Improving responsiveness of a stripe-scheduled media server

John R. Douceur and William J. Bolosky

Microsoft Research, Redmond, WA 98052-6399

ABSTRACT

Thrifty scheduling is an algorithm that improves the responsiveness of a stripe-scheduled multimedia server. It increases the determinism of the data-distribution service, reduces the likelihood of high startup delays, and enables an increase in the rated load of the system. A stripe-scheduled media server is a distributed video-on-demand system that load-balances by striping video data across multiple computer nodes and cyclically scheduling the distribution of the data. The server displays highly variable startup delays in response to requests for data streams. These delays are due to clusters of allocated slots in the distribution schedule, which form naturally as the system load increases. Thrifty scheduling is a scalable algorithm that improves responsiveness by allocating streams to schedule slots in a way that reduces the clustering in the schedule. This algorithm has been incorporated into the Tiger video fileserver.

Keywords: Video server, video-on-demand, latency reduction, distribution, responsiveness, deterministic service, cyclical scheduling, thrifty, greedy, startup delay

1. INTRODUCTION

We developed an algorithm, called thrifty scheduling, that improves the responsiveness of a stripe-scheduled media server. It decreases the variability of video stream startup delay, reduces the likelihood of excessively high startup delay, and allows the rated load of the system to be increased without increasing the mean startup delay. Stripe-scheduled media servers display highly variable startup delays in response to requests for data streams, and the mean startup delay increases much faster than linearly with increasing schedule load. Thrifty scheduling shifts startup delay from relatively high-latency starts to relatively low-latency starts in order to reduce the variability of the startup delay.

A stripe-scheduled media server is a distributed system that balances the load of video distribution by striping the data in each file across all of its disks. Data is sourced according to a cyclical schedule that ensures the availability of system resources for the video streams. The disks logically move along the schedule in real time, sourcing data for the viewers in the schedule. To improve scalability and to remove a single point of failure, the schedule is implemented in a distributed fashion among the computers that comprise the server.

Startup delay results from waiting for the disk that holds the requested content to rotate to a vacant slot in the schedule. Startup delay is increased by clusters of allocated slots in the distribution schedule, which form naturally as the system load increases. Thrifty scheduling decreases delay variability and reduces the likelihood of high delays by allocating schedule slots in a way that reduces clustering in the schedule. Large values and high variability of startup delay are unpleasant for viewers and degrade the usability of the system. Although delays in the range of several seconds may not seem significant for the initial start of a video stream, such delays and delay variances will noticeably interfere with frequent pauses, resumes, and sequencing events that imitate VCR functionality.

The remainder of this paper is organized as follows: Section 2 describes the media-server schedule, necessary background to understanding the problem of startup delay, which is presented in Section 3. Section 4 introduces the concept of thrifty scheduling to address the startup delay problem. Section 5 introduces the complication of distributing the schedule for scalability and fault-tolerance, and Section 6 describes the distributed thrifty scheduling algorithm necessitated by the distributed schedule. Section 7 presents performance results showing the effect of thrifty scheduling in a modestly sized media server. Sections 8 and 9 wrap up with related work and conclusions.

2. THE MEDIA-SERVER SCHEDULE

A prototypical example of a stripe-scheduled media server is the Tiger distributed video fileserver. Tiger balances the load of video distribution by striping the data in each content file across all of the disks in the system. Data for each active stream is sourced according to a cyclical schedule that ensures the availability of system resources for the video streams. The disks logically move along the schedule in real time, performing disk reads and network transmissions for the viewers in the schedule. The schedule assumes that all streams have comparable bandwidth.

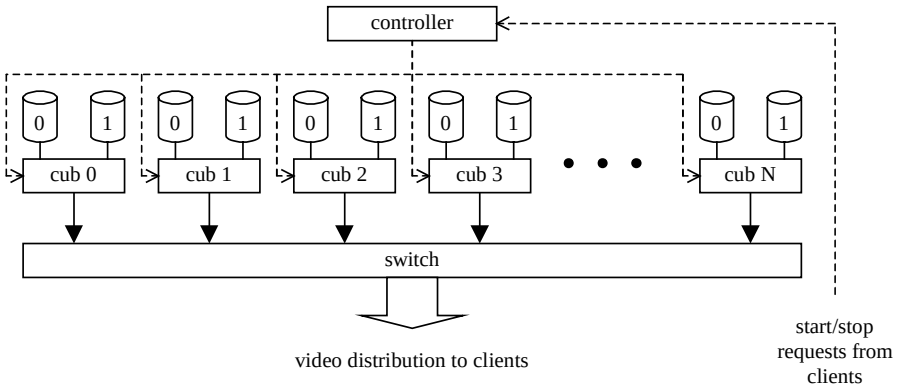

Figure 1: Typical Tiger system hardware organization

Tiger[3,4], the technology underlying Microsoft® NetShow™ Theater Server, is a video fileserver intended to supply digital video data to up to tens of thousands of users simultaneously. It is a distributed system organized as a collection of personal computers interconnected by a switched network. A typical system comprises 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. Each of the cubs hosts a number of disks, which are used to store the multimedia content. The cubs are connected to the switched network and to the controller. Figure 1 illustrates the hardware organization of a typical Tiger system.

To handle imbalances in demand for particular files, every file is striped across every disk and every cub in a Tiger system. Because each file is distributed over every disk and every cub, the load is distributed among all of the system components. Files are broken up into blocks, which are pieces of equal play 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 which point the next block is placed on the lowest numbered disk, and the process continues. 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 – 10 Mbit/s) files. The block play time is the same for every file in a particular Tiger system.

Tiger uses a schedule to ensure the availability of system resources for the video streams, as shown in Figure 2. The disks logically rotate through the schedule at a fixed rate so as to deliver successive segments of a data stream from successive disks. The schedule is an array of slots, with one slot for every stream of system capacity. The time that it takes to process one block of file data is called the "block service time" and is determined by either the speed of the disks or the capacity of the network interface, whichever is the bottleneck. The schedule is an integral multiple of both the block play and block service times.

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. The pointer for each disk is one block play time behind the pointer for its predecessor.

The Tiger schedule assumes that all video streams are of comparable bandwidth. Streams of lower bandwidth can be supported, but the low-bandwidth streams will consume the same amount of system resources as normal-bandwidth streams, leading to inefficiencies if large numbers of low-bandwidth streams are requested. There is current research into generalizing the Tiger architecture to efficiently support streams of multiple bandwidths, and in the conclusion of this paper we discuss adapting thrifty scheduling to such a system.

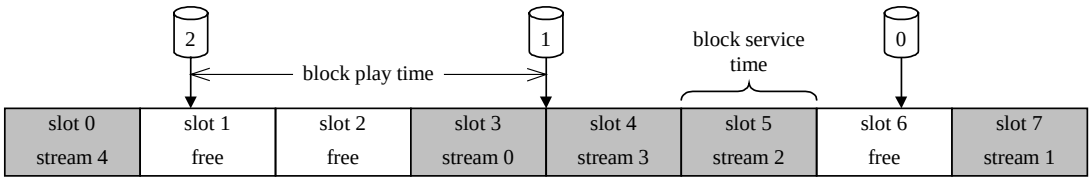

Figure 2: Example Tiger schedule

3. STREAM INSERTION AND STARTUP DELAY

Waiting for a vacant schedule slot causes a delay in starting a video stream. This startup delay is minimized by allocating schedule slots in a greedy fashion. Straightforward analytical approximation shows the startup delay to be highly variable, its distribution to be heavily tailed, and its mean value to increase much faster than linearly with schedule load. Startup delay is increased by clusters of allocated slots in the distribution schedule, which form naturally as the system load increases. Large values of startup delay are unpleasant for viewers, and high variability in startup delay degrades the usability of the system by interfering with virtual-VCR functionality.

Startup delay results from waiting for a disk to rotate to a vacant schedule slot. In the system illustrated in Figure 2, suppose that at the instant shown a new viewer requests a file whose first block is stored on disk 2. Since it takes a block service time to read a block of data from the disk, if the disk were able to begin the first read for the new stream immediately, then the stream would be ready to begin its transmission in the middle of slot 2; however, since transmission must start at the beginning of a slot, the transmission could not actually begin until slot 3. However, in this example, slot 3 is occupied by stream 0, so the start of the new stream must be delayed until the disk rotates to an open schedule slot. An examination of the schedule reveals that the next free slot is number 6, so the insertion has to slip by three more slots. If the block service time is 100 ms, then this schedule slip induces a startup delay of 300 ms.

The insertion procedure as just described employs a greedy strategy: For each new stream, the selected schedule slot is the one that minimizes the startup delay experienced by the requesting viewer. The scheduler did not need to select slot 6 for the new stream; it could have slipped three further slots to slot 1, for a startup delay of 600 ms. However, the greedy algorithm has the desirable property of minimizing the mean startup delay over all stream insertions and all schedule loads.

Approximating scheduling delay is straightforward. The procedure for slot selection is isomorphic to insertion into a hash table with linear probing, so the number of slots that will be slipped by an insertion into a schedule corresponds to the number of probes required for an insertion into a hash table. Thus[11], in a schedule of M slots of which N are occupied, the mean number of slots slipped is given by:

$$\mu = \tfrac{1}{2}\sum_{k=0}^{N}\frac{(k+1)N!}{(N-k)!M^{k}} - \tfrac{1}{2} \qquad (1)$$

This formula holds only if stream requests are random and independent, but its general shape is visible in all Tiger load-up curves: The mean delay begins at zero and increases faster than linearly as the load increases. Because the startup delay increases rapidly at high schedule loads, we suggested[4] that the schedule load be limited to 90% or lower.

The distribution of startup delays produced by greedy scheduling has a very long tail, implying that there are significant probabilities associated with high delays. The formula for the probability distribution is complex and not amenable to approximation, but a concrete example is illustrative: At a load of 80% in a schedule of 100 slots with a block service time of 100 ms, the mean startup delay for the next play request is only 0.85 seconds; however, there is a 1% probability that the startup delay will exceed 4.2 seconds. Thus, in this example, one out of every hundred users of the loaded system will experience a startup delay that is over five times greater than the mean value of the startup delay at that load.

Large values of startup delay at high loads are caused by the presence in the schedule of large clusters of contiguously allocated slots. Some amount of schedule clustering is virtually unavoidable; however, the greedy algorithm has a strong tendency to grow clusters for two reasons. First, the likelihood of a schedule insertion in the slot immediately following a cluster is proportional to the length of that cluster, so long clusters tend to grow longer. Second, two clusters near each other will be joined into a single cluster when the intervening slots are filled. Because of this second phenomenon, startup delay grows much faster than linearly at high schedule loads.

Experiments have shown that extreme variability in response time may produce lower user satisfaction than response times that are large but more deterministic[13]. Although the greedy algorithm minimizes mean startup delay over all stream insertions, there are two reasons why mean delay is not an appropriate metric for evaluating user satisfaction: First, mean behavior measures the aggregate effect of many schedule insertions, whereas each viewer experiences startup delay from a single insertion; a user who experiences the annoyance of an extraordinarily long delay is unlikely to be appeased by the knowledge that a large number of other users were serviced in a far more timely fashion. Second, user satisfaction does not vary linearly with response time; for example, the benefit from reducing one viewer's startup delay from 10 seconds to 1 second exceeds the total benefit from reducing ten viewers' startup delays from 2 seconds to 1 second.

Every stream start encounters startup delay. In a video-on-demand system, stream starts occur not only in response to the initial play request for a title, but also in response to resumes following pauses, fast-forwards, rewinds, and any other effects that disrupt the regular progression of the stream. Although delays in the range of several seconds may not seem significant for the initial start of a video stream, such delays and delay variances will noticeably interfere with the virtual-VCR functionality that consumers expect from video-on-demand systems.

4. THRIFTY SCHEDULING

We can improve perceived system responsiveness by reducing startup delays that are relatively high, at the expense of increasing startup delays that are relatively low. Thrifty scheduling allows a low startup delay to increase if doing so improves the quality of the schedule for future insertions. Schedule quality is computed using a metric that reflects the amount of clustering in the schedule, and this metric can be scalably computed using only local information in the schedule.

We define a hard cutoff between startup delays that are acceptably low and those that are unacceptably high. The thrifty scheduling algorithm is locally greedy in reducing unacceptably high startup delay, but it sacrifices immediate latency within the acceptable range for the sake of reducing the delay of later schedule insertions. An appropriate cutoff value may be determined by noting the mean startup delay at the maximum rated load of the system or by consulting the results of empirical studies of response time versus user satisfaction.

When a new stream is requested, the algorithm chooses – from among all available slots within the acceptable range – the slot that minimizes the clustering in the schedule. In the event of a tie, or if no slots are available within the acceptable range, then the algorithm reverts to greedily selecting the slot that results in the lowest startup delay.

This technique requires a measure of the clustering in a schedule. The metric should penalize not only schedules with long clusters of allocated slots but also schedules in which clusters are close together. Although the size of the gaps between clusters has no effect on the expected startup delay of the next insertion, it does affect the likelihood that those gaps will become filled, thus leading to larger clusters in the future. By reducing the likelihood of small gaps, the thrifty scheduling algorithm reduces the likelihood of coalescing clusters, which in turn reduces the expected startup delay of future insertions.

We have experimented with several clustering metrics, one of the simplest of which is the sum of the squared lengths of runs of identical occupancy. For example, the schedule illustrated in Figure 2 has four runs, one of length one (slot 6), two of length two (slots 1 through 2 and slots 7 through 0), and one of length three (slots 3 through 5); the squared sum is 18 ($1^2 + 2^2 + 2^2 + 3^2$). For a comparative example, if stream 0 were moved from slot 3 to slot 2, the clustering metric value would drop to 12, indicating a less clustered schedule.

A convenient property of this metric is that the change in clustering that results from an insertion in a given slot can be calculated from four attributes of the slot: the count of preceding occupied slots (O_P), preceding vacant slots (V_P), succeeding occupied slots (O_S), and succeeding vacant slots (V_S). The formula is:

$$\Delta \text{clustering} = 2\left(O_P O_S + O_P + O_S - V_P V_S - V_P - V_S\right) \tag{2}$$

By applying this formula to the schedule illustrated in Figure 2, we can evaluate the insertion of a stream into slots 1, 2, and 6, yielding a change in clustering of 2, 4, and 22, respectively, indicating the relative degradation in schedule quality from an insertion at each location.

5. DISTRIBUTED SCHEDULE MANAGEMENT

For reasons of scalability and fault tolerance, the Tiger schedule is not maintained in a centralized location, such as the Tiger controller. Instead, the schedule is distributed among the cubs[4]. Each cub keeps track of only a fixed-length portion of the schedule near where its disks are processing, and the schedule information propagates around the ring of cubs at the same rate that the cubs move through the schedule. The central controller forwards requests for new streams to the appropriate cubs, and the cubs queue the requests until they can be satisfied. To manage schedule modifications in this distributed system, at most one cub is permitted to assign a stream to a given slot at any given time.

The Tiger schedule is implemented in a distributed fashion. Every cub keeps track of the schedule entries that its disks will encounter in the next few seconds, as well as a little while into the past. Fault tolerance requires that every part of the schedule be stored on more than one cub, so each cub's portion of the schedule overlaps with portions of its neighbors. From time to time, each cub forwards entries to the next cub in line. The amount of time between when a schedule entry arrives at a cub and when that cub's block for that stream is due at the network is called the "forwarding lead time" of the schedule entry, and is controlled by two global system parameters that bound this value from above and below.

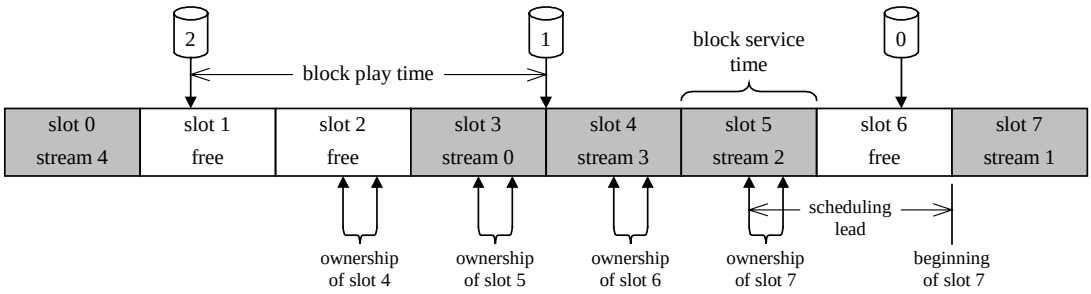

Figure 3: Ownership of schedule slots

When a viewer sends a stream request to the Tiger controller, the request is forwarded to the cub holding the first block that the viewer wishes to receive. When a cub receives such a request, the cub enters the request into a queue of viewer requests waiting for service. In accordance with greedy scheduling, whenever the cub notices a free schedule slot, it inserts the stream from the head of the queue into the slot.

In order to avoid conflicting assignments, Tiger assigns "ownership" of each slot to at most one cub at a time. A cub may 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 3 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 must begin before the beginning of the slot, in order to allow the cub that made the assignment to perform the disk read. As a result, the time between the beginning of slot ownership and the beginning of the slot, which is known as the "scheduling lead," is always at least one block service time. Typically, it is somewhat longer to allow for variations in disk performance.

As soon as a cub makes an assignment of a stream to a slot, it forwards the assignment to the next cub in line, which in turn forwards it to the next cub, and so on until the lower bound on the forwarding lead time is satisfied. Cubs do not notify their predecessors of a stream insertion, since the preceding cubs have no work to perform for the new stream.

6. DISTRIBUTED THRIFTY SCHEDULING

The distributed schedule complicates thrifty scheduling, since the cub sees only a small part of the schedule, it owns only one slot at a time, and it can queue multiple stream requests. The distributed thrifty scheduling algorithm makes conservative assumptions about slots not in its visible range. It evaluates the desirability of individual slots, rather than computing a metric of overall schedule quality. It determines whether to insert stream into its currently owned slot based on a comparison of the slot's desirability with that of other slots in the visible range. This comparison accounts for all streams in the request queue. The computational complexity of the algorithm is independent of system size, so it is scalable.

The distributed management of the Tiger schedule complicates thrifty scheduling in several ways. First, since only a portion of the schedule is visible to each cub at any time, the scheduling algorithm must make decisions based upon limited and incomplete knowledge of the schedule. Second, since a cub owns only one slot at a time, the algorithm cannot decide exactly where in the schedule to insert a new stream; it can decide only whether or not to insert the new stream into the currently owned slot. Furthermore, since a cub may not schedule a stream as soon as it receives the start play request, multiple requests can accumulate in its queue, and the scheduling algorithm will need to account for these queued stream requests in addition to the streams already in the schedule.

Since each cub sees only a portion of the schedule, it has to make assumptions about the parts of the schedule that it cannot see. A vacant slot whose ownership has been relinquished may be assigned a stream by one of the succeeding cubs, or the stream in an occupied slot may terminate or be canceled. Since the current cub does not know whether such events have occurred, it makes a best guess that the occupancy density of this no-longer-visible portion of the schedule is the same as that of the currently visible portion of the schedule. Since it has no real information about which slots are occupied, it uses a memoryless distribution as a best guess for the occupancy of the remaining slots.

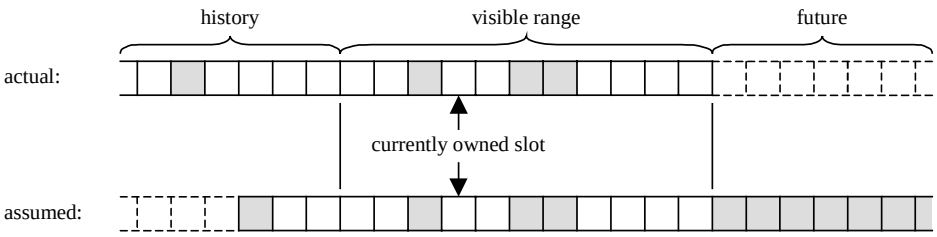

Figure 4: Assumptions about schedule occupancy

The cub could conceivably use this same technique as a best guess for the occupancy of the portion of the schedule that has not yet come into view. However, if the cub guesses that the future portion of the schedule is not heavily occupied, then it might choose to delay the assignment of a stream until this portion of the schedule comes under its ownership. Then, if it happens that the real schedule is more densely occupied than the cub's guess, the queued stream requests will have been delayed for no benefit. Thus, to avoid an overly optimistic expectation of future scheduling opportunities, the cub conservatively assumes that all not-yet-visible slots are occupied.

These assumptions are illustrated in Figure 4. The upper diagram shows the actual schedule, where the slots marked "visible range" are those for which the cub currently holds schedule entries. The cub has no knowledge of the slots whose entries have not yet arrived, so they are shown by dashed lines. Slots whose ownership has passed to the next cub can be assigned to streams without the current cub's awareness, so the cub has at best uncertain knowledge of slots in its history.

The lower diagram in Figure 4 shows the assumed schedule. All future slots are assumed to be occupied. The significant aspect of slot history is how many contiguously vacant slots follow the visible range. The cub calculates the occupancy density of the visible range – plus the first slot forward of the visible range, which is assumed to be occupied – as (3 + 1) / (11 + 1) = 1/3. If the historical region followed a memoryless distribution with this occupancy density, the expected position of the first occupied slot is after two vacant slots, as illustrated.

The scheduler determines whether to insert a stream into the currently owned slot via the following general algorithm: First, it evaluates the desirability of an insertion into the current slot. Then, it examines the visible portion of the schedule to see whether all streams in its request queue could be inserted in locations that are more desirable than the current location. If there are more desirable locations for all queued stream requests, then the current slot is left vacant. Otherwise, the stream from the head of the request queue is inserted into the current slot.

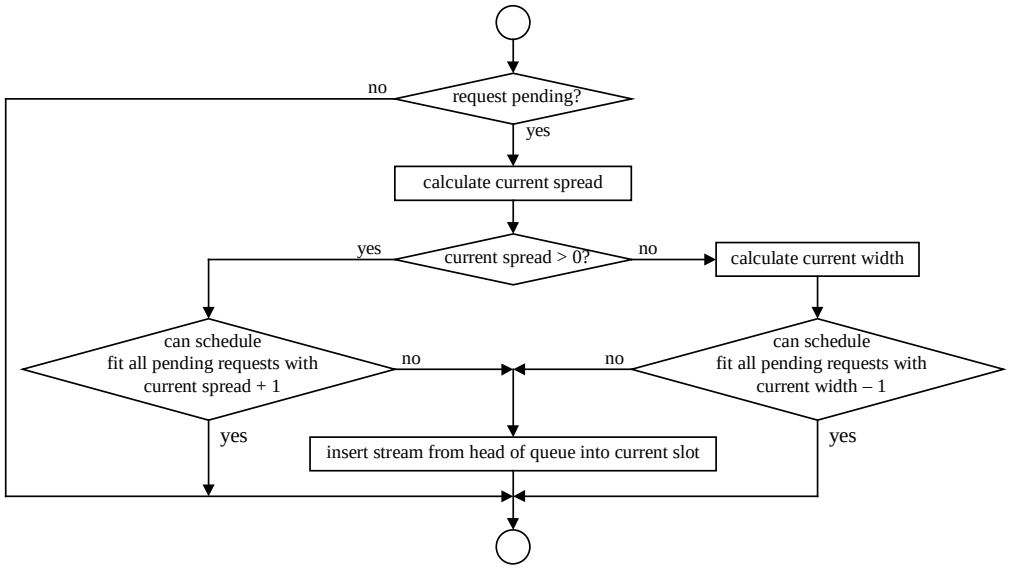

Figure 5: Determining whether to insert a stream into current slot

The distributed thrifty algorithm does not employ a measure of the clustering in the schedule. Instead, it defines two measurable values of an insertion at a target schedule location, the insertion spread and the insertion width. The insertion spread is the number of vacant slots between the target slot and the nearest occupied slot. The insertion width is the size of the cluster of occupied slots that would be created by an insertion into the target slot. (Note that width is greater than one if and only if spread equals zero.) The thrifty goals are to maximize insertion spread and to minimize insertion width.

The specific algorithm is illustrated in Figure 5. When the ownership period for a vacant slot begins, if a stream request is pending, then the insertion spread of the current slot is calculated. If the spread is greater than zero, then the algorithm attempts to find locations in the schedule with a spread of one more than the current spread for all streams in the request queue, without exceeding the acceptable startup delay for any stream, as illustrated below in Figure 6. Alternatively, if the spread equals zero, then the insertion width of the currently owned slot is calculated, and the algorithm attempts to find locations with a width of one less than the current width for all streams, without exceeding any stream's acceptable delay, as illustrated below in Figure 7. If for all queued requests, locations can be found with a larger insertion spread or a smaller insertion width, then the current slot is left vacant; otherwise, the stream from the head of the queue is inserted.

Figure 6 illustrates how to determine locations in the schedule with a target insertion spread. The algorithm walks backward from the furthest visible slot towards the currently owned slot, maintaining a count of the contiguously vacant slots beyond the slot under examination. Whenever the count exceeds twice the target spread, a stream can be inserted into the middle of the series of vacant slots with the target insertion spread. If a stream request in the queue is permitted to slip to that slot without exceeding its acceptable delay, then the request is marked as satisfiable, and the count is adjusted to account for the hypothetical insertion of a stream into the indicated slot. This process continues until either all requests in the queue are marked as satisfiable, in which case the target insertion spread is achievable, or all slots in the visible range have been examined, in which case the target insertion spread is not achievable.

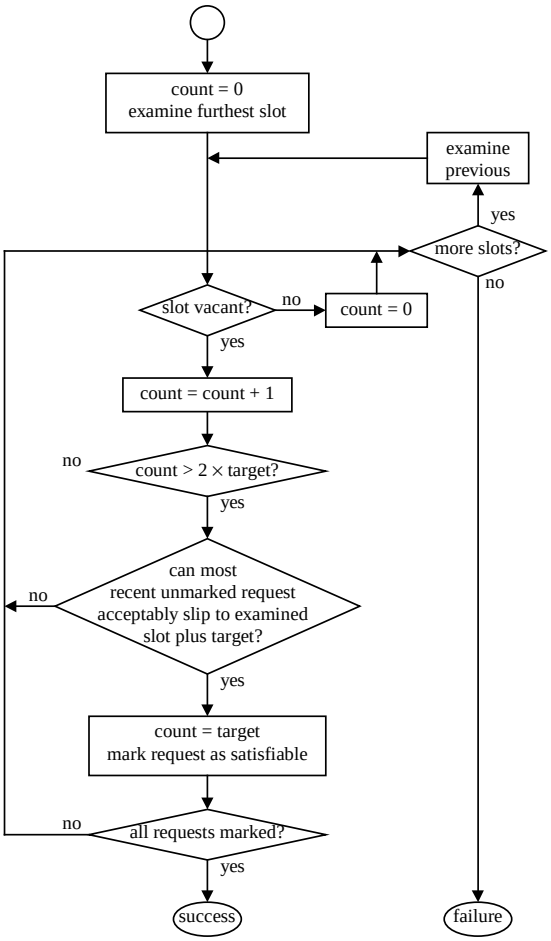

Figure 6: Can schedule accommodate target spread?

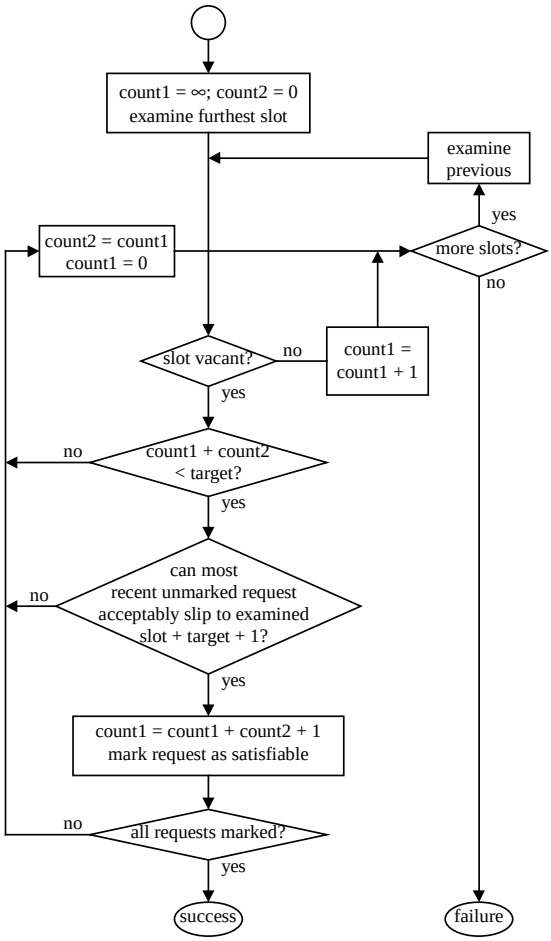

Figure 7: Can schedule accommodate target width?

Figure 7 illustrates how to determine locations with a target insertion width. It maintains two counts: one of contiguously occupied slots beyond the slot under examination, and one of contiguously occupied slots separated from the first group by exactly one vacant slot. The second count is zero if more than one contiguously vacant slot follows the first group. The sum of these two counts plus one indicates the width of an insertion into the vacant slot between the two groups. Whenever the algorithm encounters a vacant slot, the two counts of vacant slots are valid, so the algorithm calculates the width of an insertion into the slot immediately beyond the first group. If the width is no greater than the target width, and if a stream request in the queue is permitted to slip to that slot without exceeding its acceptable delay, then the request is marked as satisfiable, and the counts are adjusted to account for the hypothetical insertion of a stream into the indicated slot. This process continues until either all requests in the queue are marked as satisfiable, in which case the target insertion width is achievable, or all slots in the visible range have been examined, in which case the target insertion width is not achievable.

These algorithms are both scalable, since their execution times are independent of the system size. Their computational complexity is proportional to the lesser of the request queue size and the number of slots in the visible range. Although the request queue size can grow without bound, the number of slots in the visible range equals the ratio of forwarding lead time to block service time, the latter of which is proportional to the greater of disk read time and network transmission time. Thus, the number of execution steps is proportional to the disk or network speed. Since increases in peripheral speed are generally accompanied by increases in processor speed, the execution time should remain constant with advances in technology.

7. PERFORMANCE MEASUREMENTS

7.1. Simulation results

We extensively simulated the thrifty scheduling algorithm with various values for acceptable startup delay, comparing its responsiveness to that of the greedy scheduling algorithm. We found that thrifty scheduling reduces the probability of excess startup delay by 50% or more, depending on the value of acceptable delay. Thrifty scheduling also allows an increase in the rated load of the system by several percent without increasing the mean startup delay. The algorithm's ability to eliminate excessive delay is strongest at low schedule loads; however, it decreases mean delay at high schedule loads, where the delays are greatest. We validated the simulation results by statistical comparison with experimental results from a real Tiger system.

Our results were obtained primarily through simulation. We have chosen this approach for two reasons: First, it provides a relatively easy way to generate large numbers of data points and thus accurately quantify the responsiveness of the algorithm. Second, simulation allows us to isolate the responsiveness of the algorithm from the influence of other factors in the system.

We configured the simulator according to a working Tiger system in our lab, since we intended from the beginning to verify our simulation results against real experimental results. This 36-disk Tiger system is configured for a block play time of 1 second and a block service time of 138 msec, so it can deliver at most 261 streams. The lower and upper bounds on the forwarding lead time are 4 and 5 seconds, respectively, and the scheduling lead is 0.9 seconds.

We compared the responsiveness of the thrifty scheduling algorithm to that of the greedy scheduling algorithm. We ramped the system load from zero to full capacity by a series of stream requests. The acceptable delay ranged from 1 to 30 slots. The time between requests was governed by an exponential distribution with a mean value of 1 second. The probability that the first block of a request resided on each disk was governed by a uniform distribution (although we also ran the experiment with a Zipfian distribution but found that this made no appreciable difference in the result). The load-up series was repeated 600,000 times for the greedy algorithm and 35,000 times for each of 30 acceptable delay values for the thrifty algorithm.

As shown by the solid line in Figure 8 (read off the left-hand scale), thrifty scheduling drastically reduces the probability of excess startup delay, meaning a delay that is greater than the acceptable delay value. Each point on the curve compares the probability of excess delay using greedy scheduling to the probability of excess delay using thrifty scheduling with the specified acceptable delay value. The reduction is calculated as one minus the ratio of these two probabilities, computed over all schedule loads from zero to the rated load of the system. Rated load is defined as the load at which the mean startup delay equals the acceptable startup delay. The effectiveness at reducing excess delay increases with increasing acceptable delay.

As shown by the dashed line in Figure 8 (read off the right-hand scale), thrifty scheduling enables a small increase in the usable capacity of the system. Thrifty scheduling decreases mean startup delay at high system loads, which allows the rated load to increase without increasing mean startup delay. By defining rated load as the load at which mean startup delay equals acceptable startup delay, each point on the curve compares the rated load using greedy scheduling to the rated load using thrifty scheduling with the specified acceptable delay. The increase is calculated as the ratio of these two load values minus one. The maximum increase in usable system capacity is 2.6% at an acceptable startup delay of 9 slots.

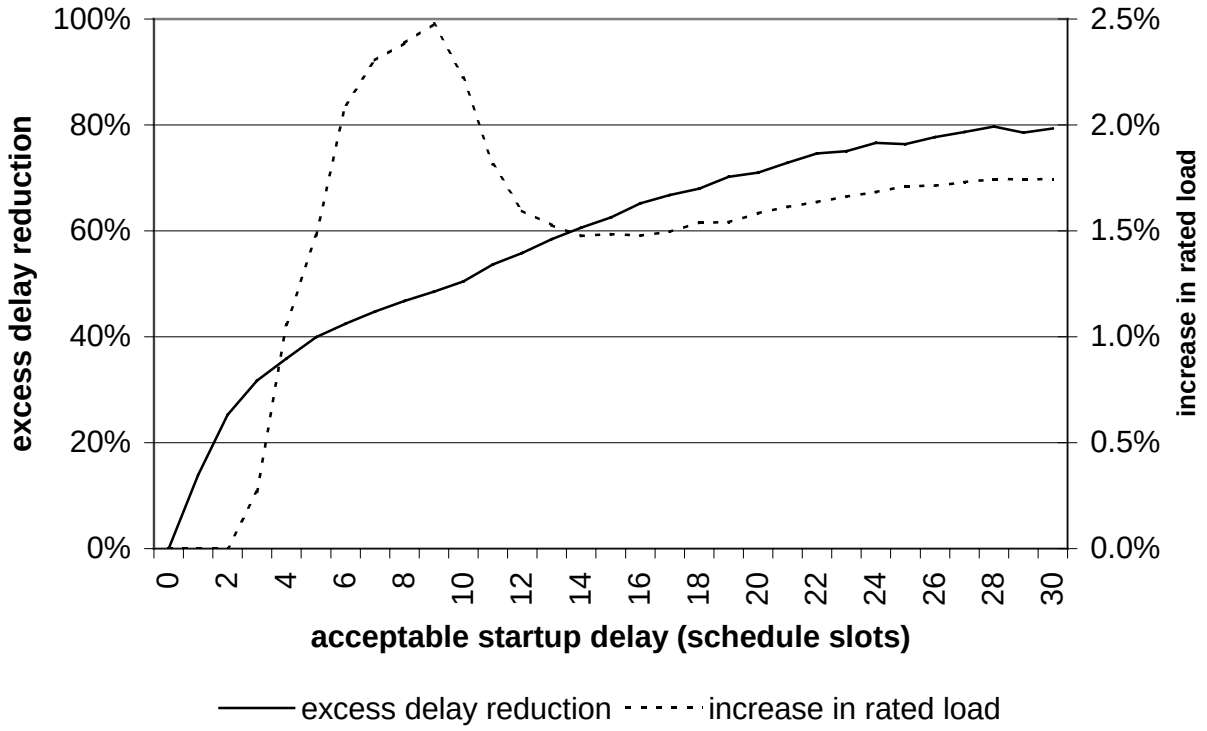

Figure 8: Effectiveness of thrifty scheduling vs. acceptable startup delay

Figures 9 and 10 show that thrifty scheduling is most effective at eliminating excess startup delay at low schedule loads. For these figures, we compared the responsiveness of the original (greedy) scheduling algorithm to that of the thrifty scheduling algorithm with an acceptable startup delay of 10 schedule slots (1.38 seconds). The load-up series was repeated 600,000 times for each case. Figure 9 plots, at each load point, the frequency of stream starts with unacceptably high delays. At all loads, the thrifty algorithm yields a lower excess delay frequency than the greedy algorithm. Figure 10 shows the reduction in the probability of excess startup delay, calculated as one minus the ratio of the two probabilities plotted in Figure 9. At schedule loads well beyond the rated load, thrifty scheduling is ineffectual at reducing the likelihood of excess startup delay. At lower loads, thrifty scheduling is progressively more effective. At each load point below 74 streams, fewer than 10 out of the 600,000 greedy runs had any excess delay, so the metric loses statistical significance and is thus not plotted. Over a load range from zero to the rated load of 90%, the reduction in excess delay probability is 51%, as indicated in Figure 8.

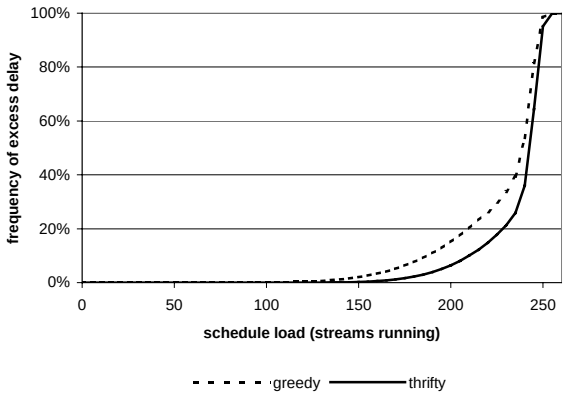

Figure 9: Frequency of excess delay vs. schedule load

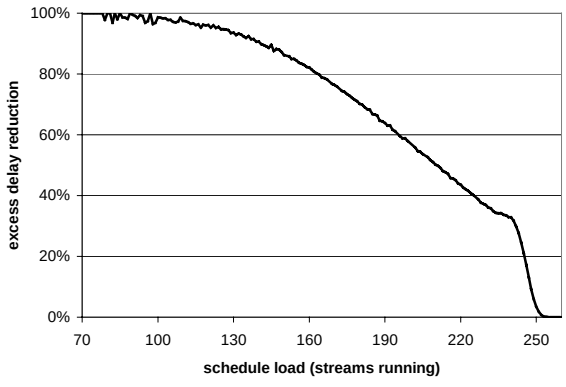

Figure 10: Excess delay reduction vs. schedule load

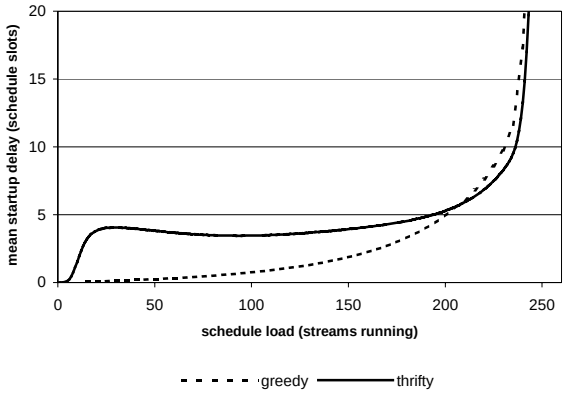

Figure 11: Startup delay vs. schedule load

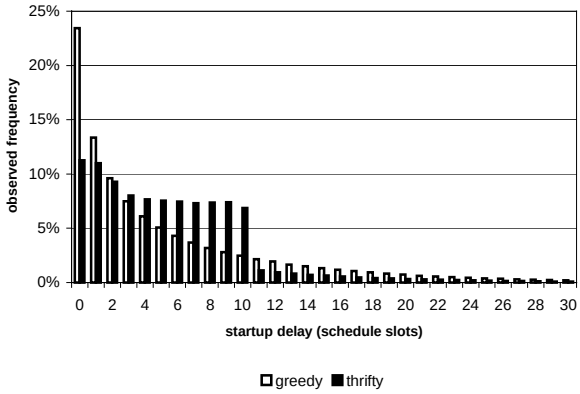

Figure 12: Histogram of startup delay at 207-stream load

Figure 11 shows that thrifty scheduling decreases mean startup delay at high schedule loads. It also shows an increase in mean startup delay at low schedule loads; however, these increased startup delays are all well below the acceptable value of 10 slots. The greedy algorithm exceeds a mean delay of 10 slots after 230 streams, whereas the thrifty algorithm does not exceed this delay until after 236 streams, which is an increase in rated load of 2.2%, as indicated in Figure 8.

Figure 12 shows how thrifty scheduling reduces the probability of unacceptably high startup delays by increasing the probability of acceptably low delays. Whereas Figure 11 shows that the mean startup delays for the two algorithms are the same at a load of 207 streams, this histogram shows that the greedy distribution is far more skewed than the thrifty distribution. The likelihood of exceeding a 10-slot startup delay is 8.7% for thrifty scheduling compared to 18.4% for greedy scheduling, as indicated in Figure 9.

7.2. Experimental validation

We tested both scheduling algorithms in a running Tiger system and verified that the results were not significantly different from those obtained through simulation. This confirmation was reassuring but not surprising, because startup delay is largely insulated from all other performance measures (such as deadline achievement, fault tolerance, and scalability), because we are measuring the end product of the algorithm rather than its speed, and because the core code that is tested by the simulation infrastructure is the same code that is used in the actual Tiger system. Both the simulations and the experiments were driven by manufactured request patterns obtained from random distributions.

We configured a real Tiger system identically to the one that we simulated. Our test system is an ATM Tiger set up for 1 Mbit/s video streams. It uses nine cubs, each of which is a Pentium Pro 200-MHz personal computer with 64 Mbytes of RAM, a PCI bus, a FORE Systems PCA 200E OC-3 ATM adapter, and two Adaptec 3940UW SCSI controllers, each connected to two disk drives; the disks are a mix of Seagate ST12550N, ST32550N, ST32155N, and ST32171N drives. The Tiger controller is also a 200-MHz Pentium Pro. A mix of 66-MHz 486s and 90-MHz Pentiums, each with a FORE Systems ESA200 TAXI NIC, comprise twenty-eight client machines. 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. Each client requested and received either 9 or 10 streams in our experiments.

We ran load-up experiments in the real system for both greedy scheduling and thrifty scheduling with an acceptable delay of 10 slots. We performed each load-up series 15 times, giving a total of 3915 stream starts per experiment. The test client records the time each stream is requested and the time the first block is received, so the total startup delay includes both the scheduling lead time and the network transmission delay (one block play time) in addition to the scheduling startup delay. We subtracted the former values from our recorded delays and divided the result by the block service time in order to estimate the number of slots slipped by the scheduling algorithm.

At a 0.05 level of significance, the experimental results match the simulation results. We combined the results over all load points and partitioned the slot counts into 20 equally sized bins (yielding a minimum expected frequency of 5.2). We then performed a chi-squared test on the null hypothesis that these experimentally obtained startup delays fit the delay distribution obtained through simulation. The greedy results fit with a P-value of 0.28, and the thrifty results fit with a P-value of 0.07.

Our technique for estimating the experimental scheduling startup delay is vulnerable to timing variability due to network jitter and protocol processing, since the data transmission delay adds to the measured startup delay. To analyze the accuracy, we computed the observed startup delays modulo the block service time. If the timing variability were small compared to the block service time, the distribution of these values would have a noticeable peak. However, the distribution was extremely flat, an observation which we quantified by measuring the mean absolute deviation to be over 97% of a quarter range. In view of these measurement errors, the P-values reported above are particularly compelling.

7.3. Explanation of thrifty load-up behavior

Although the shape of the thrifty load-up curve in Figure 11 may seem bizarre, its underlying causes are easily understood. The mean greedy delay increases monotonically as the schedule load increases, because the delays are all necessary ones induced by occupied clusters, and the clustering increases as the schedule load increases. The thrifty curve remains near zero at very low loads, because the few active streams tend to be far apart and thus outside the visible range of the cub performing an insertion. As the schedule begins to fill, the number of streams within each cub's visible range becomes significant, so the cubs delay insertions to reduce clustering, which sharply increases the mean delay. As the schedule continues to fill, there are progressively fewer opportunities to delay insertions for the benefit of schedule quality, so the delay induced by thrifty scheduling decreases. Concurrently, the growing clusters in the schedule cause startup delay to increase. The interplay between the decrease in opportunistic delay and the increase in necessary delay reaches a crossover where the delay attains a local minimum, after which it rises monotonically with schedule load.

8. RELATED WORK

Thrifty scheduling improves video responsiveness by changing the distribution of startup delays. An alternate mechanism for improving startup delay is to dynamically adjust the resolution of video streams[7], in order to allow the video server to reduce its effective load and thus be more responsive. This technique is somewhat analogous to progressive image resolution[12,17], commonly used by web browsers to reduce the apparent delay due to image download. Like thrifty scheduling, progressive image resolution does not actually reduce the waiting time, but it gives the appearance of improved responsiveness.

Pyramid broadcasting[1,10,16] is a technique for reducing the startup delay in near-video-on-demand systems by employing multiple parallel channels for each stream. Ohmura *et al.*[15] explored a method of dividing time slots in a scheduled video server to reduce startup delay.

Thrifty scheduling improves the responsiveness of video distribution without actually reducing the mean response time. Duis and Johnson[8] described several techniques for improving user-interface responsiveness without improving system performance, noting the value in increasing the responsiveness of some operations at the expense of others. A realization of this principle in a fundamental data structure is Robin Hood hashing[5], a mechanism for minimizing the search-time variance of a linear-probing hash table without affecting the mean value of the search time. This work has a peculiar relevance to thrifty scheduling in Tiger, since greedy schedule insertion is isomorphic to standard linear probing in a hash table, and if active streams could be relocated within the Tiger schedule, then the procedure of Robin Hood hashing could be applied to yield an optimal thrifty scheduling algorithm.

9. SUMMARY, CONCLUSIONS, AND FUTURE WORK

The Tiger video fileserver is a distributed, scalable system that guarantees reliable video delivery to a large number of viewers. However, the resource schedule that it employs is vulnerable to clustering, which can induce large startup delays with significant probability. It is possible to reduce schedule clustering by postponing the start of requested streams to more suitable positions in the schedule. Thrifty scheduling accomplishes this beneficial postponement by defining a delay threshold below which startup delays are acceptable. Small startup delays are extended up to the acceptable delay value in order to improve the quality of the schedule for future stream starts.

We found that thrifty scheduling improves responsiveness by decreasing the variability of video stream startup delay and reducing the likelihood of excessively high startup delay. It also allows the rated load of the system to be increased without increasing the mean startup delay.

The Tiger schedule is implemented in a distributed fashion, in which each cub keeps track of only a portion of the schedule. This complicates thrifty scheduling because the algorithm must make decisions based upon purely local data and because the cub cannot assign an optimal slot at the time of the stream request. A distributed version of thrifty scheduling makes conservative assumptions about the portions of the schedule that it cannot see, and it employs a scalable algorithm for deciding when to admit a new stream.

There is current research into generalizing the Tiger architecture to efficiently support streams of multiple bandwidths. The intention is to maintain a constant block play time for all streams but to allow the block size to vary according to the stream bandwidth. There are a number of research challenges to developing such a system, but we do not expect that adapting thrifty scheduling will be a particularly formidable task. The thrifty scheduling algorithm will surely be quite different, but it seems plausible that the thrifty scheduling strategy will be the same and the benefits just as valuable.

ACKNOWLEDGEMENTS

We would like to thank the NetShow™ Theater Server development team for the use of their equipment and talents in performing the experiments for the performance measurements, especially Aamer Hydrie for his assistance with the data collection and for his help, along with Chih-Kan Wang and Bill Schiefelbein, in resolving problems with the experimental system. We extend special thanks to Erik Hedberg for his invaluable support in configuring a Tiger development environment and test system. We would also like to thank Laura Bain for her assistance in searching the corpus of technical literature for related research, Mike Calligaro for his service in translating the text of reference 15, and Rick Rashid and our anonymous reviewers for their helpful suggestions on improving the presentation of our work.

REFERENCES

1. C. C. Aggarwal, E. L. Wolf, P. S. Yu, "A permutation-based pyramid broadcasting scheme for video-on-demand systems," *IEEE International Conference on Multimedia Computing and Systems*, pp. 118-126, Jun 1996.
2. S. Berson, S. Ghandeharizadeh, R. Muntz, X. Ju, "Staggered striping in multimedia information systems," *ACM SIGMOD '94*, pp. 79-90, 1994.
3. W. Bolosky, J. Barrera III, R. Draves, R. Fitzgerald, G. Gibson, M. Jones, S. Levi, N. Myhrvold, R. Rashid, "The Tiger video fileserver," *6th NOSSDAV*, Zushi, Japan, Apr 1996.
4. W. Bolosky, R. Fitzgerald, J. Douceur, "Distributed schedule management in the Tiger video fileserver," *16th ACM SOSP*, pp. 212-223, Oct 1997.
5. P. Celis, P-A. Larson, J. I. Munro, "Robin Hood hashing," *26th annual Symposium on Foundations of Computer Science*, pp. 281-288, Oct 1985.
6. E. Chang, A. Zakhor, "Disk-based storage for scalable video," *IEEE Transactions on Circuits and Systems for Video Technology*, Vol. 7, No. 5, pp. 758-770, Oct 1997.
7. T. C. Chiueh, R. Katz, "Multi-resolution video representation for parallel disk arrays," *ACM Multimedia '93*, New York, pp. 401-409, Aug 1993.
8. D. Duis, J. Johnson, "Improving user-interface responsiveness despite performance limitations," *IEEE Computer Society Intl. Conference*, pp. 380-386, Feb 1990.
9. C. S. Freedman, J. Burger, D. J. DeWitt. "SPIFFI – a scalable parallel file system for the Intel Paragon," *IEEE Trans. on Parallel and Distributed Systems*, 7(11), pp. 1185-1200, Nov 1996.
10. L-S Juhn, L-M Tseng, "Harmonic broadcasting for video-on-demand service," *IEEE Transactions on Broadcasting*, Vol. 43, No. 3, pp. 268-271, Sep 1997.
11. D. E. Knuth, *The Art of Computer Programming, Volume 3: Sorting and Searching*, Addison-Wesley, 1973.
12. H. Lohscheller, "A subjectively adapted image communications system," *IEEE Transactions on Communications*, Vol. COM-32, No. 12, pp. 1316-1322, Dec 1984.
13. J. Nielsen, *Usability Engineering*, Academic Press, 1993.
14. T. N. Niranjan, T. Chiueh, G. A. Schloss, "Implementation and evaluation of a multimedia file system," *IEEE International Conference on Multimedia Computing and Systems*, pp. 269-276, Jun 1997.
15. T. Ohmura, T. Hiota, M. Nakanishi, H. Oka, "Design and implementation of the video network server," *Transactions of the Institute of Electronics, Information and Communication Engineers* D-II, Vol. J79, No. 4, pp. 626-633, Apr 1996.
16. S. Viswanathan, T. Imielinski, "Metropolitan area video-on-demand service using pyramid broadcasting," *Multimedia Systems*, Vol. 4, pp. 197-208, 1996.
17. G. K. Wallace, "The JPEG still picture compression standard," *Communications of the ACM*, Vol. 34, No. 4, pp. 30-44, Apr 1991.