Report Number: WUCS-97-38

1997-01-01

Enhancements to 4.4 BSD UNIX for Efficient Networked Multimedia in Project MARS

Milind M. Buddhikot, Xin Jane Chen, Dakang Wu, and Guru M. Parulkar

Cluster based architectures that employ high performance inexpensive Personal Computers (PCs) interconnected by high speed commodity interconnect have been recognized as a cost-effective way of building high performance scalable Multimedia-On-Demand (MOD) storage servers [4, 5, 7, 9]. Typically, the PCs in these architectures run operating systems such as UNIX that have traditionally been optimized for interactive computing. They do not provide fast disk-to-network data paths and guaranteed CPU and storage access. This paper reports enhancements to the 4.4 BSD UNIX system carried out to rectify these limitations in the context of our Project Massively-parallel And Real-time Storage (MARS) [7].... **Read complete abstract on page 2.**

Enhancements to 4.4 BSD UNIX for Efficient Networked Multimedia in Project MARS

Milind M. Buddhikot, Xin Jane Chen, Dakang Wu, and Guru M. Parulkar

Complete Abstract:

Cluster based architectures that employ high performance inexpensive Personal Computers (PCs) interconnected by high speed commodity interconnect have been recognized as a cost-effective way of building high performance scalable Multimedia-On-Demand (MOD) storage servers [4, 5, 7, 9]. Typically, the PCs in these architectures run operating systems such as UNIX that have traditionally been optimized for interactive computing. They do not provide fast disk-to-network data paths and guaranteed CPU and storage access. This paper reports enhancements to the 4.4 BSD UNIX system carried out to rectify these limitations in the context of our Project Massively-parallel And Real-time Storage (MARS) [7]. We have proposed and implemented the following enhancements to a 4.4 BSD compliant public domain NetBSD UNIX operating syste: (1) A new kernel buffer management system called Multimedia M-buf (mmbuf) which shortens the data path from a storage device to network interface, (2) priority queueing within the SCSI driver to differentiate between real-time and non-real-time streams, and (3) integration of these new OS services with a CPU scheduling mechanism called Real Time Upcall [22] and a software disk striping driver called Concatenated Disk (ccd). These enhancements collectively provide quality of service guarantee and high throughput to multimedia stream connections. Our experimental results demonstrate throughput improvements and QOS guarantees on the data path from the disk to network in a MOD server.

Enhancements to 4.4 BSD UNIX for Efficient Networked Multimedia in Project MARS

Milind M. Buddhikot, Xin Jane Chen,
Dakang Wu and Guru M. Parulkar

WUCS-97-38

July 1997

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130

Enhancements to 4.4 BSD UNIX for Efficient Networked Multimedia in Project MARS*

Milind M. Buddhikot
milind@dworkin.wustl.edu
+1 314 935 4203

Xin Jane Chen
JXCHEN@us.oracle.com
+1 415 506-8617

Dakang Wu
dw@arl.wustl.edu
+1 314 935 8563

Guru M. Parulkar
guru@arl.wustl.edu
+1 314 935 7534

Abstract

Cluster based architectures that employ high performance inexpensive Personal Computers (PCs) interconnected by high speed commodity interconnect have been recognized as a cost-effective way of building high performance scalable Multimedia-On-Demand (MOD)storage servers [4, 5, 7, 9]. Typically, the PCs in these architectures run operating systems such as UNIX that have traditionally been optimized for interactive computing. They do not provide fast disk-to-network data paths and guaranteed CPU and storage access. This paper reports enhancements to the 4.4 BSD UNIX system carried out to rectify these limitations in the context of our *Project Massively-parallel And Real-time Storage (MARS)* [7]. We have proposed and implemented the following enhancements to a 4.4 BSD compliant public domain NetBSD UNIX operating system: (1) A new kernel buffer management system called Multimedia M-buf (mmbuf) which shortens the data path from a storage device to network interface, (2) priority queueing within the SCSI driver to differentiate between real-time and non-real-time streams, and (3) integration of these new OS services with a CPU scheduling mechanism called *Real Time Upcall* [22] and a software disk striping driver called *Concatenated Disk (ccd)*. These enhancements collectively provide quality of service guarantee and high throughput to multimedia stream connections. Our experimental results demonstrate throughput improvements and QOS guarantees on the data path from the disk to network in a MOD server.

1 Introduction

Large scale Multimedia-on-Demand (MOD) recording and playback services accessed by potentially thousands of concurrent clients will be critical components of the rapidly evolving information super-highway. The multimedia data handled by these services require Quality-of-Service (QOS) guarantees in the form of guaranteed bandwidth and bounded delay. The end-to-end nature of these services requires that such guarantees be provided by the end systems, namely the storage server and the client device such as a PC, a workstation or a set-top box, and the network that connects them. Within a server system, such services periodically transfer data between the storage and network subsystems and thus, both these subsystems must provide QOS guarantees. Designing high performance scalable servers and services that support such guarantees has been recognized to be a challenging task.

The interactive MOD services developed in our project are based on an innovative cluster based storage server architecture called Massively-parallel And Real-time Storage (MARS)[5, 7]. This architecture consists of several high performance storage nodes (such as PCs) interconnected by a fast desk-area or system-area ATM interconnect. Each storage node in our current prototype is a Pentium Pro 200 MHZ PC with 30 GB local storage and runs a public-domain 4.4 BSD UNIX called NetBSD. However, the existing UNIX systems used for general purpose computing do not provide mechanisms required to support QOS guarantees and also do not provide efficient data and control path between storage and network subsystems.

In NetBSD, the file system software uses a sophisticated buffer cache to transfer data between user space and the storage, whereas the network protocol stacks use a different buffering system called *mbufs* to transfer data between the user space and the network. Due to this mismatch, any application initiated data trans-

fer between a file system and the network needs excessive data copying. In MOD environment where services are implemented in user space, such data copying for every active client reduces throughput and limits the total number of clients (revenue). Therefore, a fast data path that can provide zero-copy data transfer between storage and network is desirable. Also, a simple user level API is necessary to enable user level applications to easily control such a data path.

Existing device drivers for storage systems do not distinguish between real-time and non-real-time requests and therefore provide no guarantees. Clearly, the storage driver must prioritize real-time requests over non-real-time requests. A user level application can effectively make use of such real-time service guarantees from the disk driver only if the CPU scheduler provides QOS guarantees in the form of periodic execution.

1.1 Research Contributions

We would like to note that the shortcomings discussed above have been well known to be performance bottlenecks [2, 12, 13, 21, 26]. However, none of the earlier solutions are complete and research efforts such as [3, 4, 15, 18, 25, 28] are underway to rectify them. (Please see Section 6). Our work shares some common ideas and objectives with these research efforts.

This paper describes our innovative ideas and details the software infrastructure we have developed to implement these ideas. The design, implementation and performance evaluation of the novel OS enhancements are the primary research contributions of this paper. Specifically, we have proposed and implemented the following enhancements to a 4.4 BSD compliant public domain NetBSD UNIX operating system: (1) A new kernel buffer management system called Multimedia M-buf (mmbuf) which shortens the data path from a storage device to network output device, (2) priority queueing within the SCSI driver to differentiate between real-time and non-real-time streams, and (3) integration of these new OS services with a CPU scheduling mechanism called *Real Time Upcall* [22] and a software disk striping driver called *Concatenated Disk (ccd)*. We have created a new system call API for applications to access these services.

We have implemented these ideas in NetBSD 1.2G and demonstrated clear performance improvements. Our experimental results show: (1) A $\approx$ 60% overall improvement in throughput from storage to the network interface by combined use of software striping and zero copy data path, (2) QOS guarantees in the form of periodic accesses from the enhanced SCSI system, and (3) guaranteed access to CPU and storage resources at the user level for applications that employ RTUs and access the new OS enhancements.

1.2 Organization of the paper

The rest of this paper is organized as follows: Section 2 describes the sources of inefficiencies and lack of QOS in the existing control and data path for network destined data retrieval from the storage subsystem. A brief overview of our new solutions is provided in Section ??. Section 3 describes at length the design of the new *mmbuf* buffer management system. Section 4 describes our two step approach to providing guaranteed periodic access to storage: namely, modifications to SCSI driver to support priority queueing and use of Real-Time Upcalls (RTU). This section also details the system call API available to a user application to access these new OS services. In Section 5, we present detailed performance evaluation of these OS modifications and discuss performance benefits and limitations. Finally, we present our conclusions.

2 Limitations of Existing File I/O for Networked Multimedia

In this section, we summarize the limitations of existing UNIX to support networked multimedia and thus motivate the need for our work.

- **Unnecessary data copying is a performance penalty:** Figure 1 illustrates the layered architecture used in the storage and network I/O systems of current UNIX operating systems. Clearly, in a MOD server implemented in user space the data transfer path from a disk to the network interface involves two memory copies: the first copy (in response to `read()` call) moves data from the kernel buffer cache to a user space buffer. The second copy (in response to a `send()` call) by the the socket layer copies the data from a user space buffer into the mbuf chain in the kernel. This approach works fine for small sized accesses observed in general purpose I/O, such as traditional text, and binary file accesses. However, multimedia data such as audio, video, and animations do not possess any caching properties: first they have a ravenous appetite for memory space and second, they are relevant only for very a small duration from the time of their retrieval. That is, data is often replaced before it can be reused, rendering the extra copy a performance penalty. Consider, a 128 MB machine with typically 6.5 MBs configured as buffer cache. This cache is enough to store 3 seconds of an average MJPEG file, and the the kernel has to replace everything in the buffer cache every 3 seconds. Therefore, the buffer cache blocks can be reused only if several processes reading the same video file are phase locked to each other in a 3-second time interval. Such behavior among interactive clients will be rare. Therefore, retrieving multimedia data through a buffer cache does not provide any performance benefits. Also, any application initiated data transfer from a disk file to the network requires use of two different buffer systems, which were designed with different objectives, and leads to excessive data copying and system call overheads. Large amount of data copy from memory to memory not only takes processor time but also consumes precious memory and system bus bandwidth.

- **Lack of guaranteed storage access:** The storage subsystems in the current 4.4 BSD UNIX does

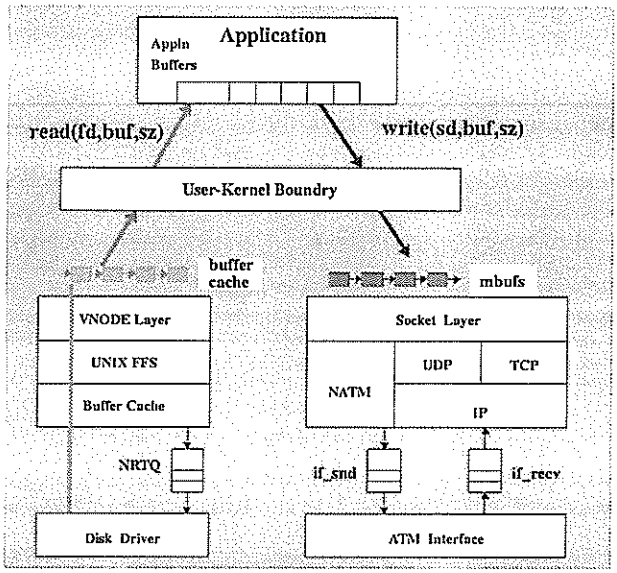


Figure 1: Existing File and Network I/O

not differentiate between real-time and non-real-time applications for disk I/O. All requests to the disk driver are queued into a single job queue and ordered using the elevator algorithm to achieve efficient disk head movement. Clearly, a job can take an arbitrarily long time to complete if there are several other processes doing disk I/O. Such behavior is undesirable for real-time requests which must be completed by a certain deadline for the retrieved data to be useful.

- **Lack of guaranteed CPU access:** CPU scheduling in current UNIX systems is optimized for interactive computing and thus, does not provide any mechanisms for applications to request periodic access to the CPU. Clearly, in the event that a file system is enhanced to support periodic QOS, user applications must be able to obtain guaranteed access to the CPU to avail these guarantees.

3 Design of the mmbuf buffering system

Figure 2 (a) shows the data structure of an mmbuf, which is a superset of an mbuf and the BSD buffer cache block. Each mmbuf consists of a header and a data buffer. The mmbuf header consists of the following four parts:

1. **Mbuf header:** The `struct mbuf` field in the mmbuf header represents the mbuf header. It is used to store information required to send the data stored in the mmbuf to the network.
2. **Buffer cache header:** The `struct buf` field in the mmbuf header represents a buffer cache block header. It is used by the file system to read data into the buffer.
3. **Pointer to a buffer manager:** The mmbuf header maintains a pointer - `bmptr`, using which a buffer manager in the upper level of the kernel can be accessed. The mmbuf can be in four different states: *empty, full, read_in_progress* and *send_in_progress*. This manager manages the mmbuf's status and operation, and provides a handler (`bm_iodone()`) used by the file system to update the mmbuf's status.
4. **Padding:** A padding of 96 bytes is used to make the header size 256 byte to avoid memory fragmentation. This padding provides sufficient room for any future enhancements to buffer cache or mbufs.

Each mmbuf has a data cluster of one or more virtually contiguous pages associated with it (Figure 2 (a)). The maximum size of a cluster is a configurable parameter. Both the mbuf and buffer headers in the mmbuf header maintain a pointer to the data cluster. When data are read from the disk, the cluster is accessed from the pointer in the buffer header whereas when data are sent to the network interface, the same cluster is accessed from the mbuf header. Note that the disk drivers can perform scatter-gather I/O to virtually contiguous clusters greater than a page in size. However, in case of systems that do not support Direct Virtual Memory Access (DVMA)[1], the maximum size of an mbuf cluster is limited to a page. Due to this network drivers traditionally do not handle DMA of buffers greater than a page in size. This means that though an mmbuf with a 16 KB cluster can be passed as a single buffer block to the disk driver, it must be

[1] Sun machines support DVMA, whereas PCI based Intel machines don't

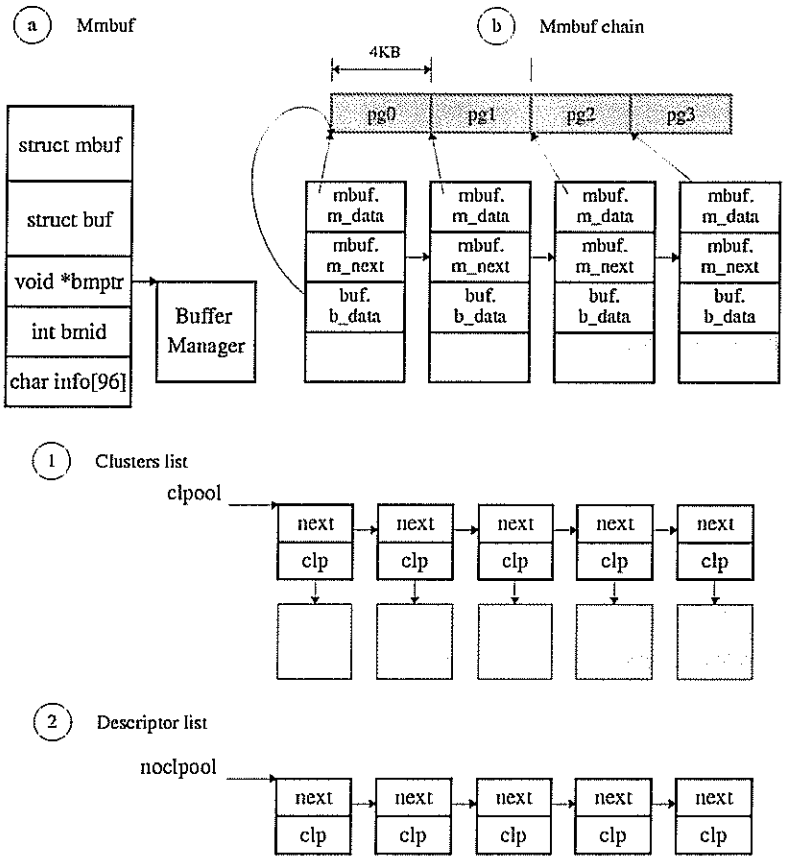


Figure 2: New Multimedia Memory Buffer (mmbuf)

passed as a chain of 4 mmbufs to the network protocol stack. This is illustrated in Figure 2 (b).

Since we unified the buffering structure in file I/O and network I/O, we have to support operations from both domains. The interface to the `mmbuf` system is described below (Figure 3).

Mmbuf system initialization: `mmbinit()`

At the system boot time, the initialization routine in the kernel sets up a separate submap — a pagemap `mmb_map` in kernel virtual address space - for mmbuf data clusters. It also invokes the mmbuf initialization function - `mmbinit()` to allocate wired-down (non-pageable) pages in the `mmb_map`. The mmbuf system maintains two lists of cluster descriptors (Figure 2, (1),(2)). The first list, called *cluster list* is accessed by the `clpool` ptr and contains descriptors that have cluster of MMCLBYTES size associated with them. The second list, called the *descriptor* list, is accessed by the `noclpool` ptr and contains empty descriptors with no associated cluster. Initially, 16 mmbuf data clusters are put on the *cluster* list.

Allocating and deallocating an mmbuf (`mmget(struct mmbuf *mp,int flag)` and `mm_free(struct mmbuf *m)`)

The MMGET routine allocates an mmbuf with an associated data cluster. It removes the cluster from the descriptor at the head of the *cluster* list and inserts the free descriptor at the head of the *descriptor* list for reuse. The relevant attribute fields in mbuf and buffer header portions of mmbuf header are initialized before it is returned. A similar function, `mm_getchain()` allocates a chain with (MMCLBYTES/NBPG) mmbufs, each pointing to a page in the cluster. If a cluster allocation is attempted when no data clusters are available on the *cluster* free list, more wired-down data clusters are allocated from virtual memory map `mmb_map` and put onto the list. A mmbuf is deallocated using the `mm_free()` function, which removes a free descriptor from the *descriptor* list, initializes it with the cluster to be freed and inserts the cluster at the head of the *cluster* list.

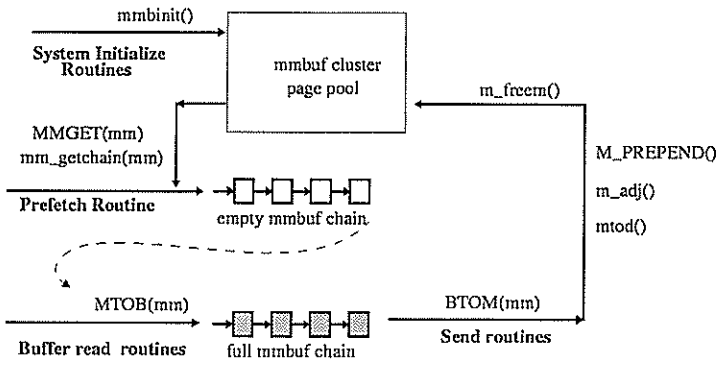


Figure 3: Invocation of mmbuf interface functions

Using mmbufs in file I/O

Since the file I/O uses the mmbuf as a buffer cache block, a macro MTOB(void *m) is provided which, given the mmbuf pointer, returns the pointer to the file system buffer header. In addition to the attributes of the buffer block that are set in the `b_flags` field of the buffer header for normal reads, two additional flags B_CALL and B_MMBUF are set in case of file I/O using mmbufs. The B_CALL flag indicates that when the data is read from the disk into the data cluster, before calling the standard `b_iodone()` handler, a custom handler `bm_iodone()` should be invoked. This new handler performs the buffer manager function and appropriately modifies the state of the mmbuf from *empty* to *full*.

Using mmbufs in network operations

It is desirable that network protocol routines and interface drivers be able to transparently use mmbufs as regular mbufs without requiring significant code changes. Typically, the network protocols compose packets by adding protocol headers or trailers in the form of mbufs to the head or tail of an existing mbuf chain containing data. Since an mbuf header in the mmbuf describing associated data cluster will be identical to a stand-alone cluster mbuf header, same operations can be carried out even if data is in an mmbuf chain. However, one crucial difference here is that the data cluster of the mmbuf is allocated from a page pool different from the one for cluster mbufs. The mmbuf chains are allocated using the memory allocation routines of the mmbuf system when data fetch requests are generated. The network I/O routines can add mbufs (allocated by the standard mbuf allocation routines) as protocol header or trailers to such chains.

In a typical packet send operation, the network interface driver calls the mbuf memory deallocation function `m_freem` after packet is copied to the network interface card. It is desirable that the driver retain the same call to free an mbuf+mmbuf chain. However, the deallocation of an mmbuf must return the associated cluster pages to the mmbuf page pool. To achieve these two objectives, we make use of an interesting feature in the original mbuf design which allows for a pointer to an external function to be called when the associated cluster page is to be freed. The current mbuf implementation does not use this pointer; instead it has a stand-alone function. This unused feature is exploited in our design; we set this pointer to our own `mm_free()`routine when we initialize an mmbuf.

The mbuf routines such as `m_copy` and `m_copym`, reference global variables of the mbuf's cluster page pool. These routines therefore need to be modified to ensure that they differentiate between an mmbuf and an mbuf, and update the global variables for different page pools consistently. These routines are commonly used in transport protocols such as TCP which support retransmissions. In our current MOD testbed, we use AAL5 and AAL0 native-mode ATM protocols that have UDP semantics and hence, do not require these functions. Therefore, our current mmbuf implementation does not support these enhanced functions which are crucial to ensure that the mmbuf system can be used with the TCP protocol. To accomplish this, a simple change is required: if the MM_MMBUF flag is set in the mbuf, the reference counter for mmbuf clusters is updated otherwise the reference counter for mbufs is modified. We believe that this change is straightforward to incorporate in our current design and will be available in the next release of our software.

4 Periodic QoS Guarantees

In this section we first describe the priority queueing within the SCSI driver. We then briefly describe a new CPU scheduling technique called Real-Time Upcalls (RTU) that user applications can use to gain periodic access to CPU to exploit QOS from the SCSI driver. We also discuss the CCD software disk array that can be combined with the mmbuf system and SCSI QOS to achieve much higher disk to network throughput.

4.1 Periodic QoS Guarantees from Storage System

In this section we describe the fair queueing over multiple priority classes within the SCSI driver. We also discuss the CCD software disk array that can be

combined with the mmbuf system and SCSI QOS to achieve much higher disk to network throughput.

4.2 Priority Queueing within the SCSI Disk Driver

The existing queueing mechanism (Figure 6 (1)) consists of a single request queue sorted using a disk scheduling algorithm such as the elevator algorithm (a variant of the Circular-SCAN algorithm) . This request queue is serviced by an event-driven service function - `sd_start`, which is invoked when the new requests are received for an idle disk or when an on-going disk read/write request is completed. Since, the multimedia retrieval requests compete with ordinary delay-tolerant non-real-time requests to the disk, lack of request differentiation results in lack of service guarantees from the storage system. Figure 6 (2) illustrates the new priority based queueing that can rectify this limitation. Specifically, the enhanced queueing mechanism should support multiple job queues with different priorities, each representing a single service class. The job queue with the lowest priority, called the NRTQ, is used for regular non-real-time requests such as those generated by existing `read()` system calls. The jobs for the other queues are generated by continuous media retrievals. The service class specified in read requests for such retrievals decides to which priority queue the jobs are assigned. Note that a multimedia applications can dynamically change its service class or be statically assigned to a fixed service class at the time of stream creation.

Every time an on-going disk request is completed, the driver invokes the job selector which based on the resource allocation policy decides which job to extract from the job queues. The job selector must satisfy following requirements: First, it must ensure that none of the service classes are starved i.e. denied accesses to storage bandwidth for unbounded amount of time. It therefore must employ a resource allocation policy that ensures fair sharing of storage bandwidth among the various priority classes. If at any given instance only jobs of a particular class are present, they must get full storage bandwidth. Second, the job selector must select the requests to processed in such a way that the seek and rotational latencies for disk accesses are minimized.

We decouple these two objectives as follows: We achieve fair resource sharing by using a simple Fair Queueing Algorithm called *Deficit Round Robin* DRR [?]. The original DRR algorithm was proposed in the context of fair sharing of a communication link bandwidth among several data flows, each with its own packet queue. The basic idea in DRR is illustrated in Figure 5. Under DRR, the link services the packet queues in a round robin fashion. However, unlike traditional round-robin schemes, in each round it provides only a fixed *quantum* of service, defined in terms of number of bytes of data to be transmitted in the round, to each queue. When serving a queue in a round, the link allocates a quantum Q to it. If the current value of quantum is less than the size of the packet -k at the head of the queue, the packet is transmitted and the counter is decremented by k. This continues until Q is less than the packet size, when the deficit k bytes of service which was not used in current round is carried over to the next round. Clearly, the quantum size must be set to the maximum size of the packet over all flows. Also, the per flow/queue quantum values need not be identical and if selected different result in weighted DRR fair queueing. The DRR algorithm requires $\bigcirc(1)$ time to process every packet and is simple and inexpensive to implement.

In order to adapt the DRR algorithm for disk driver with multiple queues, we note that disk read/write requests are always in terms of multiples of smallest size block - typically 512 bytes, called a sector. Each job request to the disk driver carries in it's header fields the size of read/write request in terms of the sector size. Therefore. we can define the quantum of service offered to a request queue in terms of number of sectors read in the round. As shown in Figure ??, each queue i is assigned a quantum Q_i, which can be set statically or changed over time to achieve adaptive resource allocation.

In addition to the N priority queue, the DRR implementation for disk driver maintains a work queue (Figure 5. This queue is formed at the start of each service round by removing jobs from the queues until the quantum associated with each queue is exhausted or there are no more jobs left. If a queue does not have any jobs at the instance the queue is formed, it's quantum is not carried over to next round.

The low level disk driver always processes request from the tt work_queue in FIFO fashion and therefore, the work_queue must be sorted using an appropriate disk scheduling algorithm, such as the BSD elevator algorithm or newer scheduling algorithms [25, 28, 32] to minimize seek and rotation latencies in each round.

4.3 Implementation of DRR with two priority queues

We describe a prototype implementation of DRR fair queueing in the current BSD UNIX. In our current implementation, we support two priority classes - a non real-time request queue and a real-time request queue. All the requests generated by the standard `read/write` or `readv/writev` system calls are enqueued in the non-real-time request queue, whereas the requests generated by the new `streamread()` API are queued in the real-time queue.

Following modifications were made to the SCSI driver:

- **Modification on sd_softc data structure:** Figure 7 (a) illustrates the `sd_softc` structure in the existing generic SCSI driver that captures the software state of a SCSI disk. Figure 7 (b) illustrates changes to this structure to support DRR fair queueing. Specifically, we added two new queues - `rtqueue` and the `workqueue`. The new state variables, `rtqnt` and `nrtqnt` keep track of the quantum values for the real-time and non real-time queues. The two other state variables - `max_rtqnt` and `max_nrtqnt` record the maximum amount in terms of number of sectors of data read for each queue in a work round. These variables

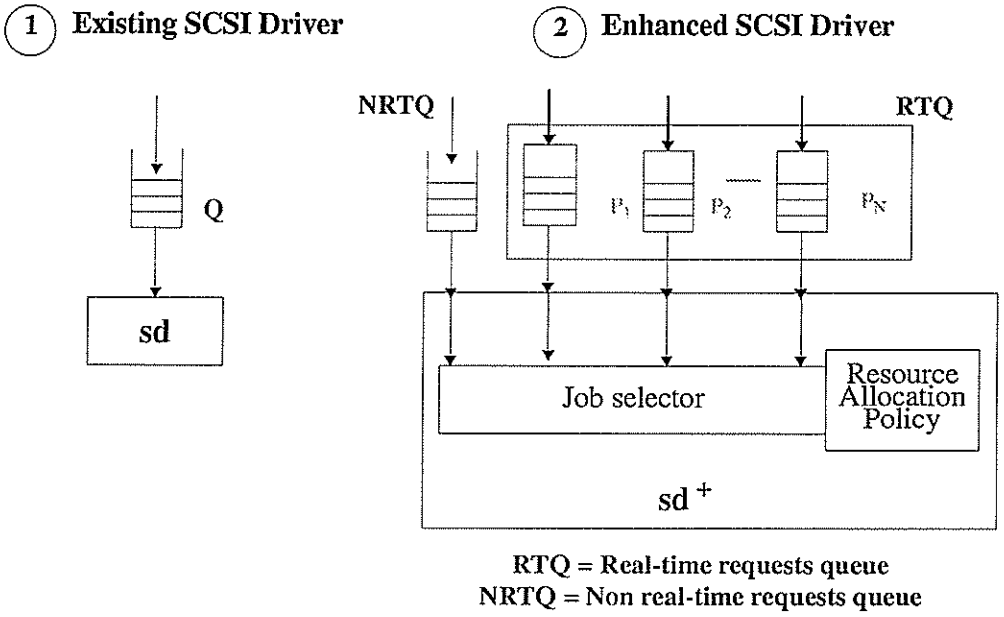


Figure 4: New priority queueing SCSI system

control the resource allocation and delay experienced by requests in each queue. They can be altered by an user level application such as a MOD server or a web server, by using a `ioctl` call.

- **Modified Enqueue procedure:** When a buffer is passed to the driver's enqueue routine `sdstrategy()`, it checks the buffer flags to see if the buffer's B_MMBUF flag is set. If the flag is not set, the buffer is enqueued into the non-real-time queue, otherwise it is inserted into the appropriate the real-time request queue. In presence of multiple priority queues, the buffer carries information about the priority class, which is used to queue it to appropriate queue.

- **Modified dequeue procedure:** In the current SCSI driver, the `sdstart` routine dequeues buffers from the job queue and issues commands to the lower level disk driver. This routine is called when a new request is received or when a request in progress is completed. In our current design, when there is spare capacity in the SCSI adaptor queue, `sdstart` checks for jobs in the work-queue. If there are any requests present, it constructs a SCSI command and sends it to the lower level driver. However, if the work-queue is empty, a `sd_form_workqueue` routine is invoked which uses the the `rtqnt` and `nrtqnt` counters to extract jobs from the `rtqueue` and `nrtqueue`. These jobs are sorted into work queue using the standard `disksort` function. If there are no jobs in the real-time queue and `max_nrtqnt` is set to a small value compared to `max_rtqnt`, the work-queue will formed more often. To minimize, this overhead, the driver can monitor the `rtqueue` occupancy to adapt the quantum allocation.

4.4 Concatenated disk driver (ccd)

The Concatenated Disk Driver (CCD) is a disk striping software developed by Jason Thorpe [30]. It allows one or more 4.4 BSD disks or disk partitions of the same or different sizes to be combined into a single virtual disk or a software disk array. As shown in Figure 8, the data stored on this virtual disk is striped across the component disks and thus, retrievals which are multiple striping units in size result in parallel I/O from multiple disks. The ccd provides near-linear increase in write throughput and sub-linear increase in read throughput as the number of disks in the software disk array is increased. In the following section, we will show that this increased throughput combined with the new mmbuf system provides significant improvement in disk to network data throughput.

Figure 8 (b) shows the interaction between read layers of the mmbuf and the ccd. When an mmbuf is passed to the ccd layer, the B_CALL flag in the mmbuf header is set to ensure the custom `bm_iodone` routine is called in the event the I/O on the virtual disk is complete. The ccd layer splits this buffer into multiple buffers, copies the original buffer header to each buffer and using a striping information table enqueues the requests to the job queues of the component disks. The handler routine for these buffers is set to the function `ccd_iodone()` from the ccd code.

Note that CCD differs from the commercial RAIDs in many ways. Unlike RAIDs which perform striping using a hardware controller, CCD is a software disk array. Other than simple mirroring, CCD does not sup-

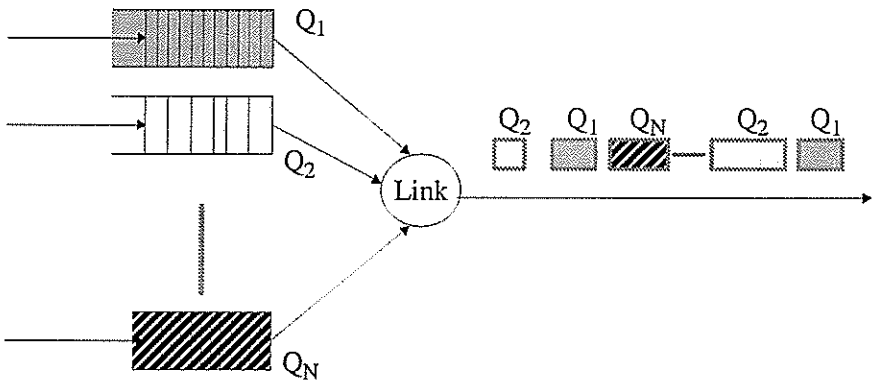


Figure 5: DRR Fair Queueing for a communication link

port any other data redundancy techniques, such as the ones supported by various levels of RAIDs (Level 3,4,5). Also, at present, a CCD can be composed only from component disks or partitions with BSD file systems. We believe that though CCD is sub-optimal, it represents a very simple and cost-effective way to build small disk arrays.

4.5 Periodic CPU access: Real Time Upcall (RTU)

Real Time Upcalls is a novel mechanism designed and implemented within our research group at Washington University [22] to provide guaranteed CPU access to user level and kernel level periodic tasks.

RTUs are an alternative to real-time periodic threads and have advantages such as low implementation complexity, portability, and efficiency. Figure 9 illustrates the basic concept behind RTUs. An RTU is essentially a function in a user program that is invoked periodically in real-time to perform certain activity [22]. Various examples of such activities are protocol processing such as TCP, UDP, multimedia and bulk data processing, and periodic data retrievals from storage systems. The user process employs a new system call-`rtu_create()` — to create an RTU by specifying a function and the period with which it needs to be executed. Two other system calls, namely, `rtu_run()` and `rtu_suspend()`, allow the user process to start and suspend an RTU.

The RTU mechanism is implemented in a manner that does not require any changes to the existing UNIX scheduler implementation. The RTU scheduler is a layer above the UNIX scheduler that decides which RTU (and as a result which process) to run. It uses a variant of the Rate Monotonic (RM) scheduling policy. The main feature of this policy is that there is no asynchronous preemption. The resulting benefits are minimizing expensive context switches, efficient concurrency control, efficient dispatching of upcalls, and elimination of the need for concurrency control between RTUs [22]. Several examples of the effectiveness of RTUs in providing excellent QOS guarantees for media processing and user-level-protocol processing have been reported in [7, 22]. The RTU facility has already been demonstrated to be useful for high performance user level protocol implementations. A more detailed discussion of the implementation and related work in this area such as *Scheduler Activations* [1], *Processor Capacity Reserves*[20], *Q-threads* [16] etc. can be found in [22, 24].

4.6 Streams API

We have designed an API consisting of a new set of system calls that allow applications to access mmbufs and real-time guarantees from the SCSI driver for network destined disk retrievals. A novel feature of these calls is that they allow aggregation of multiple read/send requests for same or different active streams into a single system call, much like a supercall [11]. Such aggregation significantly minimizes system call overheads especially under heavy loads. The streams API interface supports following four main functions:

1. `streamopen(filename,nochains)`:
   The `streamopen()` call, like a traditional `open()` system call opens a file, initializes the file entry structure, installs a pointer to it in the process file descriptor table, and returns the index of it's location. It allocates *nochains* mmbuf chains for prefetching the data from the file and initializes the buffer manager structure that manages these chains. The size of these chains can be dynamically changed.

2. `stream_read()` and `stream_send()`:
   The `stream_read()` call takes a set of descriptors opened by `streamopen` and reads data into associated mmbuf chains. It supports blocking (synchronous), and non-blocking (asynchronous, polled) semantics. The `stream_send()` also takes a set of descriptors and sends the data already read by `stream_read` calls into mmbuf chains. It also appropriately modifies the state of the mmbuf chain to *sending* and then to *empty*. Note that a separate *stream_rdsnd()* call that can combine

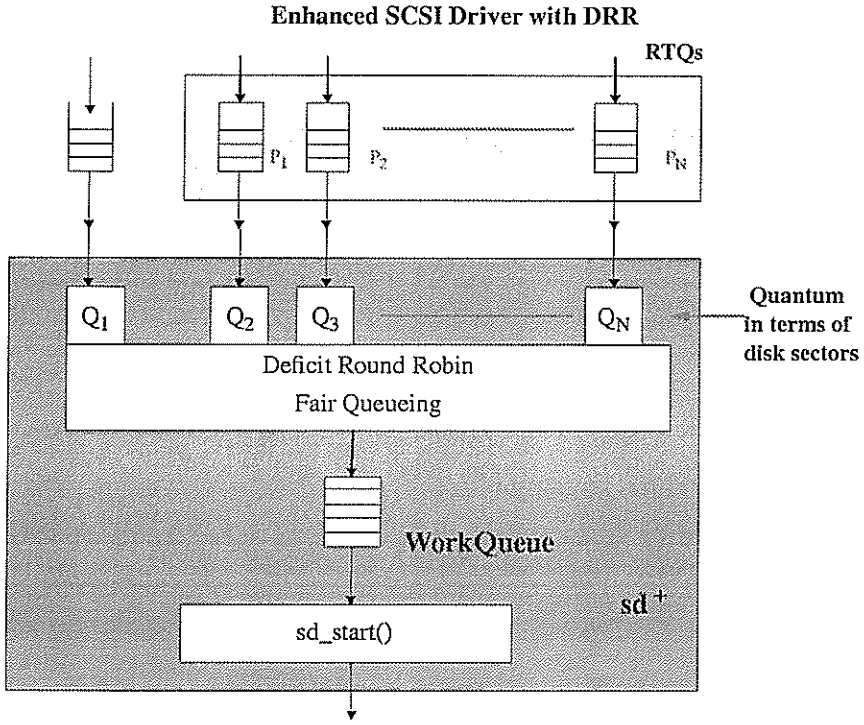


Figure 6: Fair Queueing in SCSI Driver

these two calls is also available to achieve further aggregation.

3. `stream_state()`: Using this system call a user level application can poll the state of the mmbuf chains associated with multiple open stream descriptors on which a stream_read or a stream_send has been invoked.

4. `stream_close()`: This system call closes a stream setup by a previous `stream_open` call. It ensures that any ongoing disk to network I/O is successfully completed before the descriptor and the associated mmbuf chains are released.

We used these system calls in our experiments described in the next section. The design and implementation of these system calls are out of the scope of this report and can be found in [8].

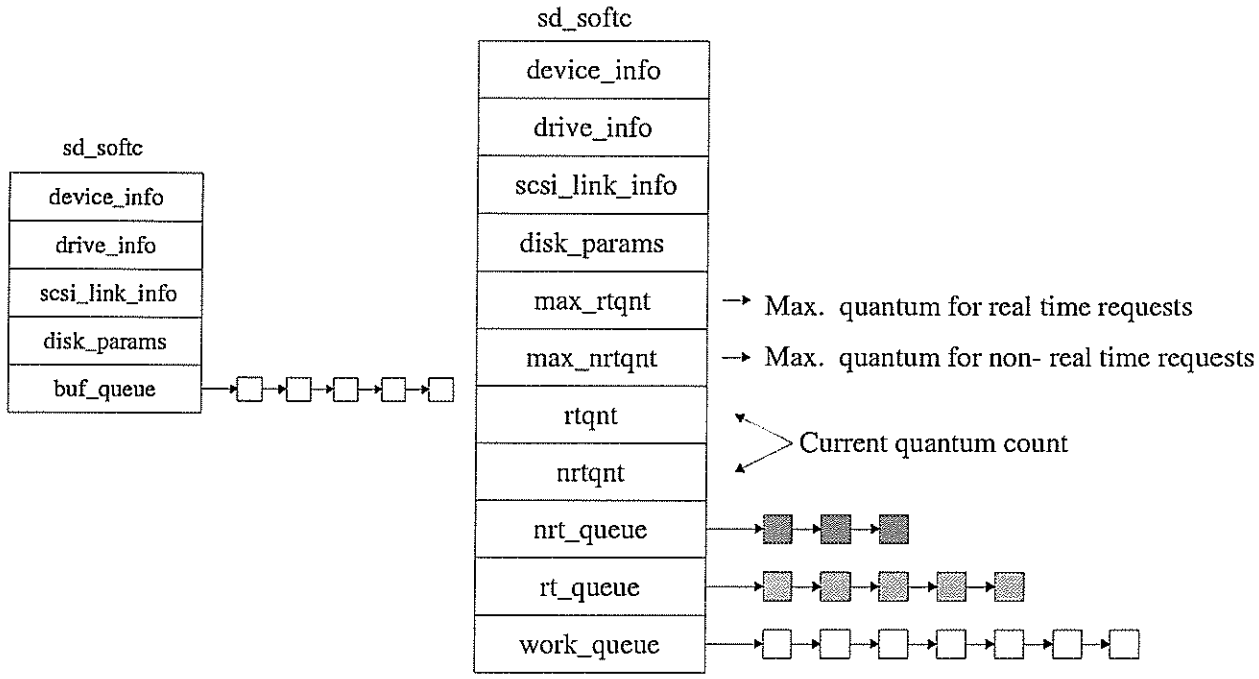
5 Performance Evaluation

In this section, we will describe the experiments carried out to characterize the performance benefits of our solutions. We have successfully implemented the mmbuf system, priority queueing in the SCSI driver and the new stream API (system calls) in the latest release of NetBSD. These enhancements have also been integrated with the CCD driver, the RTU mechanism and a locally developed driver for the ATM interface from Efficient Networks [10]. Also, experimental prototypes of a single node as well as a distributed multinode MARS video server using these enhancements are completed [8].

In all the experiments described here, we used a 200MHz Pentium PC with 128 MB RAM, ENI ATM interface, and an Adaptec dual SCSI AHC-3940, running the enhanced NetBSD 1.2G kernel. We connected two 9 GB Seagate BARACUDDA SCSI disks to the controller. Each disk has a rotational speed of 7200 RPM with internal transfer rate of 80-124 Mbps. The FFS file system created in our measurements used a block size of 8 KB and a fragment size of 1 KB. However, the results reported hold equally well for file system with different values for these parameters.

5.1 Experiment 1: Performance benefits of CCD and mmbufs

The purpose of this experiment is to demonstrate that use of mmbuf and ccds decreases the data retrieval time and improves data throughput from the disk to the network interface card. We created two test programs: Program 1, using `open()` opens a large video file and using the `socket()` call creates a NATM socket associated with an ATM connection. It reads and sends 32 KB chunks of data sequentially using standard read() and write() system calls. We measure the total time to read and send data ranging from 40 MB to 360 MB. The second program, Pro-



(a) Old sd_softc structure

(b) Modified sd_softc structure

Figure 7: Modifications to sd_softc state structure

gram 2, opens the same file as in program 1 using the `streamopen()` call. It also uses the same ATM connection. Using the `streamread()` call, it reads 32 KB of data and sends it using `streamsend()`. Similar to Program 1, we measure total time to complete the read and sends for file sizes varied from 40 to 360 MB. Note that the `streamread()` calls in our measurements are completely blocking and thus, the read semantics are the same as in Program 1. Also, note that in case of standard reads in Program 1, the kernel may perform read-ahead and thus, service some of the reads out of the buffer cache.

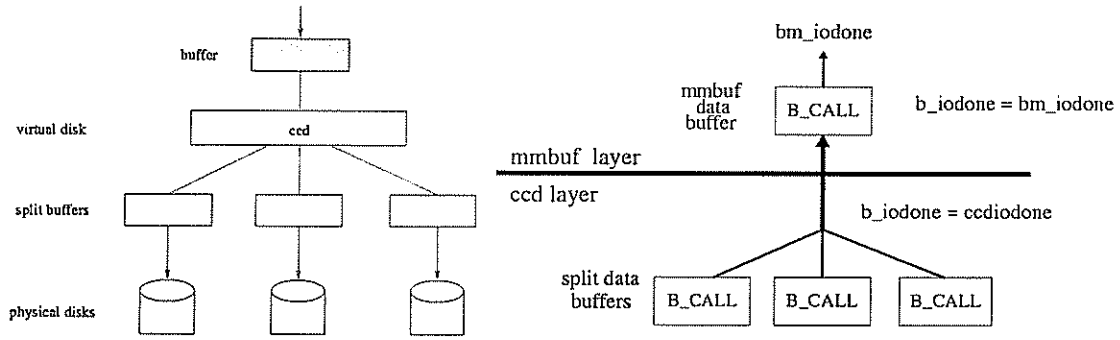
We configured two disks as a ccd device and ran the experiments. In this case, the data throughput for `streamread/send` is 17.65 MBps, which is $\approx$ 10 % faster than normal read/send on ccd, and $\approx$ 60 % faster than stream read/send on sd. The results are shown in Table 1 and Figure 10. We observed consistently lower completion times for `streamread/send` on ccds than for `read/send` and improvements varied from 5 to 12 %. We also measure the time to complete similar tasks in case of a file system on a single disk.

From the experiment results we can see that the data striping implemented by ccd improves data throughput of normal read/send by 47 %, and improves data throughput for stream read/send by 60%. The ccd breaks each read request of 32 KB into 4 8 KB buffers and alternates reads between the two disks. This parallelism in the data reads reduces the disk I/O time for the same amount of data is greatly reduced. Also, asynchronous nature of the streamread/send calls allows multiple outstanding I/Os, which is in contrast with synchronous nature regular read/sends. This I/O pipelining combined with minimization of data copies leads to roughly 12 % throughput improvement.

5.2 Experiment 2: Demonstration of priority to real-time requests from SCSI driver queue

The purpose of this experiment is to demonstrate that the enhanced SCSI driver with priority queueing gives QOS guarantees to real-time requests and provides nearly constant data throughput independent of non-real-time load. All the measurements in this experiment were done on a file system created on a 2-disk ccd as in Experiment 1.

Figure 11 illustrates the experimental set up. We created three test programs: the Program 1 forks n children which serve as background process generating disk I/O load. Each child sequentially reads a separate large file using the `read()` system call . We varied n from 1 to 6. Each of the files is large enough to overflow the system's buffer cache, and therefore disk I/O is always required for all forked processes. We did not perform any measurements for the background pro-



(a) Parallel disk I/O with ccd

(b) Interaction of mmbuf and ccd

Figure 8:

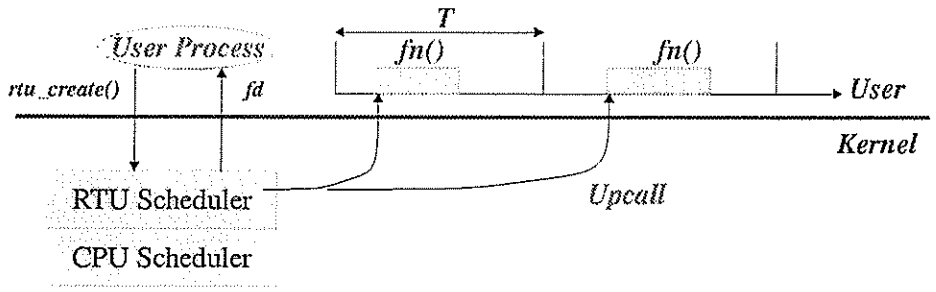


Figure 9: The model for User Level Real Time Upcalls

cesses. The second program, Program 2, is much like a background process, but it also opens a NATM socket and sends the data read from the disk using `send()` call. It reads and sends data in 32 KB blocks for a total of 400 MB of data. The file read here is different from the files background processes read. The third program, Program 3, performs same tasks as Program 2 but uses the stream API.

In PART I of the experiment we run the background processes and Program 2. We measure the total time for Program 2 to complete for different background loads. Similarly, in PART II of the experiment we run the background processes and Program 3. We measure the total time for Program 3 to complete for different background loads.

Table 2 and Figure 11 (b) present the experiment results. We can see that the time it takes for normal read/send in Program 2 increases with the number of background loads. In case of Program 3, the total time remains roughly constant.

Stream read gives better performance than normal read because it gets higher processing priority at the SCSI disk driver. Since the SCSI adaptor can buffer 4 outstanding jobs simultaneously while 1 stream can only file one request at a time, the background jobs are also sent to the SCSI adapter. When there are 4 or more streams reading, the mmbuf jobs queued to the priority queue will occupy most of the adapter's buffer leaving no space for background tasks. Therefore, all the background processes are put to sleep when the streamreads are in progress, in turn ensuring that the throughput for streamreads is nearly constant. This clearly indicates that when there are a large number video and audio retrieval requests within the system, the enhanced priority queueing provides good guarantees. Such guarantees can be enhanced even further by using appropriate resource partitioning schemes.

5.3 Experiment 3: Periodicity of data transfer with RTU

The purpose of this experiment is to demonstrate that by using RTUs and the enhanced file system, a user level process (such as a web server) can obtain excellent QOS guarantees for predictable storage and CPU access.

We created a test program that sets up a single stream and schedules stream read/send within a periodic RTU. Each stream reads data and sends to a different receiver. We ran this process in presence and absence of the CPU intensive background load and measured the number of times deadlines were missed.

We ran three concurrent copies of this test program with single stream and RTU periods of 60, 75, 120 ms. The table illustrates the throughput measured for each stream.

We observed that no deadlines are missed with or without CPU intensive background processes in both parts of the experiment. The applications were able to

Table 1: Read time for normal rd/send and streamrd/send

File Size	`read/send` on sd	`read/send` on ccd	`streamread/send` on
(MB)	normal on sd (sec)	stream on sd (sec)	stream on ccd (sec)
40	4.0	3.0	2.38
120	11.13	8.03	7.01
200	19.01	13.00	11.77
280	26.0	18.50	16.39
360	33.0	22.41	20.38
Throughput	10.90 MBps	16.064 MBps	17.65 MBps

Table 2: Streamrd/send performance under load

Background task	Normal (secs)	4 streamreads (secs)	32 Streamreads (secs)
0	22.0	21.44	20.38
2	60.0	22.44	22.0
4	217.0	22.31	22.0
6	567.0	22.40	21.14

maintain a constant transfer rate using RTU scheduling. Also, in the presence of multiple (schedulable) streams with different periods, each stream got it's share of CPU and storage bandwidth. Without RTU, constant data rate required by video streams from storage and CPU is hard to achieve. An experimental video server which uses the stream API and RTUs, is currently operational in our ATM testbed.

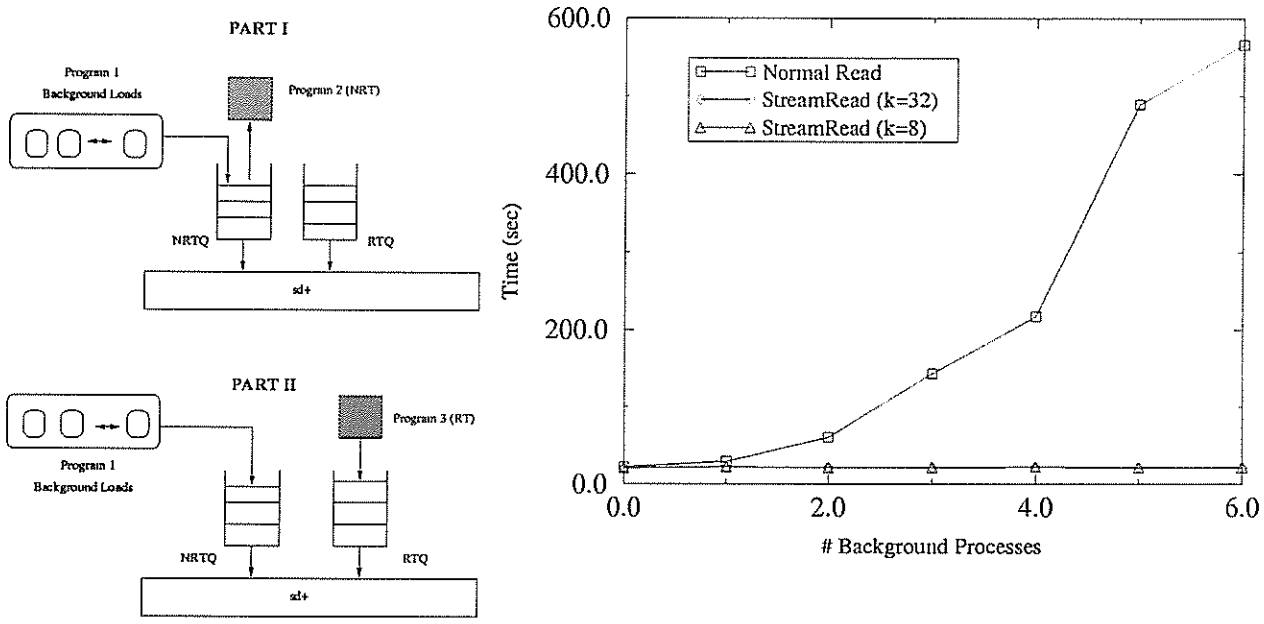
6 Related Work

In the recent past, design of high performance multimedia servers, operating systems, file systems, and specialized disk scheduling techniques for QOS guaranteed multimedia retrieval have been widely researched. Due to space limitation, we are not exhaustive in our coverage of related work in these areas. In the following, however, we try to strike a balance between recent active projects and research widely cited in literature.

The idea of minimizing data copy to achieve higher performance is well known and has been reported in early operating systems such as Tenex[2] and Accent[26]. The *Container Shipping* system [21], the DASH IPC [29] and *fbufs* [12] have addressed the problem of minimizing physical data movement across protection domains in an OS by employing virtual memory re-mapping techniques. However, none of these projects report design of the zero-copy I/O between disks and networks.

A more recent paper by Brustoloni et al. [3] proposes new copy avoidance techniques called *emulated share* and *emulated copy* which do not require any changes to I/O API as required by some of the above mentioned techniques (including ours). Their paper conclusively demonstrates advantages of avoiding copy, data passing and scheduling using a NetBSD UNIX OS enhanced with implementation of their *Genie* I/O system. In this system, the application can specify in a single *Genie* call, invocations to single or multiple I/O modules (such as drivers, protocol stacks or file systems). Also, an application can request multiple invocations in a call to be processed in one of many ways: sequential, parallel, periodic or selective. We believe that the Genie framework can support a zero copy data path between a file system and network protocol stack. However, currently no such design has been reported.

Kevin Fall et al.'s [13, 14] work on providing in-kernel data paths to improve I/O throughput and CPU availability has goals very similar to ours, namely, minimizing data copies and supporting asynchronous and concurrent I/O operations. They have designed and implemented a mechanism called *Splice* in Ultirix 4.2 operating system to meet these goals. Implemented as a system call, `splice()` takes three arguments, two UNIX file I/O descriptors (one specifying the source of data and the other sink) and a integer `sz` size parameter. The `splice()` call arranges within the kernel for `sz` bytes of data to be moved from the source descriptor to sink descriptor without user program intervention. However, this mechanism has several drawbacks: first, the current implementation supports splices between two file descriptors, two socket descriptors or a socket descriptor and a frame buffer. It does not support splice between a socket and file descriptor which would be essential for majority of networked multimedia applications. In fact, such splices can not be supported due to lack of an *mmbuf* like buffering system that can support zero copy data path between storage and network subsystems. Also, once an application invokes the splice call, the data transfer between descriptors is entirely under the control of the kernel and is performed in the software interrupt. Clearly, the lack of ability to control data path makes implementation of application level flow



(a) Setup for Experiment 2

(b) Guarantees provided by priority queuing in sd

Figure 11:

Table 3: Results for Experiment 3 for different RTUs sharing Bandwidth

StreamId	RTU Period (ms)	Throughput (Mbps)
1	60	8.536
2	75	6.824
3	120	4.264

control very difficult. Moreover, the present splice implementation is not available for NetBSD operating systems and hence, can not be easily adopted for our needs.

A more recent and on-going research effort at the Distributed Multimedia Lab at UT Austin, aims to build an integrated file system called *Symphony* [28]. Unlike approaches which use a software integration layer to create a homogeneous abstraction of a single file system out of multiple file systems geared for different data types, *symphony* handles multiple data types in a physically integrated file system. The symphony system supports a QOS aware disk scheduler, a storage manager for data type specific placement policies, a fault tolerance layer, and a two level meta information structure. It also supports admission control, server-push and client-pull service models and data type specific caching. The current implementation runs as a single multi-threaded process in user space and accesses the disks as raw devices. Unfortunately, the implementation details available at the time of writing this paper are sketchy. Some of the key similarities and differences of our work from this project are as follows: like symphony, our work also employs differentiation of disk retrievals into multiple priority classes and provides hooks for implementing suitable disk scheduling policies such as SCAN-EDF, CSCAN,, symphony disk scheduler, or Grouped Sweep Scheduling [25, 28, 32] and associated admission control algorithms. Similar to Symphony, our meta information is two level: the frame level meta info and the traditional UNIX inode information. However, in our design, the frame level info is completely independent of the inode information about the data file and can be potentially stored on different storage and/or file system [8]. Unlike symphony, we follow design advocated in [31] and keep our file system in the kernel. Also, we support an efficient zero-copy data path for network destined storage retrievals. Unlike Symphony, we believe that service models such as client

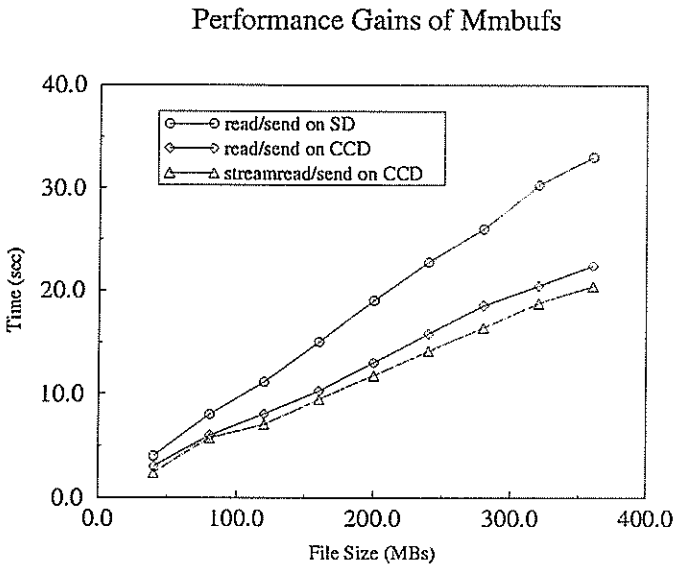


Figure 10: Improvements resulting from use of mmbufs

pull and server push are best implemented in the user space. Also, note that *symphony* has been developed for SOLARIS operating system and exploits the multithreaded nature of the kernel. Our work is entirely based on BSD class of operating systems (NetBSD, FreeBSD, OpenBSD, 4.4 BSD).

Our project MARS, of which the work reported in this paper is a part, has several similarities with the Tiger File system project at Microsoft [4]. Tiger is a distributed, fault-tolerant, real-time file server that uses a distributed storage architecture consisting of *cubs* PCs controlled by a control manager. It stripes constant-bit-rate (CBR) data such as video, and audio over cubs in fixed length data units. The striped data is played back to clients over a broadband network using a "schedule" distributed by the controller via a control network. Unlike our project MARS, the storage nodes ("cubs") in Tiger employ Windows NT operating system enhanced to support zero copy data path between disk and network. However, this data path is different from our mmbuf system and does not use any priority based disk queueing.

7 Conclusions

In this paper, we analyzed the limitations of existing 4.4 BSD UNIX operating system in supporting MOD applications and presented the design of a new *mmbuf* buffering system and an enhanced SCSI driver with support for priority queuing. We also presented experimental results for our enhanced system. Specifically, we showed: (1) A 60 % overall improvement in data throughput from disk device to the network interface, (2) QOS guarantees in the form of periodic accesses from the enhanced SCSI system, and (3) guaranteed access to CPU and storage resources at the user level for applications that employ RTUs and access the new OS enhancements.

Clearly, these measurements indicate that our OS enhancements provide QOS guarantees and significant improvements in throughput on the data-path from the disks to the network interface. The research contributions described in this paper combined with new CPU scheduling mechanism such as RTUs [22] makes 4.4 BSD UNIX a strong candidate for a true multimedia operating system.

Also, note that, since, our work is based on a public-domain operating system, all the OS extensions are freely available (via web or ftp) to interested parties.

References

[1] Anderson, T.E., et.al., "Scheduler Activations: Effective Kernel Support for User Level Management of Parallelism," *ACM Transactions on Computer Systems*, 1992, pp. 53–79.

[2] Bobrow, D., G., "Tenex, A Paged Time Sharing System for PDP-10,", Communications of ACM, Vol. 15, No. 3, pp. 135-143, Mar. 1972.

[3] Brustoloni, J., C., and Steenkiste, P. "Evaluation of Data Passing and Scheduling Avoidance," *Proceedings of NOSSDAV97*, St. Louis, MO, pp. 101-111, May 19-21, 1997.

[4] Bolosky, W., et al., "The Tiger Video Fileserver," *Proceedings of NOSSDAV96*, pp. 97-104, Zushi, Japan, Apr. 23-26, 1996.

[5] Buddhikot, M., Parulkar, G., M., and Cox, Jerome, Jr., "Design of a Large Scale Multimedia Server," *Journal of Computer Networks and ISDN Systems*, Elsevier (North Holland), pp. 504-524, Dec 1994.

[6] Buddhikot, M., and Parulkar, G., M., "Efficient Data Layout, Scheduling and Playout Control in MARS," *ACM/Springer Multimedia Systems Journal*, pp. 199-211, Volume 5, Number 3, 1997.

[7] Buddhikot, M., Parulkar, G., and Gopalakrishnan, R., "Scalable Multimedia-On-Demand via World-Wide-Web (WWW) with QOS Guarantees," *Proceedings of Sixth International Workshop on Network and Operating System Support for Digital Audio and Video, NOSSDAV96*, Zushi, Japan, April 23-26, 1996.

[8] Buddhikot, M., Wu, D., Jane, X., and Parulkar, G., "Project MARS: Experimental Scalable and High performance Multimedia-On-Demand Services and Servers," Washington University, Department of Computer Scince, Technical report (*in preparation*).

[9] Bernhardt, C., and Biersack, E., "A Scalable Video Server: Architecture, Design and Implementation," In *Proceedings of the Real-time Systems Conference*, pp. 63-72, Paris, France, Jan. 1995.

[10] Cranor, C., "BSD ATM," Release Notes, Washington University in St. Louis, Jul 3, 1996.

[11] Cranor, C., and Parulkar, G, "Design of Universal Continuous Media I/O," *Proceedings of NOSSDAV95*, pp 83-86, April 1995.

[12] Druschel, P., and Peterson, L., "Fbufs: A high-bandwidth cross domain transfer facility," *Proceedings of* 14^{th} *SOSP*, pp. 1892-202, Dec. 1993.

[13] Fall, K., and Pasquale, J., "Exploiting In-Kernel Data Paths to Improve I/O Throughput and CPU Availability", *Proceedings of the USENIX Winter Technical Conference*, San Diego, California, January 1993, pp. 327-333.

[14] Fall, K., and Pasquale, J., "Improving Continuous-Media Playback Performance With In-Kernel Data Paths", *Proceedings of the IEEE International Conference on Multimedia Computing and Systems (ICMCS)*, Boston, MA, June 1994, pp. 100-109

[15] Goyal, P., Guo, X., Vin, H.M., "A Hierarchical CPU Scheduler for Multimedia Operating Systems," *2nd Symposium on Operating Systems Design and Implementation (OSDI96)*, Oct. 96, pp. 107-121.

[16] Kawachiya, K., Tokuda, H., "Q-Thread: A New Execution Model for Dynamic QOS Control of Continuous-Media Processing," *NOSSDAV 96*, Japan, April 1996.

[17] Kleiman, S., "Design of vnode interface," *Proceedings of the USENIX Symposium*, 1986.

[18] Khanna, S., et. al., "Real-time Scheduling in SunOS5.0," *USENIX*, Winter 1992, pp.375-390.

[19] McKusik, M., et al. "The Design and Implementation of the 4.4 BSD Operating System," Addison Wesley, 1996.

[20] Mercer, C.W., Savage, S., Tokuda, H., "Processor Capacity Reserves: Operating System Support for Multimedia Applicati ons," *IEEE Intl. Conf. on Multimedia Computing and Systems*, May 1994.

[21] Pasquale, Joseph, Anderson, Eric, and Muller, P. Keith, "Container Shipping: operating system support for i/o intensive applications," *IEEE Computer Magazine*, 27 (3): 84-93, March 1994.

[22] Gopal, R., "Efficient Quality of Service Support in Computer Operating systems for High Speed Networking and Multimedia," *Doctoral Dissertation*, Washington University in St. Louis, Dec. 1996.

[23] Gopalakrishnan, R., Parulkar, G.M., "A Framework for QoS Guarantees for Multimedia Applications within an End-system," *Swiss German Computer Science Society Conf.*, 1995.

[24] Gopalakrishnan, R., Parulkar, G.M., "A Real-time Upcall Facility for Protocol Processing with QOS Guarantees," (Poster) *ACM Symposium on Operating Systems Principles (SOSP)*, Copper Mountain, Colorado, Dec. 1995.

[25] Reddy, A., L., and Wyllie, J."Disk Scheduling Algorithms for Multimedia Operating Systems," *Proceedings of ACM Multimedia'93*, Anaheim, CA, pp. 225-234, Aug. 1993.

[26] Rashid, R., and Robertson, G., "Accent: A Communication-Oriented Network Operating System Kernel,", *Proceedings of* 8^{th} *Symposium on Operating System Principles*, ACM Press, New York, pp. 64-85, 1981.

[27] Shreedhar, M., and Varghese, G. "Efficient Fair Queueing using Deficit Round Robin," *IEEE Transactions on Networking*, 1995.

[28] Shenoy, P., Goyal, P., Rao, S., S., and Vin, H., "Symphony: An Integrated Multimedia File System," Technical Report TR-97-09, Department of Computer Sciences, Univ. of Texas at Austin, March 1997.

[29] Tzou, Shin-Yan and Anderson, David, "The performance of message-passing using restricted virtual memory re-mapping," *Software - Practice and Experience*, 21(3): 251-267, March 1991.

[30] Thorpe, Jason, *Personal Communication*, March 1996.

[31] Welch, B, "The File System Belongs to the kernel?", *Proceedings of the* 2^{nd} USENIX *Mach symposium*, Nov 20-22, pp. 233-250, 1991.

[32] Yu, P., Chen., M., and Kandlur, D., "Grouped Sweeping Scheduling for DASD based Storage Management," Multimedia Systems, Springer-Verlag, pp. 99-109, Dec. 1993.