a client and provide deterministic (i.e., hard) guarantees to clients. In contrast, statistical admission control algorithms use probabilistic estimates about resource requirements and provide only probabilistic guarantees. The key tradeoff between deterministic and statistical admission control algorithms is that the latter leads to better utilization of resources than the former at the expense of weaker guarantees. Due to space constraints, we do not discuss admission control algorithms in this paper.

6. THE DATA TYPE SPECIFIC LAYER

The data type specific layer consists of a set of modules that use the mechanisms provided by the data type independent layer to implement policies optimized for a particular data types. The layer also exports a file server interface containing methods for reading, writing, and manipulating files.[25] Each module implements a data type specific version of these methods, thereby enabling applications to create and manipulate files of that data type. In what follows, we describe data type specific modules for two data types, namely video and text.

6.1. The Video Module

The video module implements policies for placement, retrieval, meta data management, and caching of video data. Before describing these policies, let us first examine the structure of a video file. Digitization of video yields a sequence of frames. Since the size of a frame is quite large (a typical frame is 300KB in size), digitized video data is usually compressed prior to storage. Compressed video data can be multi-resolution in nature, and hence, each video file can contain one or more sub-streams. For instance, an MPEG-1 encoded video file always contains a single sub-stream, while MPEG-2 encoded files can contain multiple sub-streams.[26] Depending on the desired resolution, only a subset of of these sub-streams need to be retrieved during video playback; all sub-streams must be retrieved for playback at the highest resolution.

The video module supports video files compressed using a variety of compression algorithms. This is possible since the video module does not make any compression-specific assumptions about the structure of video files. Each file is allowed to contain any number of sub-streams, and each frame is allowed be arbitrarily partitioned among these sub-streams. Hence, a

[§] An alternative is to evict cached buffers only on demand, thereby eliminating the need for the deallocated pool. However, this can slow down the buffer allocation routine, since the buffer to be evicted may be dirty and would require a disk write before the eviction. Maintaining a small pool of deallocated buffers enables fast buffer allocation without any significant reduction in the cache hit ratio.

sub-stream can contain all the data from a particular frame, a fraction of the data, or no data from that frame. Such a file structure is general and encompasses files produced by most commonly used compression algorithms. Assuming this structure for a video file, we now describe placement, retrieval, meta data management and caching policies for video data.

6.1.1. Placement Policy

Placement of video files on disk arrays is governed by two parameters: the block size and the striping policy. The video module supports both fixed-size blocks (each of which contains a fixed number of bytes) and variable-size blocks (each of which contains a fixed number of frames). Fixed-size blocks are more suitable for environments with frequent writes and deletes, whereas variable-size blocks incur lower seek and rotational latency overheads and enable a file server to employ load balancing policies.[9] In either case, the specific block size to be used can be specified by the client at file creation time (a default value is used if the block size is unspecified). The video module then uses the interfaces exported by the storage manager to allocate blocks of the specified size. While the block size is known *a priori* for fixed-size blocks, it can change from one block to another for variable-size blocks. In the latter case, the video module determines the total size of the next f frames within a sub-stream (assuming that each variable-size block contains f frames), rounds it upwards to a multiple of the base block size, and requests a block of that size from the storage manager. Since the size of f frames may not be an exact multiple of the base block size, to prevent internal fragmentation, the video module stores some data from the next f frames in the unused space in the current variable-size block. Hence, accessing a variable-size block causes this extra data to be retrieved, which is then cached by the video module to service future read requests.

To effectively utilize the array bandwidth, the video module stripes each sub-stream across disks in the array. If sub-streams are stored on the array in terms of variable-size blocks, then the module stripes each sub-stream across all the disks in the array. On the other hand, when sub-streams are stored on the array in terms of fixed-size blocks, the striping policy depends on the array size. For small disk arrays, each sub-stream is striped across all disks in the array. However, such a policy degrades performance for large disk arrays. Hence, large arrays (consisting of few tens of disks or more) are partitioned into sub-arrays and each file is striped across a single sub-array.[13] Since the storage manager allows the disk number to be specified with the hint, such a striping policy can be easily implemented by generating appropriate location hints. The striping policy also optimizes the placement of multi-resolution video files on the array. This is achieved by storing blocks of different sub-streams that are likely to be accessed together adjacent to each other on disk.[14] Such contiguous placement significantly reduces seek and rotational latency overheads incurred during video playback. These multi-resolution optimizations can be easily implemented by generating appropriate location hints.

6.1.2. Retrieval Policy

The video module uses the interface provided by the service manager to support both periodic real-time and aperiodic real-time requests. Periodic real-time requests are supported in the sever-push mode, while aperiodic real-time requests are serviced in the client-pull mode. Since periodic real-time requests are serviced by the disk scheduler in terms of periodic rounds, such requests are issued by the video module at the beginning of each round. For each periodic real-time client, the video module generates a list of blocks to be read or written and inserts them in the periodic real-time queue of the service manager at the beginning of a round. In contrast, for aperiodic real-time clients, the video module waits for an explicit request from the client before retrieving data. Such requests arrive at arbitrary instants and are inserted on arrival into the aperiodic real-time queue of the service manager.

6.1.3. Meta data Management

Symphony allows any data type specific structure to be assigned to files. Since the logical unit of access for video is a frame, the video module allows each file to be accessed as a sequence of frames. Each file can also be accessed as a sequence of bytes to support applications that require a byte stream interface. To allow efficient random access at the byte level and the frame level, the module maintains a two level index structure. The first level of the index, referred to as the frame map, maps frame offsets to byte offsets, while the second level, referred to as the byte map, maps byte offsets to disk block locations. Figure 5 illustrates the index structure. Whereas both levels of the index are used during frame-level access, only the byte map is used during byte-level access.

The two level structure permits efficient random access for fixed-size blocks. However, supporting random access for variable-size blocks is not straightforward, since variable block sizes complicate the mapping from byte offsets to block locations in the byte map. Recall that, each variable-size block consists of a sequence of base blocks which are of fixed size. Hence, by maintaining a mapping from byte offsets to block locations for *base blocks* (instead of variable-size blocks), it is possible to support efficient random access for variable-size blocks as well. The tradeoff though is the increased storage space requirement for the byte map.

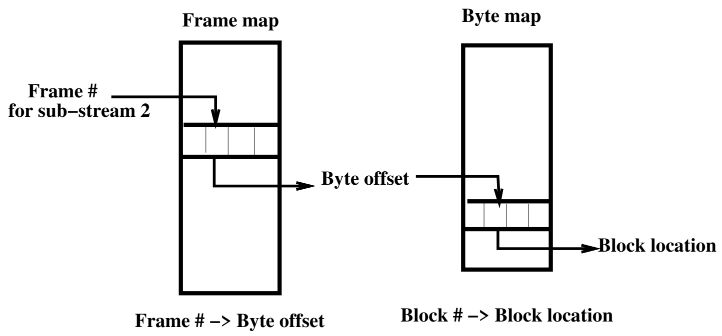


Figure 5. The index structure for the video inode. Assuming that a video file contains n sub-streams, both the frame map and the byte map contain a sequence of n-tuples. Each tuple in the frame map represents a frame, and the i^{th} field of a tuple denotes the location (i.e., byte offset) of that frame in sub-stream i. Each tuple in the byte map represents a block, and the i^{th} field of a tuple denotes the location of the block in sub-stream i.

6.1.4. Caching Policy

Since video accesses are sequential, caching policies such as LRU are ineffective for video files.[10] Recently, the Interval Caching policy was proposed for caching video blocks. The policy caches the interval between two clients accessing the same file, thereby serving requests of the trailing client from the cache. Given a fixed amount of buffer space, the policy maximizes the number of cached intervals (and hence, utilization) by caching intervals in increasing order of sizes.[18]

The video module uses the Interval Caching policy to cache video blocks. Blocks of a file that are accessed by a single client are never cached (i.e., they are returned to the deallocated pool after use). When a second client starts accessing a file, then the video module begins caching blocks being accessed by the first client in its cache partition. The trailing client must access the initial portion of the file from disk (since those blocks weren't cached). All subsequent accesses, however, are serviced from the cache.

6.2. The Text Module

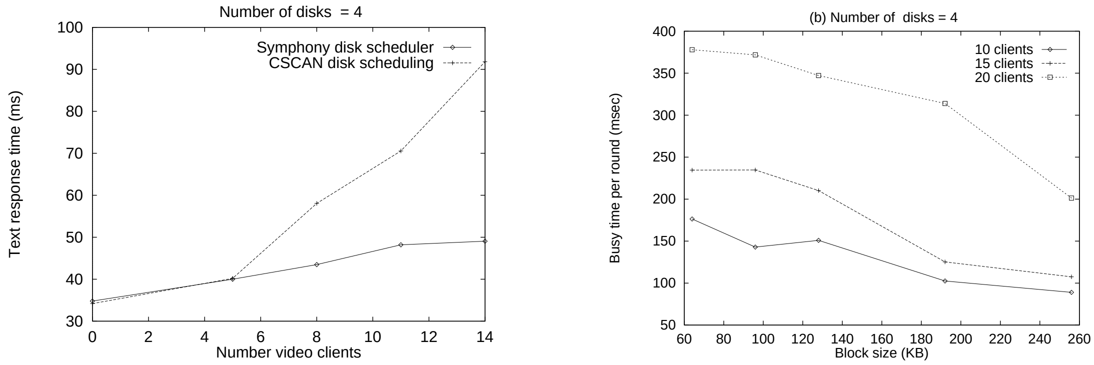
The policies implemented by the text module are very similar to those employed by conventional UNIX file systems.[27,22] For instance, the text module supports only best-effort requests, all of which are serviced in the client-pull mode. The placement policy employed by the text module supports only fixed-size blocks, and stripes successive blocks of a file onto consecutive disks in a round-robin manner. The text module supports only a byte-level access to each file. To do so, it maintains a byte map for each text file, which is similar to the UNIX inode. Finally, like in UNIX, the text module uses an LRU policy to cache text blocks.

7. EXPERIMENTAL EVALUATION OF THE SYMPHONY PROTOTYPE

We have implemented a prototype of Symphony based on the design proposed in this paper. The prototype implementation of Symphony runs as a single multi-threaded process in user space and accesses disks as raw devices. We have used the prototype to: (1) demonstrate the efficacy of the disk scheduling algorithm employed by Symphony; and (2) measure the performance of text and video clients in the fault-free state and in the presence of disk failures. The test-bed for our experiments consists of a cluster of Sun workstations connected by an ATM network. The Symphony prototype runs on a dual-processor Ultra Sparc (Model 2700) that has 128 MB of RAM and runs Solaris 2.5.1. The storage medium used for the server consists of four 2.1 GB Seagate Barracuda disks (Model ST12450W) connected to the Ultra Sparc via a fast wide SCSI interface. Symphony application programs run on a cluster of four Sparc-20 and Sparc-5 workstations, all of which run Solaris 2.5. All machines are connected to a 155 Mb/s Fore ATM switch (Model ASX-200) using ATM adapter cards. In what follows, we describe our experiments and analyze our results.

7.1. Performance of Text and Video Clients

Recall from Section 5.1.1 that, the disk scheduling algorithm delays the servicing of real-time requests until their deadlines and uses the available slack to service best-effort requests. By giving priority to text requests whenever sufficient slack is available, the disk scheduler provides low response times to text requests. To demonstrate this behavior, we compiled two versions of the prototype, one which used the scheduling algorithm described in Section 5.1.1 and the other which used the CSCAN disk scheduling algorithm. In both cases, we populated the server with a large number of text and video files. Each text file was 64KB



(a) (b)

Figure 6. (a) Response times seen by text requests. By exploiting the semantics of the data types, the Symphony disk scheduler is able to provide better response times to text requests than CSCAN. (b) Performance of video clients

in size and was striped using a block size of 4KB. Each video file was 6.5MB in size and contained 1000 MPEG-1 compressed frames, striped using a block size of 64KB.[¶] The playback rate for the each video file was 30 frames/s and the average bit rate was 1.5 Mb/s. We assigned fractions of $\alpha_1 = 0.6, \alpha_2 = 0.05$, and $\alpha_3 = 0.35$ to the periodic real-time, the aperiodic real-time and best-effort classes, respectively, in the disk scheduler. The duration of a round was set to 1 second. For both versions of the prototype, we varied the number of video clients and measured the response time seen by text requests. Each video client in our experiments was a modified version of `mpeg_play` and retrieved a randomly selected file in the periodic real-time mode. Each text client read a randomly selected text file in sequential order using 8KB requests. Figure 6(a) plots the average response time for a 8KB request observed in the two cases. The figure shows that the Symphony disk scheduler provides better response times to text requests than CSCAN. This is because, CSCAN services requests in the order of their disk cylinder numbers. Since it interleaves text and video requests based on this ordering, the response times seen by text requests increases with increase in number of video requests. In contrast, the Symphony disk scheduler always gives priority to text requests regardless of the number of video requests (provided sufficient slack is available). Moreover, unused allocation of the periodic real-time class is reassigned to the best-effort class. This results in better response times to text requests, and the response time degrades only slightly with increase in number of video clients.

To demonstrate that the improvement in the response time for text requests did not come at the expense of deadline violations for video requests, we repeated the above experiment by varying the block size used for each video file, and measured the service time of disks (i.e., the duration for which a disk was busy in each round). Figure 6(b) depicts the service time of a disk for different number of video clients and different block sizes. It shows that the service time of the disk is within 600 ms, which is the duration of each round allocated to periodic real-time requests (since $\alpha_1 = 0.6$ and the round duration is 1s). Hence, the disk scheduler is able to meet the real-time requirement of video clients.

7.2. Performance in the presence of disk failure

The fault-tolerance layer allows multiple fault tolerance policies to coexist within the file system. Specifically, it allows clients to enable or disable parity-based reconstruction (using reliable or unreliable reads) for recovering data. Since entire parity group must be retrieved to reconstruct a block requested from the failed disk, parity based reconstruction imposes a large (100%) overhead on the server[15]; no such overhead is imposed when parity-based reconstruction is disabled. To demonstrate this fact, we configured the prototype to assume that one of the disks in the array had failed. We varied the number of video clients and measured the load on the server (in terms of the service time of a disk) with parity-based reconstruction enabled. Next, we repeated the experiment with parity-based reconstruction disabled. As expected, the service time of disk with parity-based reconstruction enabled was twice of that with parity-based reconstruction disabled. Thus, disabling parity-based reconstruction significantly reduces the load on the server. The tradeoff though is that this option requires sophisticated clients that can exploit redundancies in video data to approximately reconstruct lost data. Also, approximate reconstruction causes a degradation in image quality. However, studies have shown that the degraded quality is within human perceptual tolerances for many commonly used compression algorithms (e.g., JPEG and MPEG).[17]

8. RELATED WORK

Several techniques for the storage and retrieval of continuous media that have been proposed in the literature.[2–6,28] Symphony builds upon these techniques and integrates them into a general purpose file system. For instance, the interval caching policy

[¶]The data for the video files was obtained by digitizing several television sitcoms, newscasts and sports events.

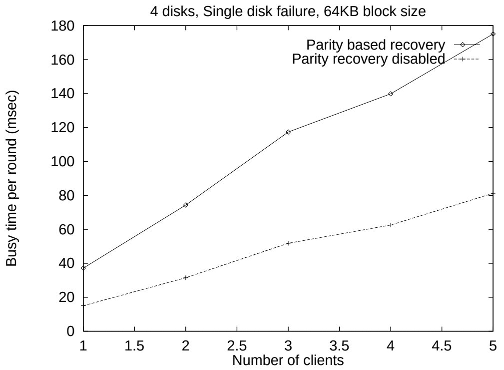


Figure 7. Reliable reads versus unreliable reads. The figure shows that disabling parity-based reconstruction significantly lowers the load on the server.

employed by the video module was proposed by Dan et. al.[18] The efficacy of using variable-size blocks for storing video files has been demonstrated by several studies.[3,29] Techniques for approximate reconstruction of image and video data (e.g., JPEG and MPEG) have been proposed by Vin et. al.[17] and Danskin et. al.[30] Admission control algorithms have been discussed by Rangan et. al.[4] and Vin et. al.[7]

Several recent and ongoing research efforts have focussed on building integrated file systems. For instance, Fellini[31] and CMFS[2] are file systems that can handle both real-time continuous media data and non-real time textual data. Both are single disk file systems and do not employ multi-disk optimizations such as striping. Similarly, MMFS is a single disk file system that adds continuous media support to a FreeBSD-based file system.[32] The Tiger Shark file system from IBM and XFS from SGI are results of commercial efforts to build integrated file systems.[33,12] These file systems come closest to Symphony in terms of the features offered. For instance, these file systems support variable-size blocks (referred to as *extents*), guaranteed rate I/O, etc. However, they do not support features such as a QoS-aware disk scheduler, data type specific placement, failure recovery, and caching policies. Moreover, they statically partition the storage space available on the disk array using logical volumes. Logical volumes can either span a mutually exclusive set of disks, or share the same set of disks by statically partitioning the space on each disk. In the former case, both storage space and disk bandwidth get statically partitioned among logical volumes, while in the latter case only the storage space is statically partitioned and the disk bandwidth is dynamically shared by logical volumes. In contrast, Symphony is a physically integrated file system, in which both storage space and disk bandwidth are dynamically shared among data types.

Finally, the logical disk abstraction[34] provides an interface that allows multiple *file systems* to coexist on a single storage device. Logical disks provide functionalities similar to those provided by the data type independent layer of Symphony, such as multiple block sizes, location hints, etc. However, a key difference between logical disks and Symphony is that the former does not differentiate between request types, and consequently provides only a best-effort service model. In contrast, Symphony employs multiple service classes that enable it to efficiently support real-time continuous media requests as well as non-real time requests.

9. CONCLUDING REMARKS

In this paper, we discussed various architectural considerations in designing an integrated multimedia file system and their tradeoffs. We argued that, to efficiently support storage and retrieval of heterogeneous data types, integrated file systems should enable the co-existence of multiple data type specific policies. We then presented the design of Symphony–an integrated multimedia file system that achieves this objective. The architecture of Symphony consists of two layers. The lower layer provides a set of data type independent mechanisms that implement core file system functionality. The upper layer uses these mechanisms to implement data type specific policies, and exports a file server interface containing methods for reading, writing, and manipulating files. Some of the novel features of Symphony include: support for multiple service classes; a QoS-aware disk scheduling algorithm; and support for data type specific placement, failure recovery, meta data management, and caching techniques.

As part of future work, we plan to add support for new data types such as audio and multi-resolution images. We also plan to add support for edit operations and implement features such as copy-on-write that allow fast copying of large multimedia files. We expect that our two-layer architecture will simplify the task of adding these features to Symphony.

ACKNOWLEDGMENTS

Renu Tewari contributed to the design and implementation of the buffer subsystem. We thank Mike Dahlin and Renu Tewari for their comments on earlier drafts of the paper.

REFERENCES

1. M. K. McKusick, W. N. Joy, S. J. Leffler, and R. S. Fabry, “A fast file system for UNIX,” *ACM Transactions on Computer Systems* **2**(3), pp. 181–197, August 1984.
2. D. Anderson, Y. Osawa, and R. Govindan, “A file system for continuous media,” *ACM Transactions on Computer Systems* **10**, pp. 311–337, Nov. 1992.
3. P. W. Jardetzky, *Network File Server Design for Continuous Media*. PhD thesis, University of Cambridge, August 1992.
4. P. V. Rangan and H. Vin, “Designing file systems for digital video and audio,” in *Proceedings of the 13th Symposium on Operating Systems Principles (SOSP’91), Operating Systems Review, Vol. 25, No. 5*, pp. 81–94, Oct. 1991.
5. F. Tobagi, J. Pang, R. Baird, and M. Gang, “Streaming RAID: A disk storage system for video and audio files,” in *Proceedings of ACM Multimedia’93, Anaheim, CA*, pp. 393–400, August 1993.
6. M. Vernick, C. Venkatramini, and T. Chiueh, “Adventures in building the Stony Brook video server,” in *Proceedings of ACM Multimedia’96*, 1996.
7. H. M. Vin, P. Goyal, A. Goyal, and A. Goyal, “A statistical admission control algorithm for multimedia servers,” in *Proceedings of the ACM Multimedia’94, San Francisco*, pp. 33–40, October 1994.
8. S.S.Rao, H.M.Vin, and A. Tarafdar, “Comparative evaluation of server-push and client-pull architectures for multimedia servers,” in *Proceedings of NOSSDAV’96*, pp. 45–48, April 1996.
9. H. Vin, S. Rao, and P. Goyal, “Optimizing the placement of multimedia objects on disk arrays,” in *Proceedings of the Second IEEE International Conference on Multimedia Computing and Systems, Washington, D.C.*, pp. 158–165, May 1995.
10. P. Cao, *Application Controlled File Caching and Prefetching*. PhD thesis, Princeton University, 1996.
11. R. H. Patterson and et. al., “Informed prefetching and caching,” in *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.
12. M. Holton and R. Das, “XFS: A next generation journalled 64-bit file system with guaranteed rate i/o,” tech. rep., Silicon Graphics, Inc, Available online as http://www.sgi.com/Technology/xfs-whitepaper.html.
13. P. J. Shenoy and H. M. Vin, “Efficient striping techniques for multimedia file servers,” in *Proceedings of NOSSDAV’97, St. Loius, MO*, pp. 25–36, May 1997.
14. P. J. Shenoy and H. M. Vin, “Efficient support for scan operations in video servers,” Tech. Rep. TR96-35, Department of Computer Sciences, Univ. of Texas at Austin, December 1996.
15. P. M. Chen, E. K. Lee, G. A. Gibson, R. H. Katz, and D. A. Patterson, “Raid: High-performance, reliable secondary storage,” *ACM Computing Surveys* , pp. 145–185, June 1994.
16. D. Patterson, G. Gibson, and R. Katz, “A case for redundant array of inexpensive disks (RAID),” *ACM SIGMOD’88* , pp. 109–116, June 1988.
17. H. M. Vin, P. J. Shenoy, and S. Rao, “Efficient failure recovery in multi-disk multimedia servers,” in *Proceedings of the 25th International Symposium on Fault Tolerant Computing Systems, Pasadena, CA*, pp. 12–21, June 1995.
18. A. Dan and D. Sitaram, “A generalized interval caching policy for mixed interactive and long video workloads,” in *Proceedings of Multimedia Computing and Networking (MMCN) Conference*, pp. 344–351, 1996.
19. P. J. Shenoy and H. M. Vin, “Cello: A disk scheduling framework for next generation operating systems,” tech. rep., Department of Computer Sciences, Univ. of Texas at Austin, 1997. http://www.cs.utexas.edu/users/dmcl.
20. A. N. Reddy and J. Wyllie, “Disk scheduling in multimedia I/O system,” in *Proceedings of ACM Multimedia’93, Anaheim, CA*, pp. 225–234, August 1993.
21. D. E. Knuth, *The Art of Computer Programming Volume 1: Fundamental Algorithms*, Addison Wesley, 1973.
22. S. J. Leffler, M. K. McKusick, M. J. Karels, and J. S. Quartermann, *The Design and Implementation of the 4.3BSD Unix Operating System*, Addison Wesley, 1989.
23. R. Tewari, H. M. Vin, A. Dan, and D. Sitaram, “Caching in bandwidth and space constrained hierarchical hyper-media servers,” Tech. Rep. TR96-30, Department of Computer Sciences, Univ. of Texas at Austin, December 1996.
24. H. M. Vin, A. Goyal, and P. Goyal, “Algorithms for designing large-scale multimedia servers,” *Computer Communications* **18**, pp. 192–203, March 1995.
25. P. J. Shenoy, P. Goyal, S. Rao, and H. M. Vin, “Design and implementation of Symphony: An integrated multimedia file system,” Tech. Rep. TR97-09, Dept. of Computer Sciences, Univ, of Texas at Austin, March 1997.
26. International Organisation for Standardisation, *Information Technology - Generic Coding of Moving Pictures and Associated Audio Systems: Systems, Video and Audio, International Standard (MPEG2), ISO/IEC 13818,*, November 1994.
27. M. J. Bach, *The Design of the Unix Operating System*, Prentice Hall, 1986.

28. M. Buddhikot, G. Parulkar, and J. Cox, "Design of a large scale multimedia storage server," *Journal of Computer Networks and ISDN Systems* , pp. 504—524, Dec 1994.
29. E. Chang and A. Zakhor, "Scalable video placement on parallel disk arrays," in *Proceedings of IS&T/SPIE International Symposium on Electronic Imaging: Science and Technology, San Jose*, February 1994.
30. J. M. Danskin, G. M. Davies, and X. Song, "Fast lossy internet image transmission," in *Proceedings of the Third ACM Conference on Multimedia, San Francisco, California*, pp. 321–332, November 1995.
31. C. Martin, P. S. Narayan, B. Ozden, R. Rastogi, and A. Silberschatz, "The Fellini multimedia storage server," *Multimedia Information Storage and Management , Editor S. M. Chung, Kluwer Academic Publishers* , 1996.
32. T. Niranjan, T. Chiueh, and G. Schloss, "Implementation and evaluation of a multimedia file system," in *Proceedings of ICMCS'97, Ottawa, Canada*, 1997.
33. R. Haskin and F. Schmuck, "The Tiger Shark file system," *Proceedings of COMPCON* , Spring 1996.
34. W. Jonge, M. F. Kaashoek, and W. C. Hsieh, "The logical disk: A new approach to improving file systems," in *Proceedings of the Fourteenth Symposium on Operating Systems Principles*, 1993.