Distributed Schedule Management in the Tiger Video Fileserver

by William J. Bolosky, Robert P. Fitzgerald and John R. Douceur
Microsoft Research
bolosky@microsoft.com

Abstract

Tiger is a scalable, fault-tolerant video file server constructed from a collection of computers connected by a switched network. All content files are striped across all of the computers and disks in a Tiger system. In order to prevent conflicts for a particular resource between two viewers, Tiger schedules viewers so that they do not require access to the same resource at the same time. In the abstract, there is a single, global schedule that describes all of the viewers in the system. In practice, the schedule is distributed among all of the computers in the system, each of which has a possibly partially inconsistent view of a subset of the schedule. By using such a relaxed consistency model for the schedule, Tiger achieves scalability and fault tolerance while still providing the consistent, coordinated service required by viewers.

1. Introduction

In the past few years, relatively inexpensive computers, disk drives, network interfaces and network switches have become sufficiently powerful to handle high quality video data. Exploiting this capability requires solutions to a number of different problems such as providing the requisite quality of service in the network, handling time sensitive data at the clients and building servers to handle the real-time, storage and aggregate bandwidth requirements of a video server. Researchers have attacked various facets of these problems, coming up with many creative solutions. We built a video server, choosing to use a distributed system structure. In building this server, we were faced with having to control the system in a scalable, fault tolerant manner. This paper describes our solution to this control problem.

Tiger [Bolosky96], the technology underlying the Microsoft® Netshow™ Professional Video Server, is a video fileserver intended to supply digital video data on demand to up to tens of thousands of users simultaneously. Tiger must supply each of these viewers with a data stream that is independent of all other viewers; multiplexing or "near video-on-demand" does not meet Tiger's requirements. The key observations driving the Tiger design are that the data rate of a single video stream is small relative to the I/O bandwidth of personal computers, and that I/O and switching bandwidth is cheaper in personal computers and network switches than in large computer memory systems and backplanes. Tiger is organized as a collection of machines connected together with an ATM (or other type of) switched network. While this distributed organization reduces the hardware cost per stream of video and improves scalability over monolithic designs, it introduces a host of problems related to controlling the system.

The data in a Tiger file is striped across all of the computers and all of the disks within the system. When a client (viewer) wants to play a particular file, Tiger must assure that the system capacity exists to supply the data to the viewer without violating guarantees made to viewers already receiving service. In order to keep these commitments, Tiger maintains a schedule of viewers that are playing files. The size of this schedule is proportional to the total capacity of the system, and so central management of the schedule would not arbitrarily scale. In order to remove a single point of failure and to improve scalability, the schedule is implemented in a distributed fashion across the computers comprising the Tiger server. Each of the computers has a potentially out-of-date view of part of the schedule (and no view at all of the rest), and uses a fault- and latency-tolerant protocol to update these views. Based on their views, the computers take action to send the required data to the viewers at the proper time, to add and remove viewers from the schedule, and to compensate for the failure of system components.

Tiger behaves as if there is a single, consistent, global schedule. For reasons of scalability and fault tolerance, the schedule does not exist in that form. Rather, each of the component computers acts as if the global schedule exists, but a component computer only has a partial, possibly out-of-date view of it. Because the component computers are acting based on a non-existent global schedule, we call the global schedule a *hallucination*. Because many component computers share a common hallucination, we say that the hallucination is *coherent*. The coherent hallucination model is a particularly powerful one for distributed protocol design, because it allows the designer to split the problem into two parts: generating a correct centralized abstraction, and creating appropriate local views of that abstraction.

The remainder of this paper is organized in four major sections. The first describes the basic design of Tiger, including the hardware organization, data layout and fault tolerance aspects of the system; necessary background to understand the functioning of the schedule. The next two sections describe the Tiger schedule, the first treating the schedule as a single, centralized object, and second covering its distributed implementation. The final major section presents performance results showing a modest sized Tiger system that scales linearly. The paper wraps up with a related work section and conclusions.

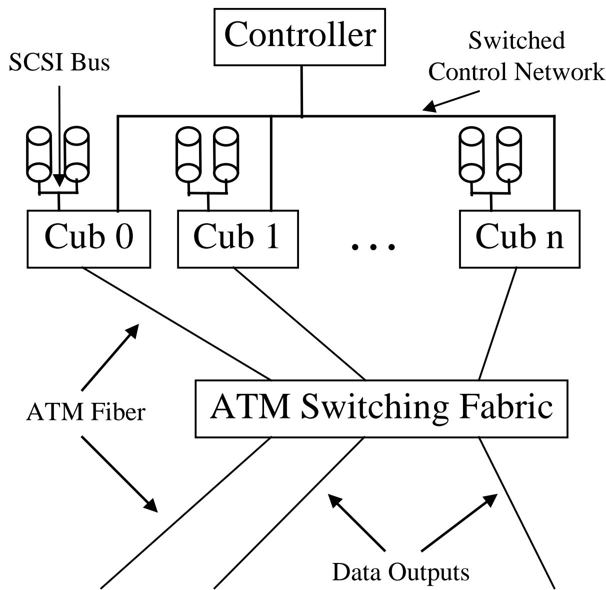

Figure 1: Typical Tiger Hardware Organization

2. The Tiger Architecture

2.1 Tiger Hardware Organization

A Tiger system is made up of a single Tiger controller machine, a set of identically configured machines – called "cubs" – to hold content, and a switched network connecting the cubs and the clients. The Tiger controller serves only as a contact point (*i.e*., an IP address) for clients, the system clock master, and a few other low effort tasks; if it became an impediment to scalability, distributing these tasks would not be difficult. Each of the cubs hosts a number of disks, which are used to store the file content. The cubs have one or more interfaces to the switched network to send file data and to communicate with other cubs, and some sort of connection to the controller, possibly using the switched network or using a separate network such as an ethernet. The switched network itself is typically an ATM network, but may be built of any scalable networking infrastructure. The network topology may be complex, but to simplify discussion we assume that it is a single ATM switch of sufficient bandwidth to carry all necessary traffic.

For the purposes of this paper, the important property of Tiger's hardware arrangement is that the cubs are connected to one another through the switched network, so the total bandwidth available to communicate between cubs grows as the system capacity grows (although the bandwidth to and from any particular cub stays constant regardless of the system size).

2.2 File Data Layout

Every file is striped across every disk and every cub in a Tiger system, provided that the file is large enough. Tiger numbers its disks in cub-minor order: Disk 0 is on cub 0, disk 1 is on cub 1, disk n is on cub 0, disk n+1 is on cub 1 and so forth, assuming that there are n cubs in the system. Files are broken up into blocks, which are pieces of equal duration. For each file, a starting disk is selected in some manner, the first block of the file is placed on that disk, the next block is placed on the succeeding disk and so on, until the highest numbered disk is reached. At that point, Tiger places the next block on disk 0, and the process continues for the rest of the file. The duration of a block is called the "block play time," and is typically around one second for systems configured to handle video rate (1-10Mbit/s) files. The block play time is the same for every file in a particular Tiger system.

Tiger uses this striping layout in order to handle imbalances in demand for particular files. Because each file has blocks on every disk and every server, over the course of playing a file the load is distributed among all of the system components. Thus, the system will not overload even if all of the viewers request the same file, assuming that they are equitemporally spaced. If they are not, Tiger will delay starting streams in order to enforce equitemporal spacing.

One disadvantage of striping across all disks is that changing the system configuration by adding or removing cubs and/or disks requires changing the layout of all of the files and all of the disks. Tiger includes software to update (or "re-stripe") from one configuration to another. Because of the switched network between the cubs, the time to restripe a system does not depend on the size of the system, but only on the size and speed of the cubs and their disks.

This paper describes two different versions of the Tiger system, called "single bit rate" and "multiple bit rate." The single bitrate version allocates all files as if they are the same bitrate, and wastes capacity on files that are slower than this maximum configured speed. The multiple bitrate version is more efficient in dealing with files of differing speed. Because the block play time of every file in a Tiger system is the same as that of any other file, all blocks are of the same size in a single bitrate server (and files of less than the configured maximum bitrate suffer internal fragmentation in their blocks). In a multiple bitrate server block sizes are proportional to the file bitrate. Tiger stores each block contiguously on disk in order to minimize seeks and to have predictable block read performance. Tiger DMA's blocks directly into pre-allocated buffers, avoiding any copies. Tiger's network code also DMA's directly out of these buffers, resulting in a zero-copy disk-to-network data path.

2.3 Fault Tolerance

One goal of Tiger is to tolerate the failure of any single disk or cub within the system with no ongoing degradation of service. Tiger does not tolerate complete failure of the switched network or of the Tiger controller. There are several challenges involved in providing this level of fault tolerance. The first is to be sure that the file data remains available even when the hardware holding it has failed. The second is assuring that the additional load placed on the non-failed components does not cause them to overload or hotspot. A final challenge is to detect failures and reconfigure the responsibilities of the surviving components to cope with the loss. This section describes our answers to the first challenge. Maintaining the schedule across failures is covered in section 4. Detecting faults is accomplished by a deadman protocol that runs between the cubs.

While the Tiger controller is a single point of failure in the current implementation, the distributed schedule work described in this paper removes the major function that the controller in a centralized Tiger system would have. The Netshow™ product group plans on making the remaining functions of the controller fault tolerant. When they have completed this task, the fault tolerance aspects of the distributed schedule will have come to full

	Disk 0	Disk 1	Disk 2
Rim (fast)	Primary 0	Primary 1	Primary 2
Middle	Secondary 2.0	Secondary 0.0	Secondary 1.0
Hub (slow)	Secondary 1.1	Secondary 2.1	Secondary 0.1

Figure 2: Tiger Disk Data Layout

fruition. Until that task is complete, distributed scheduling is interesting primarily for scalability and academic interest.

Tiger uses mirroring to achieve data availability. At first glance, this might seem like an odd choice compared to using RAID-like parity striping [Patterson88]. The combination of two factors led us to choose mirroring. First, we expect bandwidth, rather than storage capacity, to be the limiting factor in Tiger systems. Second, the requirement to survive failures not only of disks but also of entire machines means that if Tiger used parity encoding it would need to move almost all of the file data for a parity stripe between machines in order to reconstruct the lost data. Furthermore, this movement would have to happen prior to the time that the lost data would normally be sent. The cost of the inter-machine bandwidth and buffer memory for such a solution is large compared to the cost of mirroring.

While mirroring requires that each data bit be stored twice, it does not necessarily mean that half of the bandwidth of a disk or machine needs to be reserved for failed-mode operation. Tiger *declusters* its mirror data. That is, for each block of primary data stored on a cub, its mirror (secondary) copy is split into several pieces and spread across different disks and machines. The number of pieces into which the blocks are split is called the *decluster factor*. Tiger always stores the secondary parts of a block on the disks immediately following the disk holding the primary copy of the block.

Because of declustering, when a single disk or machine fails several other disks and machines combine to do its work. The tradeoff in the choice of decluster factor is between reserving bandwidth for failed mode operation and decreased fault tolerance. With a decluster factor of 4, only a fifth of total disk and network bandwidth needs to be reserved for failed mode operation, but a second failure on any of 8 machines would result in the loss of data.[1] Conversely, a decluster factor of 2 consumes a third of system bandwidth for fault tolerance, but can survive failures more than two cubs away from any other failure.

Even if Tiger suffers the failure of two cubs near to one another, it will attempt to continue to send streams, although these streams will necessarily miss some blocks of data. If two or more consecutive cubs are failed, the preceding living cub will send scheduling information to the succeeding living cub, bridging the gap.

Figure 2 illustrates Tiger's data layout for a three disk, decluster factor 2 system. The notation "Secondary m.n" means that part n of each block in primary m is stored at the indicated place. Because the outer tracks of a disk are longer than the inner ones, modern disk drives store more sectors on these outer tracks. Disks have constant angular velocity, so the outer tracks pass under the drive head in the same amount of time as the inner tracks. As a result, disks are faster on the outer tracks than on the inner ones [Ruemmler94; Van Meter97]. Tiger takes advantage of this fact in its data layout. Primaries are stored on the faster portion of a disk, and secondaries are stored on the slower part. At any one time a disk can be covering for at most one failed disk, so for every primary read there will be at most one secondary read. The primary reads are decluster times bigger than the secondary reads, so Tiger can rely on the fact that at most 1 / (decluster + 1) of the data will be read from the slower half of the disk.

3. The Tiger Schedule

Over a sufficiently large period of time, a Tiger viewer's load is spread evenly across all components of a Tiger system. However, in the short term Tiger needs to assure that there are no hotspots. A hotspot occurs when a disk or cub is asked to do more work than it is capable of doing over some small period of time. Because the block play time is the same for all files and all files are laid out in the same order, viewers move from cub to cub and disk to disk in lockstep; alternately, this can be viewed as the disks and cubs moving along the schedule in lockstep. A system that has no hotspots at any particular time will continue to have no hotspots unless another viewer starts playing. Thus, the problem of preventing hotspots is reduced to not starting a viewer in such a way as to create a new hotspot.

Tiger uses the schedule both for describing the needed work to supply data to running viewers and for checking whether starting a new viewer would create a hotspot. If there is a viewer

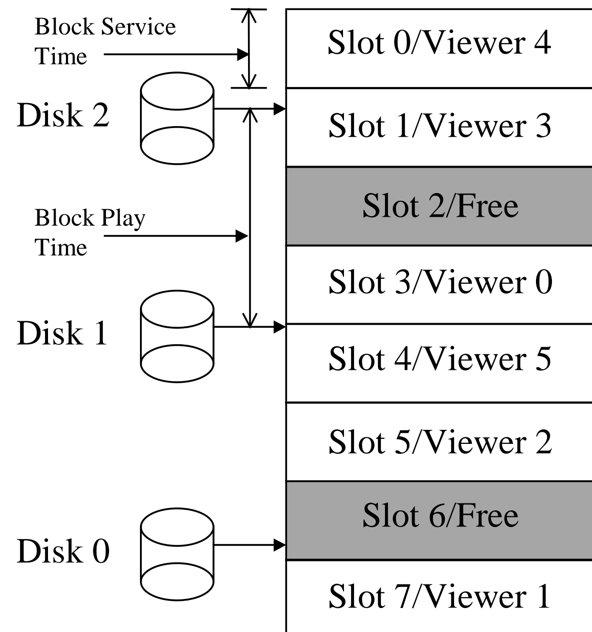

Figure 3: Example Disk Schedule

[1] In a decluster 4 system the 4 disks before the failed disk need to be alive because the failed disk's mirror area holds some of the secondary copy of their data, and the 4 disks after the failed disks need to be alive because they hold the secondaries for the failed disk.

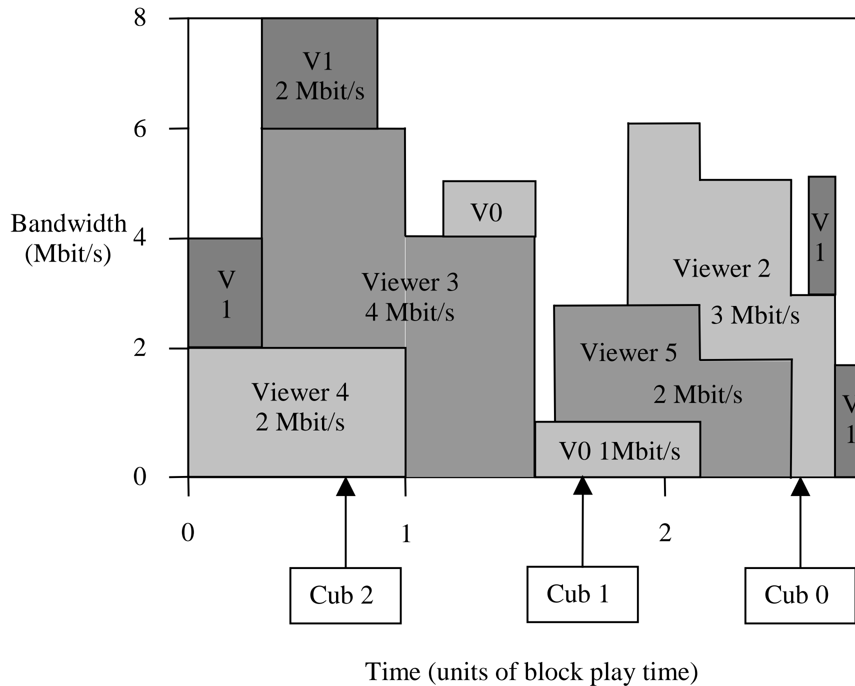

Figure 4: Example Network Schedule

who requests service, and whose request would create a hotspot, the system will delay starting the viewer until it can be done safely. This scheme gives variable delays for initial service, but guarantees that once a viewer is started there will be no resource conflicts. [Bolosky96] discusses the duration of the delays introduced, and concludes that for reasonable system parameters and restricted to running at 80-90% of capacity the delays are acceptable for most purposes. Section 5 contains measurements that support this conclusion for the particular Tiger system described there.

3.1 The Disk Schedule

In a single bitrate Tiger, the system maintains a schedule describing the work done by the disk drives. Because drives perform best when doing large transfers (amortizing a seek over a large amount of data to be read), Tiger reads each block in a single chunk. The disk schedule is an array of slots, with one slot for every stream of system capacity. One can think of the disk schedule as being indexed by time rather than by slot number. The time that it takes to process one block (the block play time divided by the maximum number of streams per disk) is called the *block service time*. This time is determined by either the speed of the disks or the capacity of the network interface, whichever is the bottleneck. So, each slot in the disk schedule is one block service time long, and the entire schedule is the block play time times the number of disks in the system. The schedule must be an integral multiple of both the block play and block service times. If not, the block service time is lengthened enough to make it so. This requirement is equivalent to saying that a Tiger system (but not a disk, cub or network card) must source an integral number of streams, and that the actual hardware capacity of the system as a whole is rounded down to the nearest stream.

Each cub maintains a pointer into the schedule for each disk on the cub. These pointers move along the schedule in real time. When the pointer for a particular disk reaches the beginning of a slot in the schedule, the cub will start sending to the network the appropriate block for the viewer occupying the schedule slot. In order to allow time for the disk operations to complete, the disks run at least one block service time ahead of the schedule. Usually, they run a little earlier, trading off buffer usage to cover for slight variations in disk and I/O system performance. The pointer for each disk is one block play time behind the pointer for its predecessor. Because of the requirement that the total schedule length is the block play time times the number of disks, the distance between the last and the first disk is also one block play time.

If a Tiger system is configured to be fault tolerant, the block service time is increased to allow for processing the secondary load that will be present in a failed state. If the disk rather than the network is the limiting factor the inside/outside disk optimization described in section 2.3 is taken into account when determining how big to make the block service time.

3.2 The Network Schedule: Supporting Multiple Bitrates

This section describes support for scheduling streams of differing bitrates on a Tiger system. The discussion is offered primarily because it serves to illustrate a particular difficulty (and its solution) in distributing the schedule. Multiple bitrate scheduling is only partially implemented in today's Tiger systems.

The concept of block service time as described in section 3.1 has a number of underlying assumptions. One is that the block service time is the same for all blocks in all files, which is true only in a single bitrate system. A slightly more subtle assumption is that the ratio of disk usage to network usage is constant for all blocks. This assumption is necessary because the block service time is chosen so the most heavily used resource is not overloaded. In a multiple bitrate system, blocks of different files

may have different sizes. The time to read a block from a disk includes a constant seek overhead, while the time to send one to the network does not, so small blocks use proportionally more disk than network. Consequently, in a multiple bitrate Tiger system whether the network or disk limits performance may depend on the current set of playing files. Different parts of the same schedule may have different limiting factors.

Because a combined schedule cannot work for a system where block sizes vary from stream to stream, multiple bitrate Tiger systems implement a second schedule that describes the activity on the network, called a *network schedule*. Unlike disks, networks interleave all of the streams being sent. Because networks process several streams simultaneously, the network schedule is a two dimensional structure. The *x*-axis is time and the *y*-axis bandwidth. The overall length of the schedule is the block play time times the number of cubs[2], while the height is the bandwidth of a cub's network interface cards (NICs). The length of an entry in the network schedule is one block play time, and the height is determined by the bitrate of the stream being serviced.

Figure 4 shows a network schedule constructed by assigning bitrates to the viewers shown in the schedule in Figure 3. Each viewer is represented by a block of a certain color. For example, viewer 4 runs at 2 Mbit/s from time 0 to time 1, and viewer 0 runs at 3 Mbit/s from time 1.125 to 2.125. A vertical slice up from the pointer for a cub shows what that cub's NIC is doing at the current time, so cub 2 is most of the way through sending its block for viewer 1, a little farther from the end of viewer 4's block and about a third of the way into viewer 3's block. As time advances, the cubs move from left to right through the schedule, wrapping around at the end. In one block play time cub 0 will be at exactly the same position that cub 2 occupies in the figure. The total height of entries at any point in the schedule shows the instantaneous load on the NICs when servicing that part of the schedule.

A multiple bitrate Tiger system not only needs to assure that its NICs aren't overrun, it also has to assure that disk bandwidth isn't exceeded. Keeping a schedule similar to the one used for the single bitrate system but with variable size slots is sufficient but not necessary. The disk schedule in the single bitrate Tiger not only avoids hotspots, it specifies the time at which each block must be sent to the network. In the multiple bitrate system the network schedule serves this function. Therefore, the specific time ordering information in the disk schedule is not necessary in the multiple bitrate system; entries in the disk schedule are free to move around, as long as they're completed before they're due at the network. Because of this reordering property, fragmentation does not occur in the disk schedule.

Fragmentation can be a problem in the network schedule. Consider the schedule shown in figure 4. The free bandwidth below the 6 Mbit/s level between when viewer 4 finishes sending and when viewer 2 starts is unusable, because any new entry would be one block play time long, and the gap in the schedule is slightly too short. In general, fragmentation can become fairly severe if viewers are started at arbitrary points. We have found that fragmentation is reduced to an acceptable level when viewers are forced to start at times that are integral multiples of the block play time divided by the decluster factor.

[2] This is different from the disk schedule, whose length is the block play time times the number of disks. The difference is because the output of all of the disks on a cub are sent to the network through the same NICs.

3.3 Scalability Considerations

The question of whether it makes sense to distribute Tiger's schedule management depends on how large Tigers can grow and how much work would be involved in central management of the schedule. This section explores a limit on the size of Tigers (which is probably not the limiting factor in the current implementation), and considers the work involved in centrally maintaining that large of a schedule.

A fundamental limit of scalability in a Tiger system is the number of different disks that hold a particular file. A typical movie is about 100 minutes (6000 seconds) in length. If a block is 1 second long and disks are the same speed as the ones used in the experiment in section 5 (which can serve 10.75 streams each), using 6000 disks to store a movie would mean that a single copy of a movie could serve over 64,000 streams. In practice, we would expect to not build systems quite this large, because serving the full 64,000 viewers would require that they be evenly spaced over 100 minutes. Still, with better disk technology it is not hard to imagine Tiger systems with as many as 30,000 to 40,000 streams. Such a system would have on the order of 1000 cubs.

In a centrally scheduled system, the controller would have to track the entire schedule. Even with 40,000 streams, just keeping up with the schedule is quite possible with a reasonable computer. However, the controller would also have to communicate the schedule to the cubs. If the message that the controller sends instructing a cub to deliver a block to a viewer is 100 bytes long (which is about the size of the comparable message sent from cub to cub in the distributed system), the controller would have to maintain a send rate of 3-4 Mbytes/s of control traffic through the TCP stack to the roughly 1000 cubs. Reliable and timely transmission of this much data through TCP, particularly to that many destinations, is probably beyond the capability of the class of personal computers used to construct a Tiger system.

In addition to making control scalability easier, distributing the schedule also eliminates the most complex aspect of having the central controller as a single point of failure. Making its remaining functions fault tolerant is a simple exercise, and will be completed by the product team. We chose to distribute schedule management because of the combination of the fault tolerance and scalability advantages.

4. Distributed Schedule Management

Consider the descriptions of the Tiger schedules in section 3. They are worded as if there is a single disk or network schedule for the entire Tiger system. Conceptually, this is true. In practice, the schedule management is distributed among the cubs. Each cub has partial (and possibly incorrect) knowledge of the global schedule, but behaves as if the entire schedule exists. The net result is a system that as a whole acts as if there were a global schedule, but which is scalable and fault tolerant.

We use the term *coherent hallucination* to mean a distributed implementation of a shared object, when there is no physical instantiation of the object. The Tiger schedule is a coherent hallucination because no particular machine holds a copy of the entire schedule, but yet each behaves as if there is a single, coherent global schedule.

There are two major components to a coherent hallucination. The first is the imaginary centralized abstraction, the "hallucination." Second is the concept of a *view*. A view is the picture that a participant in a coherent hallucination-based system has of the hallucination. Views may be incomplete or out-of-date

without compromising the coherence of the underlying hallucination. The complexity in implementing a system using coherent hallucinations lies in managing the views and in taking action based on them. A necessary but insufficient condition for scalability is that participants' views be limited to a size that does not grow as a function of the scale of the system. Fault tolerance requires that every part of the hallucination is contained in more than one view, or can be reconstructed using only data from views available after a failure.

4.1 Distributing the Disk Schedule

There are two main alternatives in distributing the disk schedule. The first is to have each cub keep a complete but mostly out-of-date copy of the schedule. In this scheme, a cub would learn that a particular viewer was in a particular slot and would send the appropriate blocks to the viewer until the viewer requests stop or hits end-of-file. The second alternative is to have each cub keep track of only the portion of the schedule that's near where its disks are processing and to propagate the schedule information around the ring of cubs at the same rate that the cubs move through the schedule. While the second alternative requires more communication between cubs, we chose it because it does not require each cub to keep track of a schedule whose size is proportional to the size of the system as a whole, and so scales better. It also requires less work to remove viewers from the schedule.

The remainder of this section describes how the schedule is maintained among the cubs. Section 4.1.1 covers steady state operations, in which no viewers are entering or leaving the schedule. Section 4.1.2 describes the stop play operation and 4.1.3 explains starting a viewer. Section 4.2 describes the network schedule in the multiple bitrate Tiger, and 4.3 covers lessons learned.

4.1.1 Propagating Schedule Information in Steady State

Every cub maintains a view of the portion of the disk schedule near each of its disks. That is, it keeps track of the schedule entries that the disk will encounter in the next few seconds, as well as a little while into the past. From time to time it forwards entries to the next cub in line. When a cub sends the contents of a schedule entry to the next cub, it does so by sending a *viewer state record* (or just "viewer state"). A viewer state contains the address of the viewer, the file being played, the viewer's position in the file, the schedule slot number, the play sequence number (how far the viewer has gotten into the current play request), and some other bookkeeping information.

The amount of time between when a viewer state arrives at a cub and when that cub's block for that viewer is due at the network is called the *lead time* of the viewer state. Two global system parameters, `minVStateLead` and `maxVStateLead` control the cubs' management of viewer state forwarding. Cubs endeavor to keep the schedule updated at least `minVStateLead` into the future, while never forwarding viewer states more than `maxVStateLead` ahead of the schedule. Typical values are 4 and 9 seconds, respectively. Maintaining a certain minimum lead time allows the cubs to tolerate some variability in communication latency, as well as allowing them to start disk I/O early and thus tolerate variable disk performance. Limiting the maximum lead time to a constant guarantees that the amount of schedule information that a cub needs to keep does not depend on the size of the system. Having a gap in between them allows the cubs to group viewer states together into a single network message before forwarding them, and so reduce communications overhead.

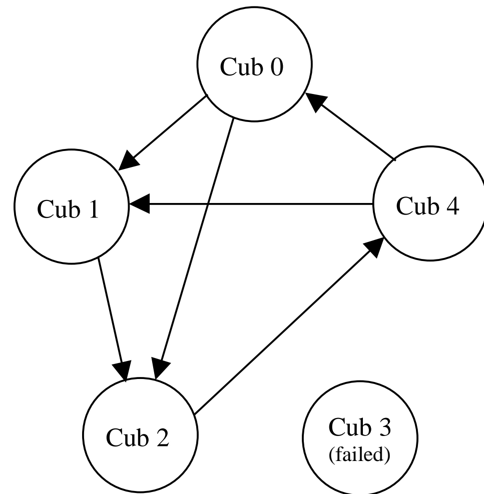

Figure 5: Viewer State Propagation Around the Ring of Cubs

Because Tiger tolerates the failure of cubs, it must ensure that schedule information is not lost when a cub crashes. As shown in Figure 5, each time a cub forwards a viewer state it sends it not only to the cub's successor, but also to the second successor. Receiving a viewer state is idempotent: Duplicates are ignored. Thus, at least two cubs are aware of each schedule entry and if any single cub fails, some other cub will be sure that the schedule information continues to propagate. In the figure, cub 3 is failed, and neither sends nor receives any messages. Double forwarding means that cub 2 sends its information on to cub 4, so the loss of cub 3 does not result in any loss of schedule information.

We could have chosen to forward viewer states only once, to the next living cub in the ring. This would have halved the number of viewer states sent between cubs, and possibly removed the necessity of ignoring duplicate viewer states. We chose not to do this because cub failure detection is timeout based, and so involves a certain amount of latency. Under the single forwarding model any time a cub failed the other cubs would have to go back, figure out what schedule information had been lost and recreate it. Furthermore, between the failure and the detection, not only would the data stored on the failed cub be lost, but so also would the data from the subsequent cubs that never received the viewer states. To us, the additional data loss and difficulty in getting a single forwarding protocol right was worse than incurring the cost of doubly forwarding viewer states.

When a cub sees a schedule entry, that entry tells the cub it should send a particular block of a file to a certain viewer at a specified time. It does not tell the cub where on its disk to find the block. Each cub keeps track of the contents of the primary region of its disks, indexed by file and block numbers. Index entries are 64 bits long. Unlike traditional filesystems, the index is stored in the cub's memory rather than on the data disks. Three factors contributed to this decision: The large block size means that there are relatively few blocks per disk and so relatively little metadata, the cost of the seek to do the metadata read is unacceptably large, and the fact that the metadata read needs to complete before the main block read begins would add latency into requests to start playing.

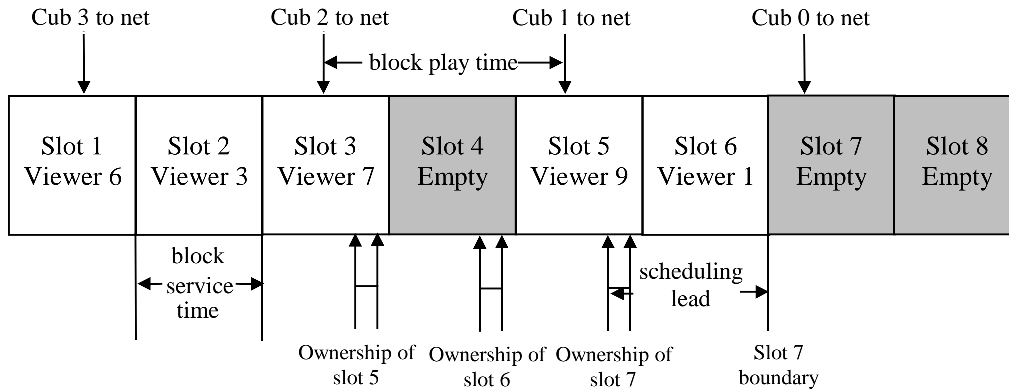

Figure 6: Ownership of Schedule Slots

When a cub or disk is failed, Tiger needs to have the cubs holding the pieces of the secondary copy of the data send their bits to the user. The decision to send this data is made by the cub succeeding the failed component. When the succeeding cub makes this decision, it creates a special kind of viewer state called a *mirror viewer state*. Mirror viewer states are much like normal ones, except that they describe mirror schedule entries and they have different timing requirements. When a block needs to be sent from the mirror copy, each piece of the mirror is separated in time from the previous piece by (block play time/decluster), rather than by (block play time) as is the case with normal viewer states. The cubs take these timing differences into consideration when deciding when to forward a mirror viewer state, and try to keep them between minVStateLead and maxVStateLead ahead of the operation they describe in the same way as normal viewer states.

4.1.2 Removing Viewers from the Schedule

Viewers leave the schedule in two different ways. They can reach end-of-file or request "stop playing." Handling end-of-file is straightforward. Stop playing requests require tracking down recorded schedule information and killing it.

In order to abort playing a file, a viewer sends a request (called a *deschedule* request) to the Tiger controller. The controller determines from which cub the viewer is receiving data, and forwards the request on to that cub and its successor. Much like viewer states, deschedule requests are idempotent. When a cub receives a particular deschedule request for the first time, it removes any schedule entries for the viewer being descheduled, forwards the deschedule request on to its successor and second successor, and remembers the deschedule. Because of variable communication latency and multiple path message propagation, it is possible for a cub to receive a copy of a viewer state after it has received a deschedule for that viewer. Before accepting a viewer state, a cub checks to see if it is holding a deschedule for that viewer in that slot, and if so it discards the viewer state.

Deschedule requests are held for at least a few seconds after the slot they describe has passed the cub holding the request, in order to catch any late viewer state records. Cubs try to keep viewer states at least `minVStateLead` in front of the slot they describe, so trailing the slot is unusual. If a viewer state arrives so late that the cub would have already discarded any deschedules for that slot, the cub discards the viewer state. We have never detected this happening, but if it did, in the worst case it would could cause a viewer to be spontaneously descheduled because the viewer state is discarded without being forwarded. Because viewer states are discarded if they arrive later than the amount of time that deschedules are held, a viewer cannot be spontaneously rescheduled.

Cubs pass deschedule requests around in much the same way that they do viewer state records, each cub forwarding every deschedule request to its successor and second successor. The deschedules propagate until they're more than `maxVStateLead` in front of the slot being descheduled, at which time they're guaranteed to have caught all viewer state records for the viewer. Unlike viewer state records, cubs forward deschedule requests as soon as they receive them. In theory, all that is necessary is that the deschedules move around the ring faster than the viewer state records, but we saw little advantage in slowing them down.

The precise semantics of a deschedule request are "If this instance of viewer is in this schedule slot, remove the viewer." It is these semantics that make the operation so simple to implement. If the correct viewer (and correct instance, where instance corresponds to the particular start request being descheduled) is not in the slot corresponding to the deschedule request, the request does nothing. In order to carry out such a request, a cub receiving it does not need to know that its local view of the schedule is correct because applying the deschedule transformation will never reduce the correctness of its view. Having a deschedule request floating around after the slot has been reallocated will not cause incorrect results.

4.1.3 Adding New Viewers into the Schedule

When a viewer wishes to start receiving a file, the viewer sends a request to the controller. The controller forwards the request to the cub holding the first block that the viewer wishes to receive and to that cub's successor for redundancy. When a cub receives such a request, or when a cub is holding a redundant copy and the cub's predecessor has failed, the cub enters the request into a queue of viewers waiting for service. When the cub notices a free schedule slot, it enters the viewer from the head of the queue into the slot.

Unlike the deschedule operation, inserting a viewer into a schedule slot requires that the cub know that the slot is not occupied. Just because a cub's local view of the schedule shows a particular slot as being empty, it cannot conclude that the slot is in fact empty; the viewer state simply may not yet have arrived. Inserting a viewer into a slot that is already occupied would result in a loss of service for one of the viewers occupying the slot.

In order to avoid conflicts in a schedule slot, Tiger assigns ownership of each slot to at most one cub at a time. A cub may

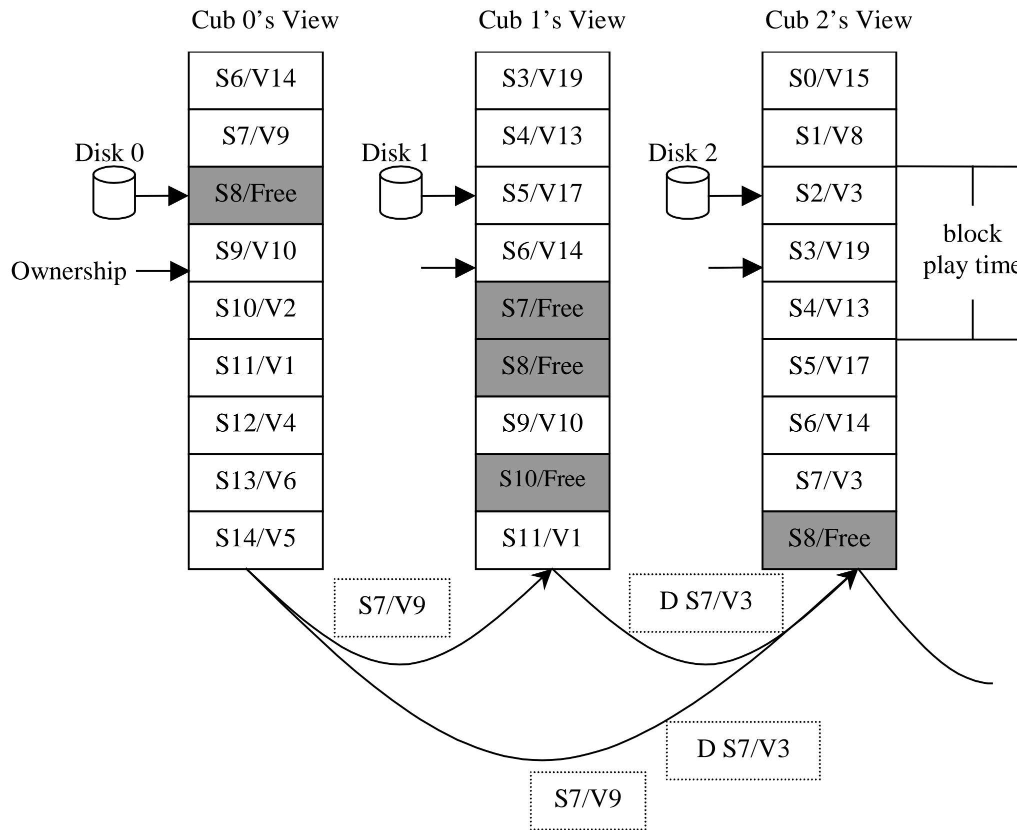

Figure 7: Example of Views of the Schedule

insert into a slot if and only if it owns that slot and the slot is empty. The time during which a cub owns a slot is small relative to the block play time, and hence to the distance between cubs. Consequently, there is a reasonable period of time for a cub that assigns a viewer to a slot to tell the next owner of the slot about the assignment. Figure 6 illustrates the concept of ownership of schedule slots. When a cub's pointer (shown on the top of the diagram) is in the region between the arrows labeled "ownership of slot *n*," the cub owns the slot and may schedule into it if it is empty. When no cub's pointer is in the ownership region for a particular slot, the slot is unowned and no cub may schedule into it. The ownership period begins some time before the beginning of the slot. This is to allow the cub that made the assignment to perform the disk read in order to get ready to send the first block to the net. As a result, the scheduling lead is always at least one block service time. Typically, it is somewhat longer to allow for variations in disk performance.

The `minVStateLead` parameter is always much larger than the scheduling lead. Thus, in normal situations the preceding cubs would have sent the viewer state for any viewer occupying the slot in question long before the scheduling cub gains ownership of the slot. When a viewer is first added to a slot, there is at least block play time minus ownership duration for the new viewer state to get to the next owner of the slot. In the single bitrate Tiger the block play time must be bigger than the largest expected inter-cub communication latency.

There is an interaction between removing and inserting viewers into the schedule. If the inserting cub believes that the slot is empty because it saw a deschedule request for the previous occupant, any cub seeing the newly inserted viewer must also have seen the deschedule, or never have seen the old occupant in the first place. Tiger uses TCP to control the communication links between cubs, so messages sent directly from one cub to another arrive in order. Therefore, any cub directly connected to the inserting cub sees the deschedule before the newly inserted viewer, since the inserting cub sent out the deschedule before doing the insertion. If any cub's predecessors either saw the deschedule before the insert, or never saw the removed viewer in the first place, they would forward the deschedule or never forward the viewer state for the descheduled viewer to the new cub, so by induction there is no conflict.

Figure 7 shows an example of views of the schedule for the first three cubs of a greater than three cub system. In this example, `minVStateLead` is artificially low so that the differences in views is more obvious. Unlike in Figure 3, each cub's pointer is at the same position within its view. This is because the region of the schedule spanned by a view is relative to the position in the schedule being processed by disk in question; cubs do not keep information about parts of the schedule that do not currently interest them. In the example, cub 0 has not yet gotten around to forwarding the viewer state for viewer 2 in slot 10. As a result, slot 10 in cub 1's view is shown as free. By the time that cub 1's ownership pointer gets to this slot, cub 0 will have forwarded the viewer state, so the slot will be filled. In practice, `minVStateLead` would be bigger, and the viewer state would have long ago arrived.

A more interesting case is slot 7. This slot was occupied by viewer 3, which was descheduled. When cub 0's ownership pointer got to slot 7, it saw an empty slot, inserted viewer 9 and forwarded the viewer state on to cubs 1 and 2. At the time shown

in the example, the initial deschedule for slot 7 is still in transit to cub 2, arriving on both of its incoming links. Because cub 2 has not yet seen the deschedule, it still shows slot 7 as holding viewer 3. Cub 1 has seen the deschedule (and is forwarding it on to cub 2), but has not yet seen the viewer state for the newly inserted viewer 9, and so it shows slot 7 as free. None of these inconsistencies causes a problem, because by the time a cub takes action based on the contents of a slot, the slot is up-to-date. In practice, there would be much more lead time between the insertion by cub 0 and cub 1's ownership pointer hitting slot 7, but the scale is shortened for illustrative purposes.

4.2 Distributing the Network Schedule

The single bitrate Tiger system has been complete and delivering data to customers in trial situations for about two years. Implementation of multiple bitrate Tiger systems is not yet completed, and in particular the disk schedule portion is not written. The network schedule is complete and working. We describe multiple bitrate Tigers only because they illustrate a more complex case in maintaining coherent hallucinations.

The subsections of 4.1 describe the implementation of various operations on the Tiger schedule. Steady state only requires making sure that scheduling information propagates by the time it's needed. Deleting a viewer from the schedule is similar in that no real coordination is required between cubs. Inserting a viewer into the schedule can still be accomplished locally by carefully limiting the circumstances under which a cub may make an insertion. In the multiple bitrate Tiger system schedule entries are a block play time wide. Inserting into the schedule requires knowing that the schedule capacity won't be exceeded at any point. By definition, cubs are separated from one another in the schedule by a block play time, so it is impossible to employ the technique of the single bitrate Tiger wherein the inserting cub has exclusive ownership of the necessary chunk of the schedule.

When a cub wants to make an insertion into the network schedule, it first checks its local copy of the schedule to see if it can rule out the insertion based solely on its view of the schedule. If it cannot, it tentatively makes the insertion, starts the disk operation to read the first block of the file, and sends out messages to the succeeding cub asking it if it's alright to make the insertion according to its view of the schedule. When a cub receives such a message, if its view of the schedule has sufficient room it makes an entry that reserves the necessary space and tells the originating cub. This entry will not result in any work being done or any schedule information moving to other cubs, only in a reservation of space. If the proposed entry would overflow the schedule, the succeeding cub tells the originating cub.

If the originating cub receives confirmations from the succeeding cub early enough to start sending the initial block of the play on time, it will commit the schedule insertion and generate a viewer state for the new viewer. When the succeeding cub that made the tentative schedule insertion receives the viewer state, it will replace the reservation with a real schedule entry. Because the originating cub overlaps the disk I/O and communication between cubs, there will almost always be time for the communication with the succeeding cub without having to increase the scheduling lead value.

If a cub receives a negative confirmation of a tentative insertion, or doesn't receive a response from the succeeding cub in time, it will abort the tentative schedule insertion and stop the disk I/O (if it's not already complete). The originating cub replaces the start playing request at the head of the queue, and retries it when there is more available schedule space.

4.3 Lessons from Tiger's Distributed Scheduling

Because of Tiger's basic striping policy, the cubs all move through the global schedule as time passes. This is not a necessary property of coherent hallucinations. It is easy to imagine other systems in which participants' views are divided statically, or in which they move throughout the hallucination in some less well structured way. The cubs' lock step movement through the schedule is a property of the problem that Tiger solves rather than of coherent hallucinations in general.

We found a number of techniques to be helpful in implementing Tiger's coherent hallucination. Idempotence of messages aids in fault tolerance by allowing routine double sending, so information is not lost even during the period between a failure and its detection. Relying on bounded communication latency in the ownership protocol for start playing requests in the single bitrate system removes the necessity of two way communications to do schedule insertions. Insertion in the multiple bitrate system shows how communications latency can be hidden by overlapping it with speculative action (the disk read). In both the single and multiple bitrate systems, schedule insertions are committed (become a part of the coherent hallucination) when a message to that effect makes it to at least one other machine; in general, n-way fault tolerance requires that a decision be known by at least n+1 machines. Tiger is able to overcome nearly all of the short-term performance variations that happen in the real world by doing work (disk reads) relatively far ahead of schedule when possible.

5. Performance Measurements

Unlike traditional systems where speed is the primary measure of success, a video server succeeds by consistently meeting its deadlines, by scaling well, and by dealing appropriately with component failures. The amount of work done to implement the Tiger schedule is small relative to the work needed to move megabytes of data per second from the disk to the network. Furthermore, the schedule protocols need to be latency tolerant to handle network delays. As a result, the speed of the schedule management operations is of little consequence.

This section describes an ATM Tiger configuration set up for 2 Mbit/s video streams. It uses fourteen cubs, each of which is a Pentium 133 MHz personal computer with 64 Mbytes of RAM, a PCI bus, four IBM Ultrastar 2.25 or 4.5 Gbyte drives and a single FORE Systems PCA 200E OC-3 ATM adapter. Most of the disks were of the 2.5 Gbyte variety, but a few of the older drives failed and had to be replaced with larger drives because the 2.5 Gbyte drives were no longer available. The 4.5 Gbyte drive is identical except that it has twice as many platters. As a result, performance is similar, but because we use only 2.5Gbytes of these disks all of the accesses are concentrated in the outer (faster) half of the disk. We arranged in our failed-mode test to have all of the mirroring disks be of the smaller variety to avoid skewing the disk performance numbers in that test. The cubs each have one Adaptec 3940UW dual channel SCSI controller with two of the disks connected to each of the SCSI channels. The Tiger controller is a Gateway 2000 133 MHz Pentium. It is on the ATM network and communicates with the cubs over it. Similarly, the cubs communicate with one another over the ATM. A variety of machines attached to the ATM network serve as clients: 22

200MHz Pentium Pro and 9 90MHz Pentium machines, with memory varying from 64 to 128 Mbytes. Each of these machines is capable of receiving between 15 and 25 simultaneous 2 Mbit/s streams depending on the processor type and memory size. For the purpose of data collection, we ran a special client application that does not render any video, but rather simply makes sure that the expected data arrives on time. This client application allows more than one stream to be received by a single client computer.

This 56 disk Tiger system is capable of storing slightly more than 64 hours of content at 2 Mbit/s. It is configured for 0.25 Mbyte blocks (hence a block play time of 1s) and a decluster factor of 4. According to our measurements, in the worst case each of the disks is capable of delivering about 10.75 primary streams while doing its part in covering for a failed peer. Thus, the 56 disks in the system can deliver at most 602 streams. The FORE ATM network cards and system PCI busses are sufficiently capable that the disks are the limiting factor in this configuration.

We ran two experiments: unfailed and failed. The first experiment consisted of loading up the system with none of the components failed. The second experiment had one of the cubs (and consequently all of its disks) failed for the entire duration of the run. In each of the experiments, we ramped the system up to its full capacity of 602 streams.

In both experiments we increased the load on the server by adding 30 streams at a time (except that we added 2 during the final step from 600 to 602 streams), waiting for at least 50s and then recording various system load factors. The clients generated reports if they did not see all the data that they expected, and we kept track of the reports of lost data.

We loaded the system with 64 different files, each 1 hour in length. These files were filled with a test pattern rather than actual video data; unlike real video which varies somewhat in bitrate from second to second, the test files completely filled the available 2 Mbit/s bandwidth. The clients randomly selected a file, played it from beginning to end and repeated. Because the clients' starts were staggered and the cubs' buffer caches were relatively small (20 Mbytes/cub), there was a low probability of a buffer cache hit. We measured the overall cache hit rate at less than 0.05% over the entire run for each of the experiments. The disks were almost entirely full, so reads were distributed across the entire disks and were not concentrated in any particular portion.

The most important measurement of Tiger system performance is its success in reliably delivering data on time. We measured this in two ways in our experiments. When the server fails to place a block on the network for whatever reason it reports that fact. When a client fails to receive an expected block, it also reports it. In the non-failed experiment, the server failed to place 15 blocks on the network, each because the disk read hadn't completed in time. These missed disk completions were spread over the entire test, rather than being clustered at the highest load. Thus, we believe that these lost blocks are indicative of occasional blips in disk performance rather than of overloads. In addition to the 15 blocks that were not available from the disk on time, the clients reported 8 blocks were undelivered. Unlike the disk-related missed blocks, the client-reported ones happened at the highest system load. Most likely, they were due to overloads at the clients rather than at the server because they happened on the more heavily loaded clients. The non-failed test sent more than 4.1 million blocks and over a Tbyte of data, for an overall loss rate of about 1 block in 180,000 (1 in 275,000 if you discount the blocks that may have been lost by the clients rather than by the server).

After the ramp-up in the failed mode test, we allowed the system to run at 100% schedule load (602 streams) for about an hour. During the ramp up phase, the server had failed to place 46 blocks on the network, in each case because the disk had not completed the read. The clients were more evenly loaded in this test than in the non-failed test, and as a result reported receiving every block that the server claimed to have sent. The ramp up phase attempted to send about 3.6 million blocks, for an overall loss rate of about 1 in 78,000. During the hour long run at full load, the disks failed to complete an additional 54 blocks among over 2.1 million total blocks scheduled, for a loss rate of just over 1 in 40,000. We believe that these end-to-end loss rates are well within acceptable limits for most applications.

In addition to measuring undelivered blocks, we also measured the load on various system components. In particular, we measured the CPU load on the controller machine and cubs, and the disk loading. The cub CPU number reported in our graphs is the mean of the load average of each cub measured over a 50 second period. The cubs typically had loads very close to one another, so the mean is representative of the load on each cub. Disk load is the percentage of time during which the disk was waiting for an I/O completion (i.e., the time between when the cub asked Windows NT to read from the disk and when NT reported that the read had completed). Again it is the mean over all disks, but all disks had similar loads, so it is representative. In the failed mode test, the disk load reported is for the disks of one of the cubs that was mirroring for the failed cub, rather than for the disks of all of the cubs in the system.

We measured the control traffic between the cubs as a function of the system load. For the most part, this traffic consists of viewer state messages. Our graphs show the control traffic in bytes per second from one particular cub to all other cubs. In the failed mode test, we measured the control traffic from a cub that was mirroring for the failed cub. As you might expect, the control traffic in failed mode is roughly double that in non-failed mode, because for each primary viewer state forwarded, the mirroring cub must also forward a mirror viewer state. In any case, the highest control traffic that we saw was under 21 Kbytes/s.

Figures 8 and 9 (on the last page) show the measured numbers for the normal operation and one cub failed tests, respectively. The mean load measurements should be read against the left hand *y*-axis scale, while the control traffic curve uses the right hand scale.

Observe that the machine's loads increase as you would expect: The cubs' load increases linearly in the number of streams, while the controller's does not depend on system load.

Even with one cub failed and the system at its rated maximum load, the cubs didn't exceed 85% mean CPU usage. We believe that most of the CPU time was spent packetizing the video data to be sent to the clients. In the failed mode test at full schedule load, Tiger ran the disks on the mirroring cubs at over 95% duty cycle while still delivering all streams in a timely and reliable fashion. Each disk delivered 3.36 Mbytes/s when running at load (10.75 0.25 Mbyte/s streams/disk, plus 25% for mirroring).

At the highest load, the mirroring cubs were delivering 43 streams (plus 10.75 streams for the failed cub) at 2 Mbits/s, and so were sustaining a send rate of over 13.4 Mbytes/s, not including overhead or control traffic.

Figure 10 shows the distribution of stream start times versus the schedule load. This graph combines the stream starts from both the failed and non-failed tests, for a total of 4050 starts. Each start is represented by a gray dot on the graph at the appropriate schedule load and delay coordinate. The heavy black line represents the mean of the starts at that particular schedule

load. It looks lower than you would expect because most dots are clustered at the lower loads and overwrite one another on the graph. We did not show startup times for schedule loads lower than 50%, but they were all clustered around 1.8 seconds, the minimum startup time. 1 second of this time is due to the time to transmit a 1 second Tiger block. The test client records the receive time of a block to be when the last byte of the block arrives rather than when the first byte arrives. Video rendering (non-test) clients are free to begin rendering before the entire block arrives, however, so they may mask some of this second. The remaining 800ms is a combination of network latency and scheduling lead (which includes time for the first block disk read).

Even at schedule loads of 95%, the mean time to start a viewer is less than 5 seconds. However, there are a reasonable number of outliers that took over 20 seconds. For that reason, we do not recommend running Tiger systems at greater than 90% load, and suggest limiting them to even lower loads. Tiger contains code to prevent schedule insertions beyond a certain level, which we disabled for this test. At very high schedule loads, some insertions took about as long as the entire 56s schedule to complete, and in larger systems would take longer.

A final measurement was the time for the system to reconfigure from a cub failure. We loaded the system to 50% of capacity and cut the power to a cub. We inspected the clients' logs and found about 8 seconds between the earliest and latest lost block.

6. Related Work

Tiger systems are typically built entirely of commodity hardware components, allowing them to take advantage of commodity hardware price curves. By contrast, other commercial video servers, such as those produced by Silicon Graphics[Nelson95] and Oracle[Laursen94], tend to rely on super-computer backplanes or massively parallel memory and I/O systems in order to provide the needed bandwidth. These servers also tend to allocate entire copies of movies at single servers, requiring that content be replicated across a number of servers proportional to the expected demand for the content. Tiger, by contrast, stripes all content, eliminating the need for additional replicas to satisfy changing load requirements. [Berson94] proposes an independently developed single-machine disk striping algorithm with some similarities to that used by Tiger. SPIFFI [Freedman96] is a parallel file system implemented on an Intel Paragon system that can stripe data across large numbers of disks and can be used for multimedia files (as well as for more traditional parallel filesystem tasks).

There is a certain similarity between the coherent hallucination model and distributed [Li88; Nitzberg91] or tightly coupled [Kuskin94; LaRowe91] shared memory multiprocessing. Both types of systems have a notion of a global abstraction upon which multiple participants act. Both require some attention by the programmer to keep coherence between the participants. The primary difference lies in that shared memory systems do not have a hallucination, but rather directly implement the global abstraction. They are usually more tightly coupled, and often lack fault tolerance. In these systems, a view corresponds to the portion of the shared data structure that is used by any particular participant. Because the view is not explicit to the programmer, it is often harder to judge the scalability and access patterns.

The implementations of some existing wide scale distributed systems can be viewed as coherent hallucinations. For example, the Domain Name System [Mockapetris88] can be viewed as a simple form of coherent hallucination. A directory of the global namespace is the hallucination, while each DNS server's authoritative knowledge and cached information make up the views. Other examples include protocols such as RIP [Malkin94], OSPF [Moy94] and BGP [Rekhter95] for IP routing. In these protocols, the existence, up/down state and speed/load of all of the routers and links in the network take the place of the hallucination, and the current set of beliefs about them correspond to views. These protocols differ from Tiger's coherent hallucination in that the views describe the entire system rather than just a subset, but like a view in a coherent hallucination they are allowed to be out of date. A further example is the portion of the Network Time Protocol [Mills91] dealing with cascaded synchronization. The synchronization tree is a hallucination; it describes the dynamically varying hierarchy of timing propagation through a synchronization subnet, yet it is not fully represented at any node in the system. Each node's view is the peer selection process performed with respect to the node's immediate neighbors.

7. Summary and Conclusions

Tiger is a video server that is designed to scale to tens of thousands of simultaneous streams of digital video. It stripes its content files across a collection of personal computers and high speed disks, and combines the file blocks into a stream through an ATM switch. It uses a schedule to prevent resource conflicts among viewers. In the abstract, the schedule is a data structure whose size is proportional to that of the Tiger system. In practice the machines comprising the Tiger system see only part of the global schedule, and have only non-authoritative knowledge about most of what they know, a technique we name "coherent hallucination."

We found that:

- Tiger maintains its schedule in a manner that is fault tolerant, robust and scalable.
- Tiger is able to provide a number of streams of video data that is not limited by its schedule management algorithms, but rather by its hardware's bandwidth.

Acknowledgements

We would like to thank Troy Batterberry, Akhlaq Khatri, Erik Hedberg, and Steven Levi for the use of their equipment and talents in collecting the data for the performance measurements. We would also like to thank Bill Schiefelbein, Chih-Kan Wang, Aamer Hydrie and the rest of the Netshow™ Pro Video Server team for their help with the software and ideas we describe. We owe a debt to the SOSP program committee and outside reviewers for their suggestions on the organization and presentation of the paper. We would like to thank Garth Gibson, Rick Rashid and Nathan Myhrvold for their architectural suggestions during the early phase of the Tiger project, and Fyeb for dex.

Bibliography

[Berson94] S. Berson, S. Ghandeharizadeh, R. Muntz, X. Ju. Staggered Striping in Multimedia Information Systems. In *ACM SIGMOD '94*, pages 79-90.

[Bolosky96] W. Bolosky, J. Barrera III, R. Draves, R. Fitzgerald, G. Gibson, M. Jones, S. Levi, N. Myhrvold, R. Rashid. The Tiger Video Fileserver. In *Proceedings of the Sixth International Workshop on Network and Operating*

System Support for Digital Audio and Video. IEEE Computer Society, Zushi, Japan, April 1996. Also available from www.research.microsoft.com in the operating systems area.

[Freedman96] C. S. Freedman, J. Burger and D. J. DeWitt. SPIFFI – A Scalable Parallel File System for the Intel Paragon. In *IEEE Trans. on Parallel and Distributed Systems*, 7(11), pages 1185-1200, November 1996.

[Kuskin94] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. Hennessy. The Stanford FLASH Multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302-313, April, 1994.

[LaRowe91] R. LaRowe, C Ellis and L. Kaplan. The Robustness of NUMA Memory Management. In *SOSP 13*, pages 137-151, 1991.

[Laursen94] A. Laursen, J. Olkin, and M. Porter. Oracle Media Server: Providing Consumer Based Interactive Access to Multimedia Data. In *ACM SIGMOD '94*, pages 470-477.

[Li88] K. Li. IVY: A Shared Memory Virtual Memory System for Parallel Computing. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages II-94 – II-101, 1988.

[Malkin94] G. Malkin. RIP Version 2 Protocol Analysis. RFC 1721. November, 1994

[Mills91] D. L. Mills. Internet Time Synchronization: The Network Time Protocol. In *IEEE Transactions on Communications*, pages 1482-1493, Vol. 39, No. 10, October, 1991.

[Moy94] J. Moy. OSPF Version 2. RFC 1583. March, 1994.

[Nelson95] M. Nelson, M. Linton, and S. Owicki. A Highly Available, Scalable ITV System. In *SOSP 15*, pages 54-67. December, 1995.

[Mockapteris88] P. Mockapetris and K. Dunlap. Development of the Domain Name System. In *Proceedings of SIGCOMM '88*, pages 123-133, April 1988.

[Nitzberg91] B. Nitzberg and V. Lo. Distributed Shared Memory: A Survey of Issues and Algorithms. *Computer*, 24(8):52-60, August, 1991.

[Patterson88] D. Patterson, G. Gibson, R. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *ACM SIGMOD '88*, pages 109-116.

[Rekhter95] Y. Rekhter, T. Li. A Border Gateway Protocol 4 (BGP-4). RFC 1771. March, 1995.

[Ruemmler94] C. Ruemmler and J. Wilkes. An Introduction to Disk Drive Modeling. *Computer*, 27(2):17-28, March, 1994.

[Van Meter97] R. Van Meter. Observing the Effects of Multi-Zone Disks. *In Proceedings of the USENIX 1997 Annual Technical Conference*, page 19-30, January, 1997.

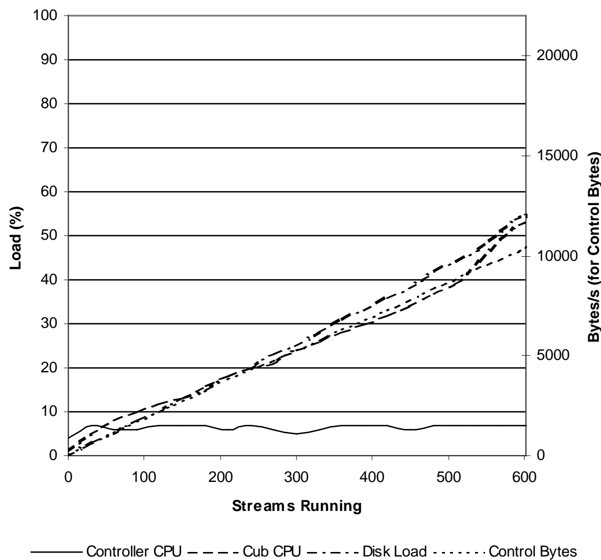

Figure 8: Tiger Loads, No Cubs Failed

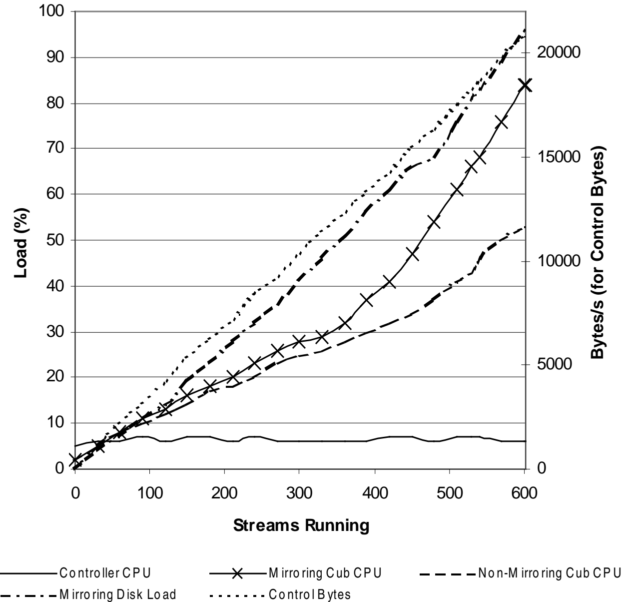

Figure 9: Tiger Loads, One Cub Failed

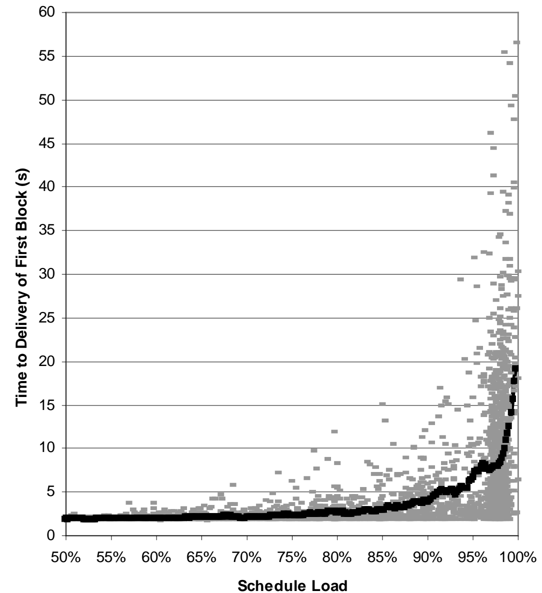

Figure 10: Stream Startup Latency