Design Considerations for the Symphony Integrated Multimedia File System*

Prashant Shenoy
Department of Computer Science,
University of Massachusetts,
Amherst, MA 01003
shenoy@cs.umass.edu

Pawan Goyal†
†Ensim Corporation
1215 Terra Bella Ave,
Mountain View, CA 94043
goyal@ensim.com

Sriram Rao‡ Harrick M. Vin‡
‡Department of Computer Sciences,
University of Texas,
Austin, TX 78712
{sriram,vin}@cs.utexas.edu

Abstract

A multimedia file system supports diverse application classes that access data with heterogeneous characteristics. In this paper, we describe our experiences in the design and implementation of the Symphony multimedia file system. We first discuss various methodologies for designing multimedia file systems and examine their tradeoffs. We examine the design requirements for such file systems and argue that, to efficiently manage the heterogeneity in application requirements and data characteristics, a multimedia file system should enable the coexistence of multiple data type-specific and application-specific techniques. We describe the architecture and novel features of Symphony and then demonstrate their efficacy through an experimental evaluation. Our results show that Symphony yields a factor of 1.9 improvement in text response time over conventional disk scheduling algorithms, while continuing to meet the real-time requirements of video clients. Finally, we reflect upon the lessons learned from the Symphony project.

1 Introduction

1.1 Motivation

Recent advances in compression, storage, and communication technologies have resulted in a proliferation of diverse multimedia applications such as distance education, online virtual worlds, immersive telepresence, and scientific visualization. A common characteristic of these applications is that impose widely differing requirements and access vast amounts of data such as text, audio, video, and images (collectively referred to as *multimedia*). Realizing such next-generation applications requires file systems that can support the following features:

- *Heterogeneity in application requirements:* General-purpose file systems must support applications with diverse performance requirements. For instance, interactive applications, such as word-processors and compilers, desire low average response times but no absolute performance guarantees. Throughput-intensive applications, such as ftp and http servers, desire high aggregate throughput and are less concerned about the response times of individual requests. Soft real-time applications, such as video players, require performance guarantees from the file system (e.g., bounds on response times) but can tolerate occasional violations of these guarantees.

*A preliminary version of this paper appeared in the Proceedings of ACM/SPIE Multimedia Computing and Networking Conference (MMCN'98), San Jose, CA, January 1998.

- *Heterogeneity in data characteristics:* Data types differ significantly in their characteristics such as form, size, data rate, real-time requirements, etc. For instance, typical textual objects are a few kilobytes in size; accesses to these objects are aperiodic and best-effort in nature. In contrast, audio and video (*continuous media*) objects can be several gigabytes in size; accesses to these objects are periodic and sequential, and impose real-time constraints. A general-purpose file system must reconcile the storage and retrieval of data types with such diverse requirements.
- *Extensibility:* Since it is difficult, if not impossible, to foresee requirements imposed by future applications and data types, next generation file systems should employ techniques that facilitate easy integration of new application classes and data types. This requires the file system to employ an extensible architecture so that file system services can be easily tailored to meet new requirements.

Existing file systems fail to meet one or more of the above requirements. To illustrate, conventional file systems have been optimized for the storage and retrieval of textual data and provide a single class of best-effort service to all applications regardless of their requirements [23, 31]. Consequently, these file systems are ill-suited for meeting the real-time performance requirements of continuous media data. To overcome these limitations, continuous media servers that are optimized for real-time audio and video have been designed [1, 16, 28, 40, 41]. These file 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. However, they are unsuitable for managing best-effort textual and image data. Finally, none of the existing file systems are designed to be inherently extensible. Thus, existing file systems are inadequate for managing the heterogeneity in application requirements and data characteristics. The design and implementation of a general-purpose multimedia file system that overcomes these limitations is the subject matter of this paper.

1.2 Research Contributions

In this paper, we first examine two different architectures for designing multimedia file systems, namely partitioned and integrated. We examine the tradeoffs of these architectures and argue that the integrated architecture yields better resource utilization and performance than a partitioned system. Hence, we choose the integrated architecture for designing our multimedia file system. Next, we discuss the design requirements imposed on integrated multimedia file systems with respect to (i) the service model, (ii) the data retrieval paradigm, and (iii) techniques for placement, fault tolerance, caching, and meta data management. These requirements indicate that, to efficiently manage heterogeneity along several dimensions, an integrated file system should enable the coexistence of multiple data type-specific and application-specific techniques.

We then present the design and implementation of *Symphony*–an integrated file system that meets these requirements. Integrating diverse techniques into a multimedia file system is a key challenge, because it complicates the file system architecture. Symphony addresses this challenge by employing a novel two layer architecture. The lower layer of Symphony implements a set of data type independent mechanisms that provide core file system functionality. The upper layer uses these mechanisms to implement multiple data type-specific and application-specific policies. The two levels of the architecture separate data-type independent mechanisms from data type-specific and application-specific policies and thereby facilitate easy extensions to the file system.

In addition to the two layer architecture, Symphony consists of a number of other novel features. These include: (1) the Cello disk scheduling framework that provides predictable performance to disk requests with diverse requirements, (2) a storage manager that supports multiple block sizes and allows complete control over their placement, (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. Symphony also 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 [43], (2) support for client-pull and server-push architectures [38], (3) support for fixed-size and variable-size blocks [45], and (4) support for data type specific caching techniques [5, 11].

We describe the prototype implementation of Symphony and presents the results of our experimental evaluation. Finally, we reflect on the lessons learned from the Symphony project.

The rest of the paper is organized as follows. In Section 2, we examine architectures for designing multimedia file systems. The design requirements imposed on an integrated multimedia file system are outlined in Section 3. Sections 4, 5, and 6 describe the architecture and design of Symphony. In Section 7, we experimentally evaluate the Symphony prototype. Section 8 describes the lessons learned from the design and implementation of Symphony. Section 9 describes related work, and finally, Section 10 summarizes our results.

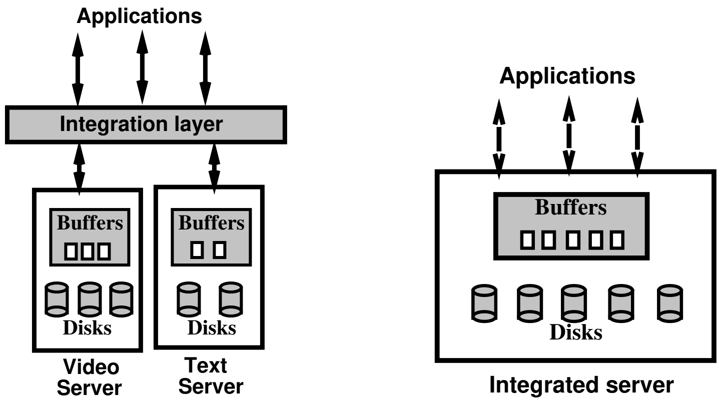
2 Design Methodologies

There are two methodologies for designing multimedia file systems: (1) *partitioned* file systems that 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) *integrated* file systems that 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 [23, 41], partitioned file systems are easy to design and implement. In such file systems, resources (disks, buffers) are statically partitioned among component file servers. This causes requests accessing different component servers to access mutually exclusive set of resources, 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 resources is governed by the expected workload on each component server. If the observed workload deviates significantly from the expected, then repartitioning of resources may be necessary. Repartitioning of resources such as disks and buffers is tedious and may require the system to be taken offline [14]. An alternative to repartitioning is to add new resources (e.g., 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 a 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.

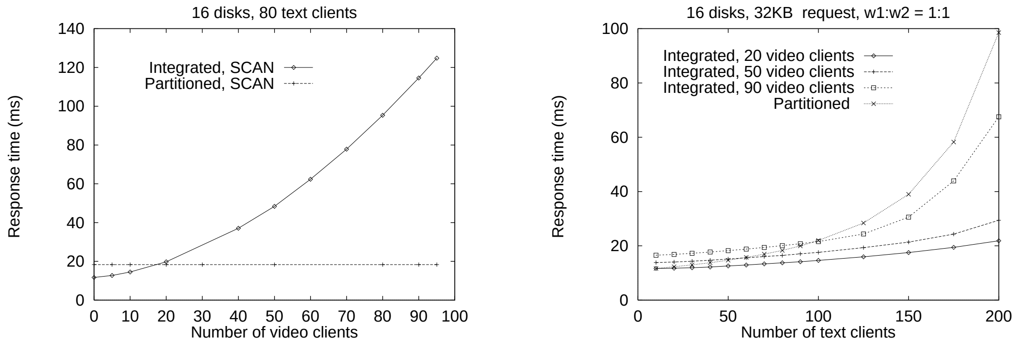


(a) Partitioned file server (b) Integrated file server

Figure 1: Partitioned and integrated file servers supporting text and video applications. The partitioned architecture divides the server resources among multiple component file systems, and employs an integration layer that provides a uniform mechanism to access files. The integrated architecture employs a single server that multiplexes all the resources among multiple application classes.

In contrast, 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, disk bandwidth, and buffer space 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 resources are shared by all applications, more resources are available to service each request, which in turn improves the performance. A disadvantage, though, of 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, requests 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.

We have recently conducted a quantitative study to examine the tradeoffs of the two architectures [32]. Our results showed that: (i) an integrated server outperforms the partitioned server in a large operating region and has slightly worse performance in the remaining region, (ii) the capacity of an integrated server is larger than that of the partitioned server, and (iii) an integrated server outperforms the partitioned server by up to a factor of 6 in the presence of bursty workloads. To provide the intuition for these results, consider the following experiment reported in our study [32]. Consider a file server consisting of sixteen disks. Let eight disks each be allocated to the text and video servers in the partitioned architecture, while all disks are shared by all applications in the integrated architecture. Let the text server employ the SCAN disk scheduling algorithm to service user requests. As shown in Figure 2(a), use of SCAN in the integrated server significantly degrades response times of text requests. This is because SCAN does not differentiate between requests with different requirements and schedules requests merely based on their cylinder numbers. The resulting interleaving of text and video requests causes video requests to interfere with text requests, thereby degrading response times of the latter. The performance of the integrated



(a) SCAN (b) Cello disk scheduling algorithm

Figure 2: Response times in partitioned and integrated file systems. Figure (a) shows that use of a naive disk scheduling algorithms such as SCAN to schedule a mix of text and video requests can degrade response time of text requests in the integrated server. In contrast, a more sophisticated disk scheduler such as Cello yields response times that are comparable or better than that in the partitioned server.

server improves significantly by employing a more sophisticated disk scheduling algorithm such as Cello [33] (see Figure 2(b)). The Cello disk scheduling algorithm employs slack stealing to service text requests before video requests whenever possible. By delaying the servicing of real-time requests until their deadlines, Cello provides low average response times to text requests without affecting the deadlines of video requests. Moreover, by employing all disks to service all applications, Cello can reallocate unused bandwidth from one request class to another, further improving performance. Thus, by carefully designing resource management algorithms, an integrated server can provide identical or better performance as compared to a partitioned server (see [33] for a detailed exposition of these issues).

To summarize, integrated file servers obviate the need for static partitioning of resources inherent in partitioned servers. Since the performance gains due to statistical multiplexing of resources outweigh the drawbacks of increased complexity, we choose the integrated architecture for designing Symphony. Next we discuss the design requirements imposed by applications and data types on an integrated multimedia file system.

3 Design Requirements for an Integrated File System

An integrated file system 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.

3.1 Service Model

Most existing file systems provide a single class of service to all applications, regardless of their requirements. The UNIX file system, for instance, provides a best-effort service to all applications. If one could cost-effectively over-provision such file systems such that the offered load is always significantly smaller than the capacity, then a single class of service may be adequate to meet heterogeneous application requirements. However, modern file system workloads exhibit significant variability at multiple time scales [12]. In such a scenario, over-provisioning for the worst case usage would require substantial amount of resources and is not cost-effective. Regardless of the amount of over-provisioning, a file system can increase the overall utility to applications by *supporting multiple classes of service.* This enables applications to use the service class that is most suited to their needs and simplifies application development. Instantiating multiple service classes requires the file system to employ (i) a disk scheduling algorithm that supports these classes and aligns the service provided with application needs, and (ii) resource reservation algorithms that provide performance guarantees to real-time applications.

3.2 Retrieval Architecture

Most conventional file systems employ the *client-pull* mode of retrieval, in which the server retrieves information from disks only in response to an explicit read request from a client.[1] Whereas the client-pull mode is suitable for textual applications, adapting continuous media applications for client-pull accesses is difficult. This is because maintaining continuity in continuous media playback requires that retrieval requests be issued sufficiently in advance of the playback instant. To do so, applications must estimate the response time of the server and issue requests appropriately. Since the response time varies dynamically depending on the server and the network load, client-pull based continuous media applications are non-trivial to develop [38]. Hence, most continuous media file servers employ the *server-push* (or *streaming*) mode of retrieval, in which the server periodically retrieves and transmits data to clients without explicit read requests. The server-push mode of retrieval is inappropriate for aperiodic text requests. Hence, to efficiently support multiple application classes, an integrated file system should support both the client-pull and the server-push retrieval modes.

3.3 Placement Techniques

Due to the large storage space and bandwidth requirements of multimedia data, integrated file systems use disk arrays as their underlying storage medium. 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*[2], which is the maximum amount of logically contiguous data stored on a single disk, and (2) the *the degree of striping*, which denotes the number of disks across which a file is striped. The characteristics of data stored on the array has a significant impact on both parameters. To illustrate, the large bandwidth requirements of real-time continuous media yields an optimal stripe unit size that is an order of magnitude larger than that for text [37]. 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, a policy that stripes each file across all disks in the array is suitable for textual data, but yields suboptimal performance for variable bit rate continuous media [37]. Moreover, since continuous media files can have a multi-resolution nature (e.g, MPEG-2 encoded video), an integrated file system can optimize the placement of such files by storing blocks belonging to different resolutions adjacent to each other on disk [34]. Such contiguous placement

[1] Observe that, although such servers generally employ some prefetching and caching techniques to improve performance, due to the aperiodic nature of accesses, requests for retrieving data from disks are triggered only in response to explicit access requests from the client.

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

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 placement policies and mechanisms to enable their coexistence.

3.4 Failure Recovery Techniques

Since disk arrays are highly susceptible to disk failures, a file server should employ failure recovery techniques to provide uninterrupted service to clients in the presence of failures. 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 [7]. 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, on-line failure recovery techniques that exploit the spatial and temporal redundancies present in continuous media to reconstruct a reasonable approximation of the lost data have been developed [44]. Unlike mirroring and parity based techniques, these techniques do not require any additional data to be accessed for failure recovery, and thereby significantly reduce the recovery overhead. Hence, to enhance system utilization, an integrated file system should support multiple data type specific failure recovery techniques and mechanisms that enable their coexistence.

3.5 Caching Techniques

File systems employ memory caches to improve application performance. To efficiently manage these caches, conventional textual file systems employ cache replacement policies such as LRU. It is well known that LRU performs poorly for sequential data accesses [5], and hence, is inappropriate for continuous media. Policies that are tailored for sequential data accesses, such as Interval Caching [8], are more desirable for continuous media, but are unsuitable for textual data. Since an integrated file system must support applications with different access characteristics, use of a single cache replacement policy for all applications can degrade cache hit ratio. Consequently, to enhance utilization of the cache, an integrated file system should support multiple data type specific caching policies, as well as mechanisms that enable these policies to efficiently share the cache.

3.6 Meta Data Management

Most conventional file systems allow files to be accessed as a sequence of bytes (i.e., they do not assign any structure to files). For continuous media files, however, the logical unit of access is a video frame or an audio sample. Accessing such files in conventional file systems requires the application to separately maintain logical unit indices. Developing such applications would be simplified if the file system were to allow files to be accessed as a sequence of logical units. 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 should 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.

3.7 Extensibility

The past few years have seen the emergence of a new generation of applications with diverse requirements and data type with heterogeneous characteristics. Rapid advances in computing technologies are likely to accelerate this trend, leading to a proliferation of new applications and data types. An integrated file system should efficiently support both present and future applications and data types. Since it is difficult, if not impossible, to foresee requirements that will be imposed by future applications and data types, an integrated file system will need to employ an extensible architecture. Such an architecture will facilitate easy extensions to the file systems, thereby simplifying the integration of new application classes and data types into the file system.

3.8 Requirements Summary

The preceding discussion demonstrates that an integrated file system differs from existing file systems in several fundamental ways. Managing heterogeneity in application requirements and data characteristics is key to an integrated file system. To achieve this objective, an integrated file system should: (i) support multiple classes of service, (ii) support both the client-pull and server-push modes of retrieval, (iii) employ data-type specific and application-specific policies for placement, caching, failure recovery and meta data management, (iv) employ data-type independent mechanisms that enable the coexistence of these policies, and (v) employ an extensible architecture. In what follows, we describe the design and implementation of *Symphony*–an integrated file system that meets these requirements.

4 Architecture of Symphony

Symphony meets the design requirements outlined in Section 3 by employing a novel 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 policies for placement, failure recovery, caching, etc. The layer also exports a file server interface containing methods for reading, writing, and manipulating files. Figure 3 depicts this architecture. By cleanly separating data type independent mechanisms from data type specific policies, the two layer architecture of Symphony enables the coexistence of multiple policies. Consequently, file system designers can easily extend the file system functionality by adding policies (modules) for new data types or modify policies for existing data types. Next, we describe the two layers of Symphony 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).

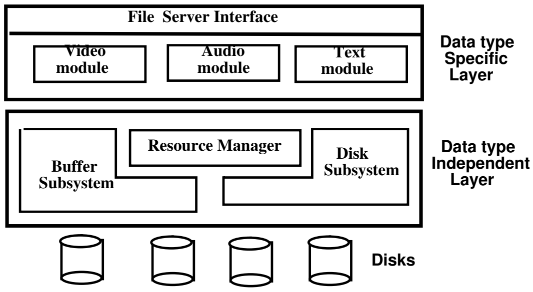
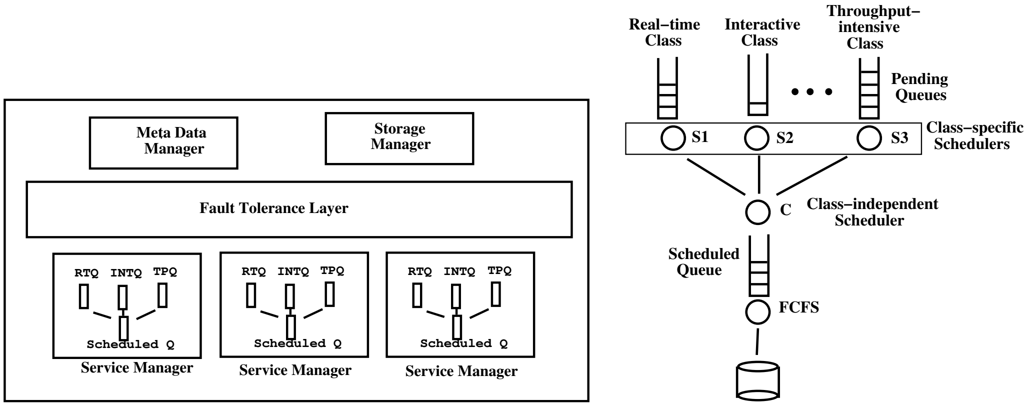


Figure 3: Two layer architecture of Symphony



(a) Disk Subsystem Architecture (b) Cello Disk Scheduler

Figure 4: The architecture of the disk subsystem and the Cello disk scheduling framework.

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 4(a)): (1) a *service manager* that supports mechanisms for efficient scheduling of interactive, throughput-intensive, and 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: interactive best-effort, throughput-intensive best-effort, and soft real-time. Interactive and throughput-intensive requests are assumed to arrive at arbitrary instants and are serviced using the client-pull architecture. Real-time requests can be periodic or aperiodic. Whereas aperiodic requests are serviced using the client-pull architecture, periodic requests are serviced in terms of periodic rounds using the server-push architecture. These requests 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).

Symphony employs the Cello disk scheduling framework to meet the requirements of requests in each class [33]. Cello employs a two level scheduling architecture, consisting of a class independent scheduler and a set of class specific schedulers (see Figure 4(b)). Together, the two levels of the scheduler allocate disk bandwidth at two time scales. The class-independent scheduler governs the coarse-grain bandwidth allocation to application classes, while the class-specific schedulers control the fine-grain interleaving of requests from the application classes. Moreover, to ensure predictable allocation of bandwidth, Cello assigns a weight to each application class and allocates bandwidth to classes in proportion to their weights.

Cello maintains four queues—three pending queues, one per application class and a scheduled queue. Newly arriving requests are queued up in the appropriate pending queue. These requests are eventually moved to the scheduled queue, from where they are dispatched to the disk for service. Conceptually, the class-independent scheduler determines *when* and *how many* requests are moved from each pending queue to the scheduled queue, whereas the class-specific schedulers determine *where* to insert these requests in the scheduled queue. Cello provides performance guarantees to application classes at the granularity of *intervals*. Within each interval, disk bandwidth is allocated to classes in proportion to their weights. To ensure this criteria, the class independent scheduler moves a request r from the pending queue to the scheduled queue only if the following two conditions are satisfied: (1) the class to which the pending request belongs has sufficient unused allocation, and (2) the total allocation used up by all classes does not exceed the interval duration [33].

The class specific schedules employ the following policies to determine where to insert a pending request in the scheduled queue:

- *Interactive requests:* Since interactive requests desire low average response times, the class-specific scheduler for these requests employs the classic *slack stealing* technique [21]. By inserting interactive requests before real-time requests whenever possible (i.e., whenever slack is available), the scheduler ensures low average response times to these requests without violating deadlines of real-time requests. Moreover, a sequence of interactive requests is maintained in SCAN order to reduce disk seek and rotational latency overheads.

- *Throughput-intensive requests:* Since these requests desire high aggregate throughput, but are less concerned about response times of individual requests, the class-specific scheduler inserts these requests at the tail of the scheduled queue in SCAN order. Doing so, enables Cello to minimize response times or deadline violations for other classes without affecting the throughput received by applications in this class.

- *Real-time requests:* Real-time requests are inserted into the scheduled queue in SCAN-EDF order [29]. That it, requests are inserted in increasing order of deadlines (i.e., EDF order) and requests with identical deadlines are inserted in SCAN order.

To efficiently utilize disk bandwidth, Cello reallocates unused disk bandwidth from classes with no pending requests to classes that have used up their allocations but still have pending requests. This ensures that, regardless of the weights, the disk will not idle so long as there are requests waiting to be serviced. Thus, Cello scheduler is work-conserving in nature. Our experimental results show that Cello (i) aligns the service provided with the application requirements, (ii) protects application classes from one another, (iii) is work-conserving and can adapt to changes in work-load, (iv) minimizes the seek time and rotational latency overhead incurred during accesses, and (v) is computationally efficient [33].

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.[3]

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. 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 [14]. This enables file systems built for single disks to operate without any modifications on a logical volume containing multiple disks. The disadvantage, though, 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. The drawback, though, of allowing a large number of block sizes to coexist is that it can lead to fragmentation. The storage manager attempts to minimize fragmentation effects

[3] Note that, the notion of a block in Symphony is different from that of an *extent* in UNIX file systems [24]. Both extents and blocks are constructed using a sequence of contiguous base blocks. Unlike extents, however, Symphony does not permit access to individual base blocks within a block; the entire block must always be accessed to access any portion of it.

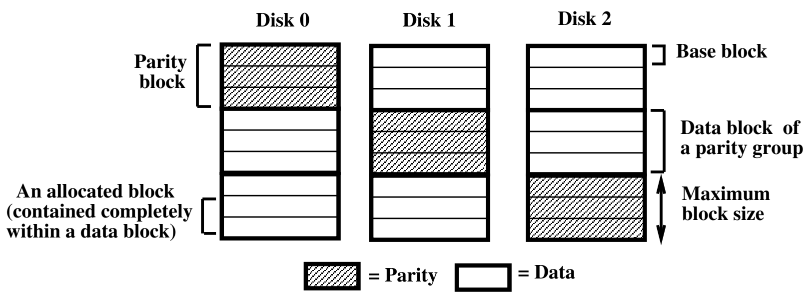


Figure 5: Parity placement in the fault tolerance layer

by employing *free block coalescing*—a techniques that coalesces adjacent free blocks to construct larger free blocks [19]. While such coalescing significantly reduces disk fragmentation, it does not completely eliminate it. Hence, the flexibility provided by the storage manager must be used judiciously by placement policies instantiated in the data type specific layer—for instance, 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 [44]. Unlike read requests, parity computation can *not* be disabled while writing blocks, since parity information is required to rebuild failed disks onto spare disks.

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 [27]), 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 5). 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 [20], meta data structures

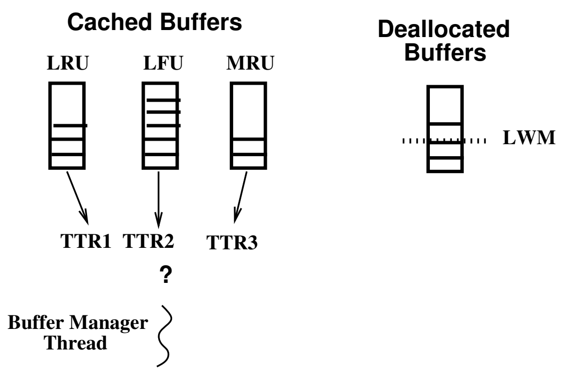


Figure 6: The buffer subsystem of Symphony

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 novel 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 cache replacement 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 order of increasing access probabilities; the buffer that is least likely to be accessed is stored at the head of the list (see Figure 6). Thus, 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 [39]. 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.[4] 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 (see Figure 6). 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 quality of service (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, etc. Depending on the service class of the client (i.e., periodic real-time or aperiodic real-time), the resource manager then invokes an appropriate admission control algorithm. The admission control algorithm determines if there is sufficient disk bandwidth and buffer space to service the client, and returns a set of QoS parameters indicating this availability. Based on resource availability, these QoS parameters can be less than or equal to the requested QoS. The resource manager also tentatively reserves these resources for the client. In the second phase, the client either confirms or rejects the QoS parameters returned by the admission control algorithm. In the former case, the tentatively reserved resources are committed; otherwise resources are freed and the negotiation process is restarted, either with a reduced QoS requirement, or at a later time. If the resource manager does not receive a confirmation or rejection from the client within a specified time interval, it frees resources that were tentatively reserved. 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; 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 [28, 43, 42]. 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. Symphony supports both deterministic and statistical admission control algorithms; a detailed discussion of these algorithms is beyond the scope of this paper [43].

[4] 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.

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 [35]. 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 [15]. 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 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 are suitable for predominantly read-only environments [45]. 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. Such a policy, however, degrades performance for large disk arrays. Consequently, large arrays are typically partitioned into sub-arrays and each file is striped across a single sub-array [37]. 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 [36]. Such contiguous placement significantly reduces seek and rotational latency overheads incurred during video playback.

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 serviced using the server-push architecture, while aperiodic real-time requests are serviced using the client-pull architecture. Periodic real-time clients are serviced by the video module in terms of periodic rounds For each such client, the video module generates a list of blocks to be read or written at the beginning of each round and issues requests for these blocks. 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 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 both 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 7 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 inode structure permits efficient random access for fixed-size blocks. Supporting random access for variable-size blocks, however, is difficult, since variable block sizes complicate the mapping from byte offsets to block locations in the byte map. For such blocks, the video module maintains a mapping from byte offsets to block locations for *base blocks* (instead of variable-size blocks). Since base blocks are of a fixed size, this policy permits efficient random access for variable-size blocks, albeit at the expense of increased storage space requirements for the byte map.

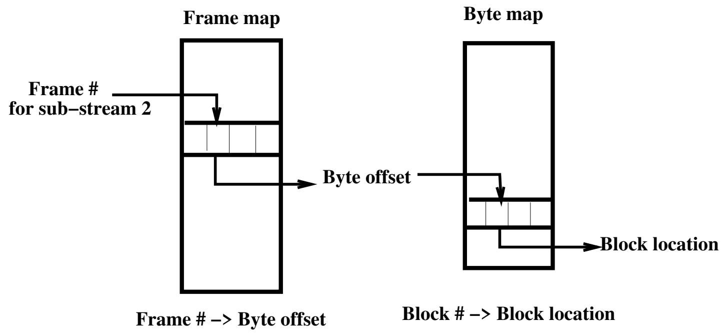


Figure 7: The structure of the video inode. Assuming that a video file contains n sub-streams, both the frame map and the byte map contain a sequence of n-tuples. Each tuple in the frame map represents a frame, and the i^{th} field of a tuple denotes the location (i.e., byte offset) of that frame in 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 [5]. 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 [8].

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 [2, 20]. 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.

6.3 Prototype Implementation

We have implemented a prototype of Symphony on the Solaris 2.5.1 platform. The Symphony prototype runs as a single multi-threaded process in user space and access disks as raw devices; a separate kernel module implements the vnode interface for Symphony so as to allow existing applications to access files stored on Symphony. Symphony employs a thread-per-request model to service client-pull requests—a new thread is created for each new request,

Table 1: The Symphony File Server Interface

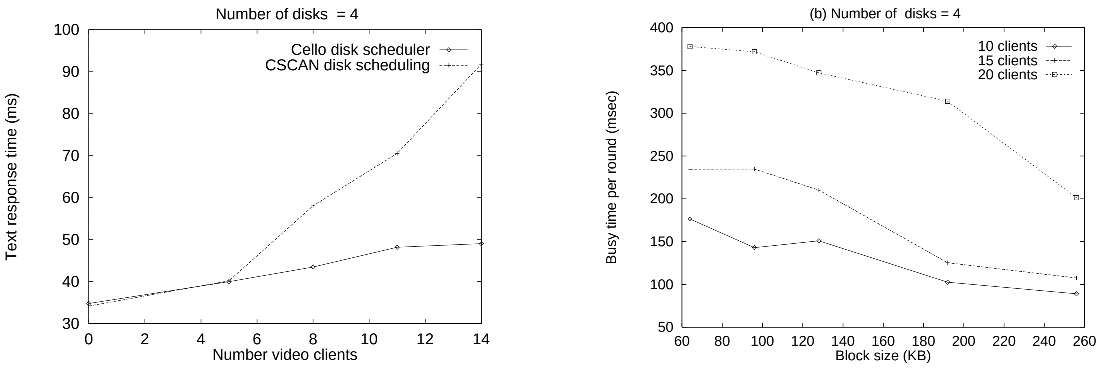
fileHandle	=	ifsCreate(file name, data type, options)
fileHandle	=	ifsOpen(file name, options)
result	=	ifsDelete(file name)
result	=	ifsClose(fileHandle)
result	=	ifsFlush(fileHandle)
result	=	ifsRead(fileHandle, size, buffer, optional deadline)
result	=	ifsWrite(fileHandle, size, buffer, optional deadline)
result	=	ifsSeek(fileHandle, offset)
result	=	ifsPeriodicRead(fileHandle, recvPort[numSubStreams])
result	=	ifsPeriodicWrite(fileHandle, sendPort[numSubStreams])
result	=	ifsStop(fileHandle)
result	=	ifsQosNegotiate(fileHandle,qosIn,qosOut)
result	=	ifsQosConfirm(fileHandle,qos)
metaData	=	ifsgetMetaData(fileHandle,options)

which then checks if the requested data is cached, hands the request to the disk scheduler on a cache miss, and finally transmits the retrieved data to client. Server-push requests are serviced using two perpetual threads—a disk thread and a network thread. The disk thread issues read/write requests for each server-push client at the beginning of each round, and the network thread transmits this data to clients and also receives data that is being written to disk. The Cello disk scheduler employs its own threads to instantiate the class-independent and class-specific schedulers; these threads are responsible for processing requests in the pending and scheduled queues.

Symphony supports two sets of interfaces:

- *NFS interface:* The kernel module of Symphony implements the vnode interface of Solaris, and thereby allows Symphony to support NFS operations. To do so, the kernel module implements two new system calls that enable the user-space Symphony server to communicate with the kernel. The kernel module also maps a set of buffers from the kernel address space to Symphony address space. This enables efficient, zero-copy data transfer from the kernel address space to the Symphony address space and vice versa. Thus, the Symphony-kernel communication employs system calls on the control path and the kernel remapped buffers on the data path. Note that, by implementing the NFS interface, Symphony can support all existing applications without any modifications.

- *Native RPC interface:* Due to its stateless nature, the NFS protocol can not be used to support server-push (i.e., streaming) requests. To overcome this drawback, Symphony supports a native RPC interface that supports server-push access to files. In addition to methods for server-push accesses, the interface also supports methods for creating and deleting files, methods for client-pull-based reads and writes, and methods for negotiating quality of service for real-time requests (see Table 1).



(a) (b)

Figure 8: Performance of text and video clients.

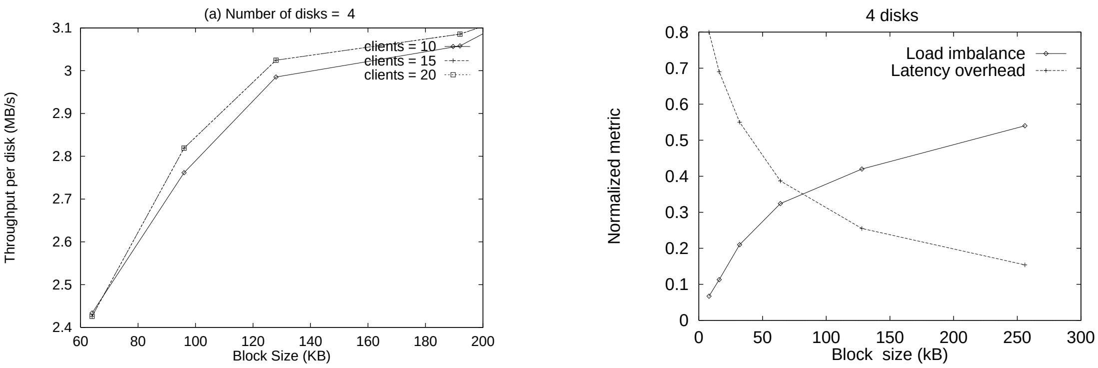
7 Experimental Evaluation of the Symphony Prototype

In this section, we experimentally evaluate the Symphony prototype to demonstrate the efficacy of its disk scheduling, placement, and failure recovery algorithms. The testbed for our experiments consists of a cluster of Sun workstations interconnected by 100Mb/s Ethernet as well as a 155Mb/s 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. 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 Cello disk scheduler delays the servicing of real-time requests until their deadlines and uses the available slack to service interactive requests. By giving priority to interactive text requests whenever sufficient slack is available, Cello provides low response times to these requests. To demonstrate this behavior, we compiled two versions of the prototype, one which used Cello and the other which used CSCAN. In both cases, we populated the server with a large number of text and video files. Each text file was 64KB 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.[5] The playback rate for the each video file was 30 frames/s and the average bit rate was 1.5 Mb/s. We assigned weights of $w_1 = 0.6, w_2 = 0.05$, and $w_3 = 0.35$ to the real-time, throughput-intensive, and interactive classes, respectively. 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 8(a) plots the average response time for a 8KB request observed in the two cases. The figure shows that the Cello 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

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



(a) (b)

Figure 9: Impact of the placement policy. Figure (a) demonstrates the effect of block size on server throughput. Figure (b) demonstrates its effect on the disk latency overhead and the load imbalance on the array.

requests increases with increase in number of video requests. In contrast, Cello always gives priority to text requests regardless of the number of video requests (provided sufficient slack is available). Moreover, unused allocation of the real-time class is reassigned to the interactive 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 8(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 real-time requests (since $w_1 = 0.6$ and the round duration is 1s). Hence, the disk scheduler is able to meet the real-time requirement of video clients. Together, Figures 8(a) and (b) show that, even at a moderate disk utilization level of 25%, Symphony yields a factor of 1.9 improvement in response time of text requests over conventional disk scheduling algorithms, while meeting the deadlines of all real-time video requests,

Recall that, Symphony allows a client to specify the block size at file creation time. The block size used to stripe a file can have a significant impact on the server performance. Choosing a large block size reduces seek and rotational latency overheads and increases disk throughput (see Figures 9(a) and (b)). Doing so, however, also reduces the the total number of blocks accessed from the array and results in a sparsely loaded array. This increases the load imbalance across disks in the array (see Figure 9(b)) and can result in a reduction in throughput. Hence, clients must choose a block size that maximize throughput by balancing the latency overhead as well as the load. In [37], we proposed analytical models that use server configuration and workload characteristics to determine such an optimal block size. Clients can use these models to determine a block size appropriate to their needs.

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

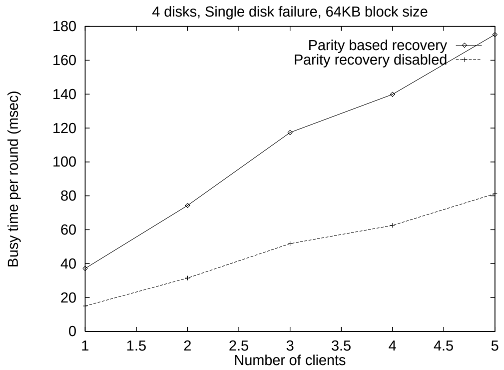


Figure 10: Reliable reads versus unreliable reads. The figure shows that disabling parity-based reconstruction significantly lowers the load on the server.

reconstruction imposes a large (100%) overhead on the server [7]; 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 reduces the failure recovery overhead from a factor of two to zero. 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) [44].

8 Lessons Learned from the Symphony Project

In this section, we discuss the lessons learned from the design and implementation of Symphony over the fast four years. We first discuss the merits of the Symphony approach and then discuss disadvantages and hurdles that we encountered.

8.1 Advantages and Benefits

- *Modularity simplifies file system design.* Our initial implementation of Symphony was monolithic. The increasing complexity of our file system implementation (due to the need for supporting multiple policies) necessitated a redesign, resulting in a modular two layer architecture. Our two layer architecture, based on the adage of separating mechanisms from policies, substantially simplified the addition of new modules to the file system.

- *Multi-service file system design.* File systems that provide a single class of service (e.g., best-effort service) and are optimized for a single performance criterion are no longer adequate to handle the the increasing diversity of today's application workloads. Hence, file systems of the future must provide multiple classes of service to meet diverse application needs. Operating systems such as Nemesis [30] are also founded upon this tenet.

- *Application-specific and data-type specific policies yield performance gains.* Exploiting application-specific and data type-specific semantics while implementing file system policies results in substantial performance gains. The class-specific schedulers of Symphony exemplify this principle by yielding a factor of 2.5 improvement in response time over conventional disk scheduling algorithms such as SCAN [33].

8.2 Disadvantages and Hurdles

- *Generality of mechanisms can hurt performance.* Symphony attempts to keep the mechanisms in the data type independent layer very general so as to allow a diverse set of policies to be implemented in the data type specific layer. In some cases, this generality can hurt performance. For instance, the storage manager allows any block size to be chosen for a file—a policy that can result in disk fragmentation. Instead, restricting the set of block sizes to a small set of values may reduce fragmentation effects without any significant loss in performance or flexibility.

- *Complex scheduling polices.* An integrated file system multiplexes its resources among different application classes; to do so, it must employ complex scheduling and resource management policies that can be simultaneously optimized for different performance criteria. The design and implementation of such policies make file system design more complex.

- *Lack of benchmarks.* Most file system benchmarks (e.g., the Andrew benchmark) have been designed for best-effort textual workloads and are inadequate for evaluating the performance of multimedia file systems that service heterogenous workloads. The lack of a benchmark tailored for multimedia file systems was a key hurdle in evaluating the performance of Symphony. Consequently, our evaluation was restricted to the use of micro-benchmarks that focused on specific components of Symphony.

9 Related Work

Several techniques for the storage and retrieval of continuous media that have been proposed in the literature [1, 16, 28, 40, 41, 4]. Symphony builds upon these techniques and integrates them into a general purpose file system. For instance, the interval caching policy employed by the video module was proposed by Dan et. al.[8] The efficacy of using variable-size blocks for storing video files has been demonstrated by several studies [16, 6]. Techniques for approximate reconstruction of image and video data (e.g., JPEG and MPEG) have been proposed by Vin et. al.[44] and Danskin et. al. [9] Admission control algorithms have been discussed by Rangan et. al. [28] and Vin et. al.[43]

Several recent and ongoing research efforts have focussed on designing integrated file systems. The Fellini storage manager [22] and CMFS [1] are file systems that can handle both real-time continuous media data and best-effort textual data. Similarly, MMFS is a single disk file system that adds continuous media support to a FreeBSD-based file system [25]. The Tiger Shark file system from IBM and XFS from SGI are results of commercial efforts to build integrated file systems [13, 14]. These file systems come closest to Symphony in terms of the offered features. For instance, these file systems support variable-size blocks (referred to as *extents*), guaranteed rate I/O, etc. However, they do not employ features such as a disk scheduler that supports diverse applications, 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 all resources are dynamically shared among applications.

The logical disk abstraction [17] 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 requests as well as best-effort requests.

Several efforts have focussed on designing extensible and adaptive systems. Exokernel [10] and SPIN [3] are two efforts towards building extensible operating systems. These systems focus on securely multiplexing resources among untrusted best-effort applications, whereas Symphony focuses on multiplexing resources so as to provide different qualities of service to applications. Stackable file systems enable file system extensions by layering one file system abstraction on top on another [18]. Finally, the Odyssey system employs application-aware adaptation (i.e., a collaborative partnership between the operating system and applications) to efficiently support diverse applications in a mobile computing environment [26].

10 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.

Our experiments with the Symphony prototype demonstrated the efficacy of dynamic allocation of resources and supporting multiple data type specific policies. Our results showed that Symphony yields a factor of 1.9 improvement in text response time over conventional disk scheduling algorithms even in the presence of moderate video loads, while continuing to meet the real-time requirements of video clients. We also showed that (i) tailoring the block size and placement policy to application requirements improves server throughput, and (ii) supporting data type specific failure recovery policies enables Symphony to reduce the recovery overhead for continuous media applications from a factor of two to zero.

References

[1] D. Anderson, Y. Osawa, and R. Govindan. A File System for Continuous Media. *ACM Transactions on Computer Systems*, 10(4):311–337, November 1992.

[2] M. J. Bach. *The Design of the Unix Operating System*. Prentice Hall, 1986.

[3] B. Bershad, S. Savage, P. Pardyak, E. Sirer, M. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, Safety, and Performance in the SPIN Operating System. In *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*, pages 267–284, December 1995.

[4] M. Buddhikot, G. Parulkar, and J. Cox. Design of a Large Scale Multimedia Storage Server. *Journal of Computer Networks and ISDN Systems*, pages 504—524, Dec 1994.

[5] P. Cao. *Application Controlled File Caching and Prefetching*. PhD thesis, Princeton University, 1996.

[6] 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.

[7] 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*, pages 145–185, June 1994.

[8] 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*, pages 344–351, 1996.

[9] 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*, pages 321–332, November 1995.

[10] D. Engler, M. Kaashoek, and J. O'Toole. Exokernel: An Operating System Architecture for Application-Level Resource Management. In *Proceedings of the Fifteenth ACMSymposium on Operating Systems Principles*, pages 251–266, December 1995.

[11] R. H. Patterson et. al. Informed Prefetching and Caching. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles*, December 1995.

[12] S. D. Gribble, G. Manku, D. Roselli, E. Brewer, T. Gibson, and E. Miller. Self-Similarity in File Systems. In *Proceedings of ACM SIGMETRICS '98, Madison, WI*, June 1998.

[13] R. Haskin. Tiger Shark–A Scalable File System for Multimedia. *IBM Journal of Research and Development*, 42(2):185–197, March 1998.

[14] M. Holton and R. Das. XFS: A Next Generation Journalled 64-bit File System with Guaranteed Rate I/O. Technical report, Silicon Graphics, Inc, Available online as http://www.sgi.com/Technology/xfs-whitepaper.html, 1996.

[15] 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.

[16] P W. Jardetzky. *Network File Server Design for Continuous Media*. PhD thesis, University of Cambridge, August 1992.

[17] 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.

[18] Y. Khalidi and M. Nelson. Extensible File Systems in Spring. In *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*, pages 1–14, December 1993.

[19] D. E. Knuth. *The Art of Computer Programming Volume 1: Fundamental Algorithms*. Addison Wesley, 1973.

[20] 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.

[21] J. P. Lehoczky and S. Ramos-Thuel. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Proceedings of Real Time Systems Symposium*, pages 110–123, December 1992.

[22] 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.

[23] 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):181–197, August 1984.

[24] L. McVoy and S. Klieman. Extent-like Performance from a UNIX File System. In *Proceedings of Summer USENIX Conference, Annaheim, CA*, pages 137–144, June 1990.

[25] T. Niranjan, T. Chiueh, and G. Schloss. Implementation and Evaluation of a Multimedia File System. In *Proceedings of ICMCS'97, Ottawa, Canada*, 1997.

[26] B. Noble, M. Satyanarayanan, D. Narayanan, J. Tilton, J. Flinn, and K. Walker. Agile Application-Aware Adaptation for Mobility. In *Proceedings of the 16th ACM Symposium on Operating System Principles, St. Malo, France*, October 1997.

[27] D. Patterson, G. Gibson, and R. Katz. A Case for Redundant Array of Inexpensive Disks (RAID). In *Proceedings of ACM SIGMOD'88*, pages 109–116, June 1988.

[28] P. Venkat Rangan and H.M. 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*, pages 81–94, October 1991.

[29] A.L. Narasimha Reddy and J. Wyllie. Disk Scheduling in Multimedia I/O System. In *Proceedings of ACM Multimedia'93, Anaheim, CA*, pages 225–234, August 1993.

[30] Timothy Roscoe. *The Structure of a Multi-Service Operating System*. PhD thesis, University of Cambridge Computer Laboratory, April 1995. Available as Technical Report No. 376.

[31] M. Rosenblum and J. Ousterhout. The Design and Implementation of a Log-Structured File System. In *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, pages 1–15, October 1991.

[32] P. Shenoy, P. Goyal, and H M. Vin. Architectural Considerations for Next Generation File Systems. In *Proceedings of the Seventh ACM Multimedia Conference, Orlando, FL (to appear)*, November 1999.

[33] P Shenoy and H M. Vin. Cello: A Disk Scheduling Framework for Next Generation Operating Systems. In *Proceedings of ACM SIGMETRICS Conference, Madison, WI*, pages 44–55, June 1998.

[34] P. Shenoy and H M. Vin. Efficient Support for Interactive Operations in Multi-resolution Video Servers. *ACM/Springer Multimedia Systems Journal*, 7(3):241–253, July 1999.

[35] P J. Shenoy, P. Goyal, S. Rao, and H M. Vin. Design and Implementation of Symphony: An Integrated Multimedia File System. Technical Report TR97-09, Dept. of Computer Sciences, Univ, of Texas at Austin, March 1997.

[36] P J. Shenoy and H M. Vin. Efficient Support for Scan Operations in Video Servers. Technical Report TR96-35, Department of Computer Sciences, Univ. of Texas at Austin, December 1996.

[37] P J. Shenoy and H M. Vin. Efficient Striping Techniques for Multimedia File Servers. In *Proceedings of NOSSDAV '97, St. Loius, MO, Extended version available as Technical Report TR96-27, Dept. of Computer Sciences, Univ. of Texas at Austin*, pages 25–36, May 1997.

[38] 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*, pages 45–48, April 1996.

[39] R. Tewari, H M. Vin, A. Dan, and D. Sitaram. Caching in Bandwidth and Space Constrained Hierarchical Hyper-Media Servers. Technical Report TR96-30, Department of Computer Sciences, Univ. of Texas at Austin, December 1996.

[40] F A. Tobagi, J Pang, R Baird, and M Gang. Streaming RAID – A Disk Array Management System For Video Files. In *Proceedings of ACM Multimedia '93, Anaheim, CA*, pages 393–400, 1993.

[41] M. Vernick, C. Venkatramini, and T. Chiueh. Adventures in Building the Stony Brook Video Server. In *Proceedings of ACM Multimedia'96*, 1996.

[42] H. M. Vin, A. Goyal, and P. Goyal. Algorithms for Designing Large-Scale Multimedia Servers. *Computer Communications*, 18(3):192–203, March 1995.

[43] 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*, pages 33–40, October 1994.

[44] 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*, pages 12–21, June 1995.

[45] H.M. Vin, S.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.*, pages 158–165, May 1995.