BubbleUp : Low Latency Fast-Scan for Media Servers

Edward Chang and Hector Garcia-Molina
Department of Computer Science
Stanford University
{echang, hector}@cs.stanford.edu

Abstract

Interactive multimedia applications require fast response time. Traditional disk scheduling schemes can incur high latencies, and caching data in memory to reduce latency is usually not feasible, especially if fast-scans need to be supported. In this study we propose a disk-based solution called BubbleUp. It significantly reduces the initial latency for new requests, as well as for fast-scan requests. The throughput of the scheme is comparable to that of traditional schemes, and it may even provide better throughput than mechanisms based on elevator disk scheduling. BubbleUp incurs a slight disk storage overhead, but we argue that through effective allocation, this cost can be minimized.

Keywords: multimedia, disk scheduling, memory utilization, initial latency.

1 Introduction

In recent years substantial effort has been devoted to the design of video servers that meet the continuous real-time deadlines of many clients. Although progress has been made regarding the throughput of such servers, one of the major VCR-like functions, fast-scan (both forward and backward) has not received much attention. A fast-scan operation allows a viewer to jump to any segment of a playing video. A media server can support fast-scan in two ways. One, the media server provides thumb-nail pictures to allow the viewer to select a video sequence to jump to. Two, the server maintains a separate fast version of a clip [2, 14]. When a viewer performs a fast-scan, the video server switches playback to the fast version and returns to the regular version when the operation ends.

One of the most critical performance requirement for a fast-scan operation is fast response time. For example, consider a video game in which at each step the player's actions determine what short video sequence to play next. Clearly, we do not want the player to have to wait a significant amount of time before each video scene starts. Another application that requires low latency is hypermedia documents, as found in the World Wide Web or a Digital Library. Here a user may examine a web page that contains links to a variety of other pages, some of which may be multimedia presentations. Even in movie-on-demand applications where a few minutes' delay before a new multi-hour movie starts may be acceptable, when the viewer decides to fast-scan, response time should be very low.

In this paper, we focus on minimizing a media server's latency, both for an initial request and for a fast-scan request of an ongoing presentation. We measure this initial latency as the time between the request arrival and the time when the data is available in the server's main memory. The initial latency depends how fast the disk arm can start transferring data for the request. For current disk scheduling algorithms the initial latency can be relatively high. For example, most media server designs use an elevator-like disk scheduling policy that we call *Sweep* in this paper [7, 8, 13, 15, 17, 19]. The Sweep policy amortizes the latency of the disk among the requests it services to reduce the seek overhead. However, if a new request accesses a video segment on the disk that has just been passed by the disk arm, the request must wait until the disk arm finishes the current sweep and returns to the data segment in the next sweep. This worst initial latency can be tens of seconds depending on the data layout and the disk configuration [6, 8, 9]. To work around this problem, memory caching techniques have been suggested. However, if a viewer is allowed to fast-scan to any segment of the video, then the entire video must be kept in memory, something that is typically too expensive.

In this study, we propose a resource management scheme, named *BubbleUp*, that minimizes the initial latency for servicing a newly arrived or a fast-scan request. BubbleUp works with a disk scheduling policy *Fixed-Stretch* that gives the disk arm the freedom to move to any disk cylinder before each IO. At first glance, giving the disk arm more freedom seems to be a bad idea, since the increased disk latency may degrade disk bandwidth. However, *Fixed-Stretch* also spaces out IO requests more regularly, and this significantly reduces the memory requirements. Thus, it turns out that the increase in the disk latency is compensated by the more efficient memory use, so overall throughput does not suffer (and may be better in some cases). In particular, our results will show that Fixed-Stretch supports comparable throughput to Sweep and another scheme called Group Sweeping Scheme (GSS) [20]. Furthermore, the freedom to schedule IOs in any order makes possible our latency-reducing BubbleUp policy.

BubbleUp services requests in cycles, where each cycle is divided into equal duration slots. An ongoing presentation performs one IO per cycle, using up one of the slots in that cycle. The key to BubbleUp is that, as it schedules IOs into slots, it attempts to keep the next slot open to quickly handle an "unexpected" new request or fast-scan. This is analogous to how a person would schedule office work when expecting an important visitor. Here the slots are hours (say) and a cycle is a working day. Suppose that at 9am the visitor has not shown up, there is no work scheduled for 9am, but there is work scheduled for 10am. In this case it makes sense to perform the 10am work at 9am, thus making the 10am slot free to handle the visitor if he/she shows up then. In this way, the 9am free slot "bubbles up" to 10am. If the visitor still does not show up at 10am, we may move 11am work to 10am, bubbling up our open slot to 11am.

Of course, the situation is more complex with IO requests than with office work because the data read in one slot for a particular presentation is supposed to sustain that presentation until the same slot in the next cycle. If we move an IO from one slot to another within a cycle, BubbleUp must compensate by reading less or more data, or by changing the slot for this presentation in future cycles. The details as to how BubbleUp compensates and manages to give each presentation a steady stream of data are given in Section 3. Using these techniques, BubbleUp manages to keep slots open in the immediate future. This makes it possible to start a new presentation or a fast-scan very quickly (e.g., under a quarter of a second even with a heavy load), scheduling it into one of these open slots.

Supporting fast-scan operations presents a tradeoff with throughput. That is, a fast-scan operation may require two IOs for a single presentation in a cycle: one before the fast-scan request is made (this reads data from the "old" part of the video; some of this data will be discarded when playback of the "new" part starts), and one after the fast-scan request (to read in the new data). This implies that the extra IO bandwidth will not be available to service a brand new request that arrives. Thus, given some maximum number of IOs one wishes to support during a cycle, some of the capacity can be reserved for new requests and some for fast-scans of ongoing requests. In this paper we show that the memory required for a fast-scan operation is actually less than that required for a new request. Thus, by reducing the allowable number of new requests, one can actually support a larger number of fast-scans. As we will see, BubbleUp allows the media server to *dynamically* shift resources from supporting new requests to supporting fast-scans and vice versa without any transitional delay.

The rest of this paper consists of five sections. Section 2 describes the Fixed-Stretch disk scheduling scheme. Section 3 presents policy BubbleUp and examples. Sections 4 and 5 quantitatively analyze the performance of BubbleUp and compare it with other competing schemes. Finally, we offer our conclusions in Section 6.

2 Fixed-Stretch

We assume that the media server services requests in cycles. During a service cycle (time T), the server reads one *segment* of data for each of the requested *streams*, of which there can be at most N_{Limit}. We assume that each segment is stored contiguously on disk. The data for a stream is read (in a single IO) into a memory buffer for that stream, which must be adequate to sustain the stream until its next segment is read.

In a feasible system, the period T must be large enough so that in the worst case all necessary IOs can be performed. Thus, we must make T large enough to accommodate N_{Limit} seeks and transfer N_{Limit} segments. Fixed-Stretch achieves this by dividing a service cycle T into N_{Limit} equal service slots. Since the data on disk for the requests are not necessarily separated by equal distance, we must add time delays between IOs to make all service slots last for the same amount of time. For instance, if the seek distances for the IOs in a cycle are cyl_1, cyl_2,..., and $cyl_{N_{Limit}}$ cylinders, and cyl_i is the maximum of these, then we must separate each IO by at least the time it takes to seek to and transfer this maximum i^{th} request. Since in the worst case the maximum cyl_i can be as large as the number of cylinders on the disk (CYL), Fixed-Stretch uses the worst possible seek distance CYL and rotational delay, together with a segment transfer time, as the universal IO separator, Δ, between any two IOs. We use $\gamma(CYL)$ to denote the worst case seek and rotational delay. If the disk transfer rate is TR, and each segment is S bytes long, then the segment transfer time is S/TR, so $\Delta = \gamma(CYL) + S/TR$.

The length of a period, T, will be N_{Limit} times Δ. Figure 1 presents an example where $N_{Limit} = 3$. The time on the horizontal axis is divided into service cycles each lasting T units. Each service cycle T (the shaded area) is equally divided into three service slots, each lasting Δ units (delimited by two thick up-arrows). The vertical axis in Figure 1 represents the amount of memory utilized by an individual stream.

Fixed-Stretch executes according to the following steps:

1. At the beginning of a service slot (indicated by the thick up-arrow in Figure 1), set the *end of slot timer* to

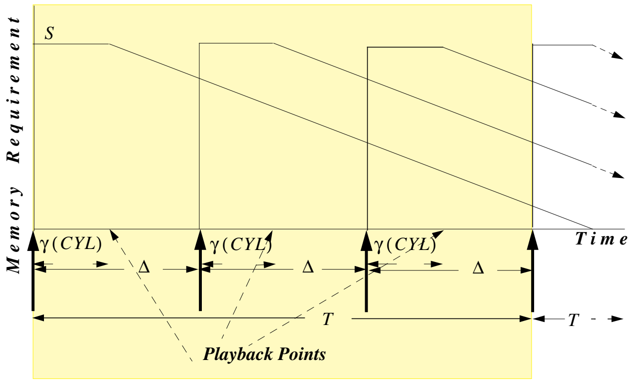


Figure 1: Service Slots of Fixed-Stretch

expire in Δ.

2. If there is no request to be serviced in the service slot, skip to Step 6.

3. Allocate S amount of memory for the request serviced in this time slot.[1]

4. Set the *IO timer* to expire in $\gamma(CYL)$, the worst possible seek overhead, and start the disk IO. Since the actual seek overhead cannot exceed $\gamma(CYL)$, when the *IO timer* expires the data transfer must have begun.

5. When the *IO timer* expires, the playback starts consuming the data in the buffer (indicated by the "playback point" pointers in Figure 1), and the memory pages are released as the data is consumed.

6. When the *end of slot timer* expires, the data transfer (if issued in Step 4) must have completed.[2] Go to Step 1 to start the next service slot.

As its name suggests, the basic Fixed-Stretch scheme has two distinguishing features:

- Fixed-order scheduling: A request is scheduled in a fixed service slot from cycle to cycle after it is admitted into the server. For instance, if a request is serviced in the k^{th} slot when it first arrives, it will be serviced in the same k^{th} slot in its entire playback duration, regardless if other requests depart or join the system. (As we will see in Section 3, the "fixed" scheduling may be changed by policy BubbleUp.)

- Stretched out IOs: The allocated service slot assumes the worst possible disk latency $\gamma(CYL)$ so that the disk arm can move freely to any disk cylinder to service any request. This property ensures that the fixed-order scheduling is feasible no matter where the data segments are located on the disk.

As mentioned in Section 1, Fixed-Stretch appears to be inefficient since it assumes the worse seek overhead between IOs. However, Fixed-Stretch uses memory very efficiently because of its very regular IO pattern, and this compensates for the poor seek overhead. In Section 4 we analyze the memory requirement of Fixed-Stretch and compare its performance with other disk scheduling policies (e.g., Sweep and GSS). We show that Fixed-Stretch achieves comparable throughput to the other schemes.

3 Policy BubbleUp

Policy BubbleUp builds upon Fixed-Stretch to minimize *initial latency*. We define initial latency to be the time between the arrival of a *single* new request (when the system is unsaturated) and the time when its first data segment becomes available in the server's memory. In computing the initial latency we do not take into account any time spent by a request waiting because the system is saturated (with N_{Limit} streams), as this time could be unbounded no matter what scheduling policy is in place. In other words, our focus is on evaluating the worst initial delay when both disk bandwidth and memory resources are available to service a newly arrived or a fast-scan request.

To simplify our discussion we temporarily assume that each media (i.e., all of its segments) is laid out contiguously on the disk. We relax this restriction in Section 3.5.

BubbleUp minimizes the initial latency by rescheduling the requests in Fixed-Stretch's service slots and by carefully managing the amount of data to retrieve for each request to prevent both data overflow and underflow. Table 1 summarizes the parameters defined so far, together with other parameters that will be introduced later.

3.1 Reducing Initial Latency

We use an example to illustrate how BubbleUp reduces initial latency. Assume that a media server supports up to three requests ($N_{Limit} = 3$) in a period T, as depicted in Figure 1. To service three requests in T, BubbleUp divides the period into three service slots, s_1, s_2, and s_3, each lasting time Δ.

Table 2 shows a sample execution for three requests R_1, R_2, R_3. Each row of the table represents the execution for one service slot. The first column identify the time involved. Each slot *instance* is labeled as Δ_i to remind us that its duration is Δ time units. Note that the first period or cycle has three slot instances (Δ_1, Δ_2, Δ_3). The first slot in a cycle, s_1, occurs in instances Δ_1, Δ_4, Δ_7, ... The last three columns in Table 2 show the requests that are

[1] When an IO is initiated, the physical memory pages for the data it reads may not be contiguous due to the way buffers are shared. There are several ways to handle these IOs. One idea is to map the physical pages to a contiguous virtual address, and then initiate the transfer to the virtual space (if the disk supports this). Another idea is to break up the segment IO into multiple IOs, each the size of a physical page. The transfers are then chained together and handed to an IO processor or intelligent DMA unit that executes the entire sequence of transfers with the same performance as a larger IO. Other ideas are discussed in [11].

[2] The accuracy of the timers used by Fixed-Stretch can be tuned periodically by cross-checking the amount of data in the stream buffers.

Parameter	*Description*
DR	Data display rate, Mbps
TR	Disk transfer rate, Mbps
CYL	Number of cylinders on disk
$\gamma(d)$	Function computes seek overhead
S	Segment size, without fast-scans
S'	Segment size, with fast-scans
T	Service time for a round of requests
Δ	Service time for a service slot
T_{Seek}	Total worst seek (and rotational) time in T
$T_{Transfer}$	Total worst data transfer time in a period
N	Number of requests being serviced
n	No. of fast-scans serviced in the cycle
θ	The service slot where fast-scan initiated
N_{Limit}	System enforced limit on no. of requests
M_{Limit}	System enforced limit on no. of fast-scans
R_i	i^{th} request
s_i	i^{th} service slot
cyl_i	Seek distance of the i^{th} request
Mem_{Min}	Minimum memory requirement, MBytes
$T_{Latency}$	Initial Latency, seconds

Table 1: Parameters

currently scheduled for each slot in a cycle.

For our example, let us assume that request R_1 arrives before Δ_1, R_2 arrives during Δ_2, and R_3 during Δ_4. When R_1 arrives the system is empty and it is scheduled into the first slot s_1. The following steps then occur under policy BubbleUp:

- Δ_1: During the first instance of s_1, Δ_1, request R_1 is serviced; this is indicated by the asterisk next to R_1. (Note that the asterisk moves from slot to slot as time progresses.) The amount of data retrieved for R_1 is S.
- Δ_2: The current slot is s_2, and no new request has arrived before the start of this time unit. Instead of staying idle, we service R_1 *again* in s_2. The amount of data retrieved for R_1 is $S/3$, since only $S/3$ has been consumed from the last "playback point" of R_1 to the playback point of this slot instance. Note that in Row 2 of Table 2, R_1 now has moved to s_2 and s_1 is empty. Empty slots s_3 and then s_1 are (roughly) Δ and 2Δ time units away, ready to accept any new requests that might arrive. This is the goal of the BubbleUp policy: trying to keep open slots as close time-wise as possible.
- Δ_3: The current slot is s_3, and request R_2 has just arrived (during Δ_2). We service R_2 immediately in s_3.
- Δ_4: Slot s_1 is the current slot, no new request has arrived before the start of this time unit. Policy BubbleUp selects the next future request that would be due for service, R_1 in this case, and services it early. Enough data is read in ($2S/3$) to fill up the R_1 buffer, and now R_1 has moved to slot s_1. Again, we have freed the following slot, s_2, to be able to service some new request that might arrive.

Time	s_1	s_2	s_3
Δ_1	* R_1		
Δ_2		* R_1	
Δ_3		R_1	* R_2
Δ_4	* R_1		R_2
Δ_5	R_1	* R_3	R_2

Table 2: BubbleUp Example

- Δ_5: The current slot is s_2, and request R_3 has arrived (during Δ_4). We service R_3 immediately. If we had not swapped R_1 out of this slot in Δ_4, R_3 would have had to wait two slots before receiving service.

Policy BubbleUp "bubbles up" empty slot instances in the future so they occur in the next slot instance. Thus, if there is a free slot, it always occurs in the very next time slot, and the maximum delay a new request faces is limited. In particular, the worst delay occurs if the new request arrives just after a slot has started. It then has to wait for the next slot ($\Delta = \gamma(CYL) + S/TR$), plus the time for its own seek to be completed ($\gamma(CYL)$) to reach the playback point. Thus, the worst case latency:

$$T_{Latency} = 2 \times \gamma(CYL) + S/TR, \tag{1}$$

is independent of N_{Limit}. When two new requests arrive at the server and the system is currently servicing fewer than $N_{Limit} - 1$ requests, we can still service these two new requests immediately in consecutive slots. The latency for the second request is larger than the first one by $\gamma(CYL) + S/TR$.

3.2 Supporting Fast-Scans

A fast-scan operation is initiated by an ongoing request that decides to jump to another segment of the video. Fast-scans can be supported either by providing thumb-nail pictures to allow the viewer to select a video sequence to jump to, or by playing a separate fast (skipping frames) version of the video that leads to the desired segment of the film [2, 14]. Since a fast-scan operation is initiated by an interactive user, having fast response time is critical, as we have discussed.

If open slots are available, we can treat a fast-scan request just like a new request, scheduling for the next open slot (and removing the old request from the slot it was assigned to). However, if no slots are available, then the fast-scan cannot be serviced. This is illustrated by the following example, which also motivates an extension to BubbleUp. We continue with the previous example, starting with from the 5th time unit (Δ_5).

- Δ_5: The current slot s_2 has been used to service request R_3. The amount of data in the buffer for R_1,

Time	s_1		s_2		s_3	
Δ_5		R_1	*	R_3		R_2
Δ_6		R_2		R_1	*	R_3
Δ_7	*	R_2		R_1		R_3
Δ_8		R_2	*	R_1		R_3

Table 3: Fast-Scan Example

R_2, and R_3 at the playback point of this slot instance is $\frac{2S}{3}$, $\frac{S}{3}$, and S respectively.

- Δ_6: The current slot is s_3. The slot is supposed to service R_2, but a fast-scan request has been issued by the viewer of R_3 in the previous time unit. If we do not service the fast-scan request until slot s_2, where R_3 is next scheduled, the initial latency will be three time slots. On the other hand, if we service the fast-scan request in s_3, the delay service will cause data underflow for R_2 and R_1.

To service the fast-scan request right away in s_2 as well as to avoid data underflow for the delayed requests, BubbleUp must increase the segment size from S to S'. Let us assume that we only wish to service (with low latency) one fast-scan operation per service cycle. In this case, the new segment size S' must be able to sustain the playback for the possible delay of one extra service slot. While an expression for S' is derived in Section 4, for this example it suffices to say that S' should be large enough to sustain the playback time of 4Δ. Assuming that the system now reads segments of size S', we repeat the example from the beginning of Δ_5. Table 3 shows the execution; the steps are as follows:

- Δ_5: The current slot s_2 has been used to service request R_3. The amount of data in the buffer for R_1, R_2, and R_3 at the playback point of Δ_5 is $\frac{3S'}{4}$, $\frac{S'}{2}$, and S' respectively. Note that the data in R_2's buffer can last for another two slot instances rather than one.
- Δ_6: The current slot is s_3. The slot is supposed to service R_2, but a fast-scan request has been issued in the previous slot by the R_3 viewer. BubbleUp services R_3 immediately in s_3. Requests R_2 and R_1 are delayed by one slot to s_1 and s_2. However, since S' is larger, both R_1 and R_2 can be delayed without causing data underflow. The amount of data in the buffer for R_1, R_2, and R_3 at the playback point of Δ_6 is $\frac{S'}{2}$, $\frac{S'}{4}$, and S'. Note that since the system supports only up to one fast-scan operation per cycle (this was our design choice), the amount of data in the buffer for R_1 and R_2 is insufficient to sustain a second delay.
- Δ_7: The current slot is s_1. BubbleUp transfers S' amount of data for R_2. The amount of data in the buffer at the playback point of Δ_7 for R_1, R_2, and R_3 is $\frac{S'}{4}$, S', and $\frac{3S'}{4}$. Note that, the amount of data in the buffer for R_1 is still insufficient to take another delay. BubbleUp still cannot take another fast-scan request at this point.

Time	s_1		s_2		s_3		s_4	
Δ_8		R_2	*	R_1		R_3		*NA*
Δ_9		R_2		R_1	*	R_4		R_3
Δ_{10}		R_2		R_1		R_4	*	R_3
Δ_{11}	*	R_1				R_4		R_3

Table 4: Switch Example

- Δ_8: The current slot is s_2. BubbleUp transfers S' amount of data for R_1. The amount of data in the buffer at the playback point of this slot instance for R_1, R_2, and R_3 is S', $\frac{3S'}{4}$, and $\frac{S'}{2}$. The system has now recovered and is ready to service another fast-scan request, if necessary.

The above example shows that BubbleUp can enlarge the segment size to S' so that the service for the scheduled requests can be postponed to support fast-scan operations. The more fast-scans the media server is designed to support in a service cycle, the larger S' needs to be.

3.3 Supporting New Requests vs. Fast-Scans

BubbleUp can elect to use its resources to service new requests or fast-scan operations. Continuing with the example from Section 3.2, we extend the execution by three more slot instances. Let a new request R_4 arrive in Δ_8 and let request R_2 depart the server (either the viewer stops the playback or the playback ends) in Δ_{10}. Table 4 shows the sample execution from slot Δ_8 to Δ_{11}. Time slots marked with "NA" do not exist.

- Δ_8: At the playback point of slot Δ_8, the amount of data in the buffer for R_1, R_2, and R_3 is S', $\frac{3S'}{4}$, and $\frac{S'}{2}$. The system can accept a new request or a fast-scan request in the next time unit.
- Δ_9: The current slot is s_3, and a new request R_4 has arrived (during Δ_8). If the media server decides not to support the additional request, R_4 is turned away. In this example, suppose the server decides to increase its throughput. The server sets N_{Limit} to 4 and opens up a new service slot s_4. The server uses the current slot s_3 to service R_4 immediately. R_3 is delayed into slot s_4 that previously did not exist. Since the amount of data in the buffer is sufficient for the rest of the requests to be delayed by one time slot, there is no concern about "jitter." The amount of data in the buffer for R_1, R_2, R_3, and R_4 at the playback point is $\frac{3S'}{4}$, $\frac{S'}{2}$, $\frac{S'}{4}$, and S'.
- Δ_{10}: The current slot is s_4. Request R_3 is serviced in s_4 to fill its buffer up to S'. The amount of data in the buffer for R_1, R_2, R_3, and R_4 at the playback point is $\frac{S'}{2}$, $\frac{S'}{4}$, S', and $\frac{3S'}{4}$.
- Δ_{11}: The current slot s_1 is scheduled to service R_2, but R_2 has departed. The server can either close down s_4,

so the system returns to the mode where it can support one fast-scan, or the server can remain at $N_{Limit} = 4$. In this example, we assume that the server makes the latter choice, Keeping N_{Limit} at 4, we bubble R_1 up to s_1 and keep s_2 open to service the next potential new request.

The example demonstrates that BubbleUp has the flexibility to shift resources from supporting fast-scans to supporting additional streams, and vice versa.

3.4 BubbleUp Specification

We now describe policy BubbleUp formally. Let s_i, for $i = 0, 1, 2, ..., N_{Limit} - 1$, denote the service slots. The slots are $\Delta = T/N_{Limit}$ apart. Let ψ denote the pointer to the current service slot that is being serviced by the disk, and θ denote the pointer to the service slot in which the request initiates the fast-scan operation. Let M_{Limit} be the maximum number of fast-scans permitted in a service cycle T, and n be the number of fast-scans that have been serviced in the cycle ($n \leq M_{Limit}$). For bookkeeping, BubbleUp maintains two sets of data: 1) $s_i.R$ records the request scheduled to be serviced in the i^{th} service slot, and 2) $s_i.N$ records the number of times the request that is currently scheduled in the i^{th} slot has been delayed (bumped) in the cycle due to fast-scans ($s_i.N \leq M_{Limit}$). We use λ_{new} to represent the newly arrived request. Figure 2 shows policy BubbleUp.

Every $\Delta = T/N_{Limit}$ seconds, the system services a slot. The first key in policy BubbleUp is the step where the current service slot s_ψ is empty ($s_\psi.R = nil$) and there is neither a new nor a fast-scan request (step 2(c) in Figure 2). When this happens, policy BubbleUp swaps slot s_ψ with the next occupied slot, and makes the empty slot available in the future. Procedure *Swap* in Figure 4 describes how a swap slot is chosen and the amount of data to retrieve for the swapped request.

The second key in policy BubbleUp is the steps where the scheduled requests are delayed to service a fast-scan request. This can happen either when the system still has empty slots (step 2(b)) or when the system is fully loaded (step 3(b)), as long as the number of fast-scans issued in the cycle has not exceeded the limit ($n < M_{Limit}$). In both steps, procedure *Bump* is invoked to push some scheduled requests backward so that the fast-scan request can be serviced promptly.

Figure 3 describes how procedure *Bump* works. Procedure *Bump* pushes the requests in front of the service slot that initiates the fast-scan request (s_θ) one slot back. This not only opens up the current slot to service the fast-scan request, but also garbage-collects the slot in which the request issues the fast-scan. Another important function of procedure *Bump* is to keep track of how many times a request has been bumped in a service cycle (stored in $s_i.N$). This information is used in procedure *Swap* to determine how much data to retrieve for the swapped request (discussed shortly).

Policy BubbleUp

- Initialization:
  - For i from 0 to $N_{Limit} - 1$:
    - ∗ $s_i.R \leftarrow nil$
    - ∗ $s_i.N \leftarrow 0$
  - $\psi \leftarrow 0$, $n \leftarrow 0$
- At beginning of every slot:
  1. If the playback in s_ψ has ended
     - ∗ $s_\psi.R \leftarrow nil$
  2. If ($s_\psi.R = nil$)
     - (a) If a new request λ_{new} has arrived
       - ∗ $s_\psi.R \leftarrow \lambda_{new}$
       - ∗ Transfer S' for $s_\psi.R$
     - (b) Else if a fast-scan has been initiated by $s_\theta.R$
       - ∗ If $M_{Limit} > n$
         - ∘ Execute Procedure *Bump* (Figure 3)
         - ∘ $n \leftarrow n + 1$
         - ∘ Transfer S' for $s_\psi.R$
       - ∗ Else Queue up $s_\theta.R$
     - (c) Else Execute Procedure *Swap* (Figure 4)
  3. Else ($s_\psi.R \neq nil$)
     - (a) If a new request λ_{new} has arrived
       - ∗ Queue up λ_{new}
     - (b) Else if a fast-scan has been initiated by $s_\theta.R$
       - ∗ If ($\theta = \psi$) Transfer S' for $s_\psi.R$
       - ∗ Else If $M_{Limit} > n$
         - ∘ Execute Procedure *Bump* (Figure 3)
         - ∘ $n \leftarrow n + 1$
         - ∘ Transfer S' for $s_\psi.R$
       - ∗ Else Queue up $s_\theta.R$
     - (c) Else Fill the buffer up to S' for $s_\psi.R$
  4. $\psi \leftarrow (\psi + 1)\ mod\ N_{Limit}$
  5. if $\psi = 0$
     - (a) $n \leftarrow 0$
     - (b) For i from 0 to $N_{Limit} - 1$: $s_i.N = 0$

Figure 2: Policy BubbleUp

Figure 4 describes procedure *Swap* formally. Procedure *Swap* determines which slot to swap in the second step of the execution. The slot to swap is the first occupied slot due up for service. If all slots are empty, then the procedure does nothing. After a swap candidate is chosen, procedure Swap calculates how much data to retrieve for the swapped request. First, procedure Swap determines how many slots (denoted as N_{Slots}) the swapped request has to be moved forward. Next, procedure Swap determines how many times the swapped request has been moved backward due to the fast-scans serviced in the cycle (can be found after 3(a) in $s_\psi.N$). Since the swapped request was scheduled

Procedure Bump

- Variables:
  $* \iota, \kappa, tmp$;
- Input:
  $* \psi, \theta, N_{Limit}, s_i$ for $0 = 1, 2, ..., N_{Limit} - 1$;
- Output:
  $* s_i$ for $i = 0, 2, ..., N_{Limit} - 1$;
- Execution:
  Invariant: if there are empty slots between s_ψ and s_θ, they must be contiguous and start at s_ψ.
  1. $\kappa \leftarrow \theta$
  2. Save the request that issues the fast-scan in tmp
     $tmp.R \leftarrow s_\theta.R$; $tmp.N \leftarrow s_\theta.N$
  3. While ($\kappa \neq \psi$ and $s_k.R \neq nil$) do
     $* \iota \leftarrow (\kappa - 1 + N_{Limit})$ mod N_{Limit}
     $* s_\kappa.R \leftarrow s_\iota.R$; $s_\kappa.N \leftarrow s_\iota.N + 1$
     $* \kappa \leftarrow (\kappa - 1 + N_{Limit})$ mod N_{Limit}
  4. $s_\psi.R \leftarrow tmp.R$; $s_\psi.N \leftarrow 0$

Figure 3: Procedure Bump

to run out of data in $N_{Slots} + (M_{Limit} - s_\psi.N)$ slots away, moving it N_{Slots} forward leaves $S' \times \frac{N_{Slots}+M_{Limit}-s_\psi.N}{N_{Limit}+M_{Limit}}$ amount of data in the buffer. To replenish the buffer to S' requires the data retrieval size to be:

$$S' \times \frac{N_{Limit} - N_{Slots} + s_\psi.N}{N_{Limit} + M_{Limit}}.$$

The value of $s_\psi.N$ is reset at the end of *Swap* (Step 3(e)) since the data in the buffer for the swapped request has been filled up to S' to take up to M_{Limit} bumps.

Policy BubbleUp can be enhanced beyond what we have described here. In particular, instead of checking if $n < M_{Limit}$ before calling procedure *Bump* (under steps 2(b) and 3(b) in Figure 2), BubbleUp can check if each request to be bumped satisfies $s_i.N < M_{Limit}$. Thus, as long as the affected requests have sufficient data in the buffer to be delayed, the fast-scan request can be admitted even though n can be equal to or larger than M_{Limit}. We do not consider this enhancement further in the rest of this paper.

3.5 Data Placement

We have assumed so far that a video is laid out contiguously on the disk in its entirety. However, policy BubbleUp does not require such a stringent data placement policy.

To keep the seek overhead of an IO to at most $\gamma(CYL)$, BubbleUp requires that the data read by each disk IO is physically contiguous on the disk. Since each IO may read variable amounts of a segment, we have to carefully store data on the disk. For example, suppose that we store the segments S'_1, S'_2, S'_3... of a particular presentation in different places on disk. It could be the case that in the first

Procedure Swap

- Variables:
  $* \kappa, N_{Slots}$;
- Input:
  $* \psi, M_{Limit}, N_{Limit}$;
  $* s_i$ for $0 = 1, 2, ..., N_{Limit} - 1$;
- Output:
  $* s_i$ for $i = 0, 2, ..., N_{Limit} - 1$;
- Execution:
  1. $\kappa \leftarrow (\psi + 1)$ mod N_{Limit}
  2. While ($s_\kappa.R \neq nil$ and $\kappa \neq \psi$) do
     $* \kappa \leftarrow (\kappa + 1)$ mod N_{Limit}
  3. If $\kappa \neq \psi$ (swap slot found)
     (a) $s_\psi.R \leftarrow s_\kappa.R$; $s_\psi.N \leftarrow s_\kappa.N$
     (b) $s_\kappa.R \leftarrow nil$; $s_\kappa.N \leftarrow 0$
     (c) Compute swap distance: $N_{Slots} \leftarrow (\kappa - \psi + N_{Limit})$ mod N_{Limit}
     (d) Retrieve $S' \times \frac{(N_{Limit}-N_{Slots}+s_\psi.N)}{N_{Limit}+M_{Limit}}$ amount of data for request $s_\psi.R$.
     (e) $s_\psi.N \leftarrow 0$

Figure 4: Procedure Swap

cycle we read three fourths of S'_1, and then need to read a full segment's worth of data in the second cycle. Thus, in the second cycle we need to read $1/4$ of S'_1 and $3/4$ of S'_2, requiring two IOs. This violates our one IO per request rule.

To remedy this problem, we propose placing data on the disk in contiguous *chunks* following two rules:

1. Each chunk is physically contiguous and is a minimum two segments in size.
2. The last segment of a chunk is always replicated at the beginning of the next chunk. (This rule does not apply to the last chunk.)

To illustrate, consider a video stored in three chunks: chunk A contains segments 1, 2, and 3; chunk B contains segments 3 and 4, and chunk C segments 4, 5, 6, and 7. It is clear that BubbleUp can now read one segment worth of data in a single IO starting at any point of the video. For instance, if it needs to read a fraction of S'_2 and S'_3, it can read from chunk A; if it needs to read a fraction of S'_3 and S'_4 it reads from chunk B.

As we break up a video into more and more chunks, we have higher storage overhead. At the extreme, all chunks contain two segments, and every segment is replicated, for a 100% storage overhead. Thus, we would like to store each video on disk using as few chunks as possible. The following example illustrates a good policy for achieving such allocation. Say we have a 30 segment video to place on disk. We denote the i^{th} segment of the video by V_i.

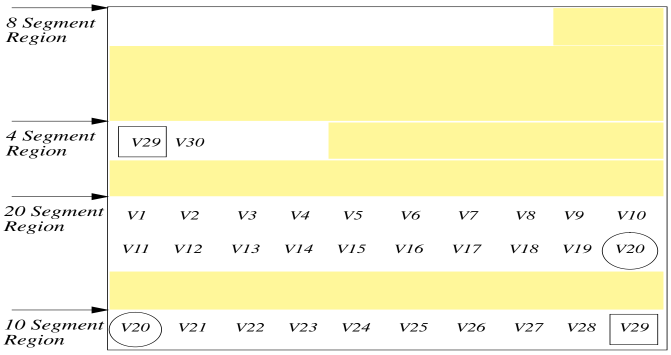


Figure 5: Data Placement

Figure 5 presents the disk layout, with the unshaded regions representing the available space on disk. These free regions can accommodate (from top to bottom) 8, 4, 20, and 10 segments respectively. We first use a "largest fit" policy, using the largest available spaces on disk to hold the largest possible chunks. For the last chunk we use instead a "best fit" strategy in order to reduce disk fragmentation. Figure 5 shows the result of using these policies. The first 20 segments of the video, from V_1 to V_{20} are placed in the largest 20 segment region. Then, we have to replicate V_{20} at the beginning of the second chunk, and place segments V_{20} to V_{29} in the next largest 10 segment region. The last chunk holds a copy of V_{29} together with V_{30}. It is placed in the smallest region that can hold it, i.e., it goes into the 4 segment region.

The "largest fit" policy reduces the number of chunks, decreasing the replication overhead to the minimum possible. The "best fit" policy for the last chunk reduces fragmentation, and is beneficial for future allocations. We believe that under normal circumstances (i.e., a disk that is not terribly fragmented), videos can be stored in small number of chunks, and the replication overhead will be minimal.

4 Analysis

Three factors together determine the memory requirement and hence the throughput of a media server: disk seek overhead, the variability of the IO time, and the degree of memory sharing among the requests. In this section, we analyze these factors for scheme Fixed-Stretch (the underlying scheduling policy of BubbleUp). We also compare it with a representative disk scheduling policy, Sweep, which minimizes disk seek overhead. Finally, we study the impact on throughput of the extra data read when BubbleUp handles fast-scans. In Section 5 we also evaluate the Group Sweeping Scheme (GSS) [20], a scheme that can be thought of as a hybrid between Fixed-Stretch and Sweep. However, due to space limitations, in this section we do not discuss the GSS analysis. (The analysis can be found in [5].)

4.1 Seek Overhead

The throughput analysis for Fixed-Stretch is independent of whether policy BubbleUp is used or not, as long as BubbleUp does not read larger segments for fast-scans. For now, assume that larger segments are not used.

Fixed-Stretch services requests in cycles each lasting T time units. As we have discussed in Section 2, the period T must be large enough so that all necessary IOs can be performed. The period T therefore must be larger than or equal to the worst case seek and transfer times, i.e., $T \geq T_{Seek} + T_{Transfer}$. For optimal performance, however, we take the smallest feasible T value, since otherwise we would be wasting memory resource. That is,

$$T = T_{Seek} + T_{Transfer} = N_{Limit} \times (\gamma(CYL) + S/TR). \tag{2}$$

In a feasible system, the amount of data retrieved in a period must be at least as large as the amount of data displayed. That is, $S \geq DR \times T$, where DR is the display rate. (Recall that Table 1 summarizes our parameters.) We refer to this constraint as *continuity* constraint. However, if we want a stable system, the input rate should equal the output rate, else in every period we would accumulate more and more data in memory. Thus, we have the equation

$$S = DR \times T. \tag{3}$$

Substituting $T = S/DR$ (Eq. 3) into Equation 2, we can solve for S, the segment size needed to support the N_{Limit} requests (without fast-scan):

$$S = \frac{N_{Limit} \times \gamma(CYL) \times TR \times DR}{TR - (DR \times N_{Limit})}. \tag{4}$$

From Equation 4, it is clear that the segment size is directly proportional to the seek overhead function $\gamma(CYL)$, given TR, DR, and N_{Limit}.

For scheme Sweep, since γ is a concave function [10, 16, 19], the largest value of the total seek overhead occurs when the segments are equally spaced on the disk. This means that the worst case seek overhead per request is $\gamma(CYL/N_{Limit})$. Thus, the equations for Sweep are identical to the above, except that $\gamma(CYL)$ is replaced by $\gamma(CYL/N_{Limit})$.

4.2 IO Time Variability

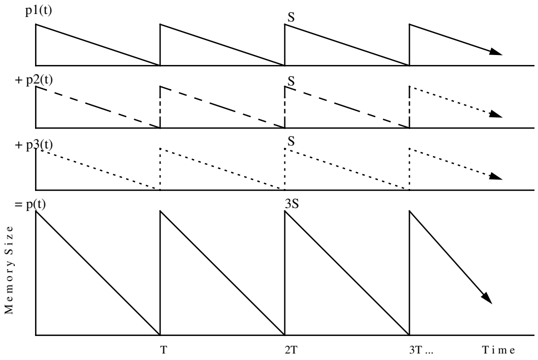
For Fixed-Stretch (without considering fast-scan), the time between servicing IOs for a request is capped by T. As such, S amount of memory can sustain playback without violating the continuity constraint stated in Equation 3. The memory requirement (without sharing of buffers) for Fixed-Stretch is therefore $N_{Limit} \times S$.

For Sweep, however, although in a period we read only S bytes for each stream, it turns out we need a buffer of twice that size for each stream to cope with the variability in read times. To see this, consider a particular stream in progress, where we call the next three disk arm sweeps A, B, and C. Assume that the segments needed by our stream are a, b, and c. It so happens that because of its location on the disk segment, a is read at the beginning of sweep A, while b is read at the end of sweep B, $2 \times T$ time units after a is read. At the point when a is read we need to have in memory $2 \times S$ data, to sustain the stream for $2 \times T$ time.

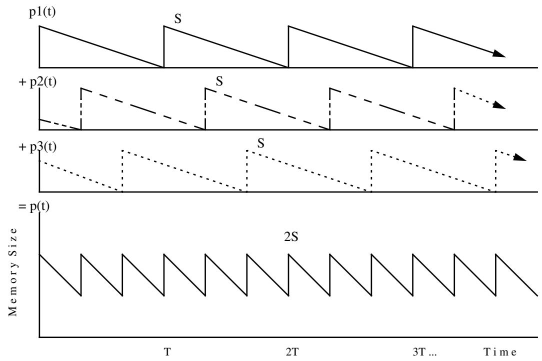
When segment b is read, we will have only S bytes in memory, which is only enough to sustain us for T seconds. Fortunately, because b was at the end of its sweep, the next segment c can be at most T seconds away, so we are safe. Actually, c could happen to be the very first segment read in sweep C, in which case we would again fill up the buffer with roughly $2 \times S$ data (minus whatever data was played back in the time it takes to do both reads).

Intuitively, what happens is that half of the $2 \times S$ buffer is being used as a *cushion* to handle variability of reads within a sweep. Before we actually start playing a stream we must ensure that this cushion is filled up. In our example, if this stream were just starting up, we could not start playback when a was read. We would have to wait until the end of sweep A (the sweep where first segment a was read) before playback could start. Then, no matter when b and c and the rest of the segments were read within their period, we could sustain the DR playback rate.

Adding a cushion buffer for each request doubles the memory required. For N_{Limit} requests, the total memory for Sweep is $2 \times N_{Limit} \times S$ (ignoring sharing of free space among buffers). Note that although Sweep has a smaller segment size, it needs a two-segment buffer for each request to cope with the IO time variability.



(a) Worst Memory Requirement



(b) Minimum Memory Requirement

Figure 6: Memory Requirement Examples

4.3 Memory Sharing

Memory can be shared among N_{Limit} requests. Figure 6 depicts three streams using the memory resource. When an IO occurs, the memory buffer of each stream is filled with S amount of data. It is then drained by the display device at rate DR in the next period T. This pattern repeats itself until the playback ends. The bottom curves show the total memory used, i.e., the sum of the top three curves.

Note that when the IOs are close to each other as shown in Figure 6(a), the peak memory requirement (bottom curve) is $3 \times S$. On the other hand, Figure 6(b) shows that when the IOs are spaced out with equal distance, the memory requirement is reduced to $2 \times S$. In general, a scheme like Sweep that minimizes seek latency has shorter gaps between IOs, and hence the memory sharing is limited. On the other hand, when IOs are delayed with equal gaps, like in the case of Fixed-Stretch, the memory sharing factor is optimal [3, 5, 12]. Thus, Fixed-Stretch has an important advantage over Sweep with respect to memory sharing.

In [3, 5] we derive precise formulas for the memory required by each scheme, [3] assuming sharing of free space among requests. Due to space limitations, here we simply report the expressions: The memory requirement for Fixed-Stretch is

$$Mem_{Min(Fixed-Stretch)} = \frac{S \times (N_{Limit} + 1)}{2}$$

$$+(N_{Limi} \times \gamma(CYL) \times DR). \qquad (5)$$

The memory requirement for Sweep with memory sharing is

$$Mem_{Min(Sweep)} = (N_{Limit} - 1) \times S$$

$$+N_{Limit} \times DR \times (T - \frac{(N_{Limit} - 2) \times S}{TR}). \qquad (6)$$

From this memory requirement one can compute the maximum throughput given some fixed amount of available memory. For example, one can assume that $N_{Limit} = 1$ and see if the computed $Mem_{Min(Sweep)}$ is less than the available memory. If so, we can try $N_{Limit} = 2$ and so on, until we find the largest feasible value of N_{Limit}. This last value would be the maximum throughput supportable.

[3] In computing the memory requirement, we did not consider the memory required by the page table that maps the contiguous virtual address to the physical memory pages. Since it is reasonable to assume a large page size for a media server, the memory overhead for storing the page table is probably negligible relative to the segment size.

Parameter Name	Value
Disk Capacity	*2.25 GBytes*
Number of cylinders, CYL	*5,288*
Min. Transfer Rate TR	*75 Mbps*
Max. Rotational Latency Time	*8.33 milliseconds*
Min. Seek Time	*0.9 milliseconds*
Max. Seek Time	*17.0 milliseconds*
$\alpha 1$	*0.6 milliseconds*
$\beta 1$	*0.3 milliseconds*
$\alpha 2$	*5.75 milliseconds*
$\beta 2$	*0.0021 milliseconds*

Table 5: Seagate Barracuda 4LP Disk Parameters

4.4 BubbleUp with Fast-Scan Operations

Assuming that up to M_{Limit} fast-scan operations are allowed in each service cycle, BubbleUp (and Fixed-Stretch) now has to read more data to sustain the playback for the duration of $N_{Limit}+M_{Limit}$ service slots, rather than N_{Limit}.

We now derive the memory requirement for supporting M_{Limit} fast-scan requests in a cycle T. If the server allows M_{Limit} fast-scan operations per service cycle, then the segment size, instead of being S, must be $S' = \frac{N_{Limit}+M_{Limit}}{N_{Limit}} \times S$. When a request arrives, $\frac{N_{Limit}+M_{Limit}}{N_{Limit}} \times S$ is read into its buffer. If no fast-scan operations are issued in the service cycle, only S amount of data is consumed, and hence only S amount is replenished in the subsequent cycle. If M_{Limit} fast-scans occur in the service cycle, then another $\frac{N_{Limit}+M_{Limit}}{N_{Limit}} \times S$ amount of data needs to be replenished in the next service cycle. In the worst case, all IOs in a cycle transfer $\frac{N_{Limit}+M_{Limit}}{N_{Limit}} \times S$ amount of data. Rewriting the continuity requirement in Equation 3, we get

$$S/DR = N_{Limit} \times (\gamma(CYL) + \frac{(N_{Limit} + M_{Limit}) \times S}{N_{Limit} \times TR}). \quad (7)$$

Solving for S we get

$$S = \frac{N_{Limit} \times \gamma(CYL) \times TR \times DR}{TR - (DR \times (N_{Limit} + M_{Limit}))}. \quad (8)$$

The segment size S' for supporting M_{Limit} fast-scan operations in a service cycle is

$$S' = \frac{(N_{Limit} + M_{Limit}) \times S}{N_{Limit}}. \quad (9)$$

This S' is to replace the S in Equation 5 to compute the total memory requirement to support N_{Limit} streams and M_{Limit} fast-scan operations.

5 Evaluation

In this section, we use a case study to compare the initial latency and throughput of BubbleUp (with Fixed-Stretch), Sweep and GSS. For our evaluation we use the Seagate Barracuda 4LP disk [1]; its parameters are listed in Table 5.

For computing the seek overhead we use the following function [10, 16]:

$$\gamma(d) = \alpha 1 + (\beta 1 \times \sqrt{d}) + 8.33 \text{ if } d < 400$$

$$\gamma(d) = \alpha 2 + (\beta 2 \times d) + 8.33 \text{ if } d \geq 400$$

In each seek overhead we have included a full disk rotational delay of 8.33 ms. The rotational delay depends on a number of factors, but we believe that one rotation is a representative value. One could argue that rotational delay could be eliminated entirely if a segment were an exact multiple of the track size. (In that case we could start reading at any position of the disk.) However, the optimal segment size depends on the scenario under consideration, so it is unlikely it will divide exactly into tracks. If we assume that the first track containing part of a segment is not full, then in the worst case we need a full rotation to read that first portion, even with an on-disk cache. If we assume that the last track could also be partially empty, then we would need a second rotational delay, and our 8.33ms value might be conservative! Note incidentally that we use a *full* rather than an average rotational delay since we are estimating a worst case scenario.

5.1 Initial Latency

For Sweep, the worst initial latency happens when a request arrives just after the disk head has passed over the first segment of the media. The request must wait for a cycle (T) until its first segment can be retrieved. As discussed in Section 4.2, playback cannot start right away, since this first segment just fills up the playback cushion. Actual playback can start at the end of the first cycle, which in the worst case can be another T seconds away. The worst initial latency is therefore $T_{Latency} = 2 \times T$. The latency expression for GSS is the same, although the actual values can be higher because GSS typically uses a larger segment size to improve throughput. As shown in Section 3, the latency expression for BubbleUp (with Fixed-Stretch) is $T_{Latency} = 2 \times \gamma(CYL) + S/TR$.

Note that in a particular service cycle T, schemes Sweep and GSS may have some slack time, and it may be possible for the disk arm to temporarily alter its sweep pattern to service a new request [18]. Thus, if there happens to be slack available, a particular new request could be started earlier (although we still have to fill the cushion buffer before playback starts). However, the amount of slack left in the cycle is stochastic, and hence cannot be relied on in computing the worst case initial latency.

We show in Figure 7 that BubbleUp improves the initial latency significantly compared to Sweep. While the initial latency of Sweep goes up superlinearly with N_{Limit}, BubbleUp maintains an almost constant initial latency which is under a quarter of a second.

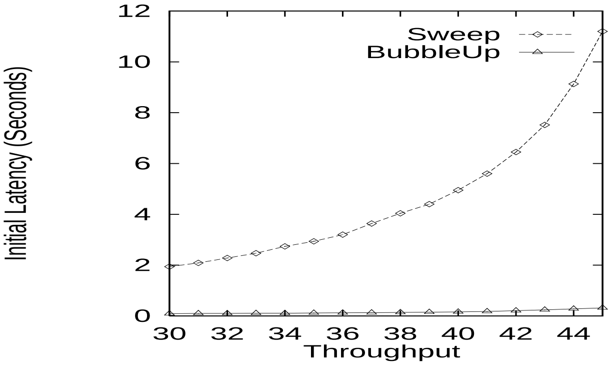


Figure 7: Initial Latency vs. Throughput

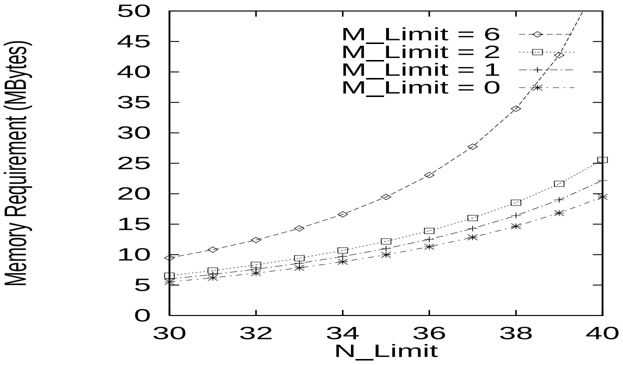


Figure 9: Memory Required for Fast-Scans

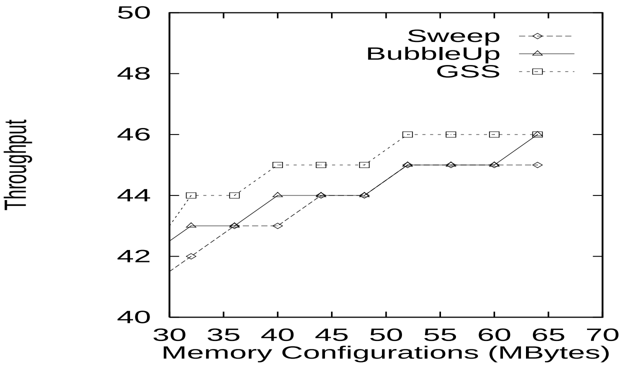


Figure 8: Throughput vs. Memory

5.2 Throughput

Next we compare the throughput of BubbleUp with Sweep and GSS. We plot the throughput figure by first computing the memory requirement using Equation 5 for BubbleUp and Equation 6 for Sweep (for GSS please refer to [5]), then picking the largest N_{Limit} each scheme can achieve under a given memory configuration. Figure 8 shows BubbleUp consistently supports up to one more stream than Sweep does when memory is limited. Both schemes support the same number of streams when memory is abundant. Compared to the optimal throughput GSS can achieve, BubbleUp trails by at most one stream.

These results show that minimizing disk latency alone (like Sweep does) does not necessarily lead to better throughput. A scheme that conserves memory like BubbleUp (with Fixed-Stretch) by fixing the scheduling order and improving memory sharing achieves comparable, sometimes better, throughput. Furthermore, as we have shown, BubbleUp minimizes initial latency (with essentially no throughput penalty), and this gives it a big edge over the other schemes. Thus, for interactive applications that require fast response time, BubbleUp may be the choice.

5.3 The Cost for Supporting Fast-Scan

Figure 9 shows the memory requirement for supporting 1, 2, and 6 fast-scan operations on top of the N_{Limit} requests. The horizontal axis represents N_{Limit}, while the vertical axis shows the total memory needed to support that number of requests plus the M_{Limit} fast-scans. The marginal memory requirement goes up as throughput (N_{Limit}) or the number of fast-scan operations (M_{Limit}) increases. For instance, supporting one fast-scan ($M_{Limit} = 1$ instead of zero) at $N_{Limit} = 40$ requires 2.7 MBytes of additional memory. By comparison, the incremental memory requirement at $N_{Limit} = 35$ for one fast-scan (compared to none) is 1.0 MBytes. Supporting a second fast-scan operation at $N_{Limit} = 40$ needs 3.4 MBytes of additional memory. This is not a surprise since Equation 8 shows that as $N_{Limit} + M_{Limit}$ grows the denominator of Equation 8 approaches zero, causing S to grow super-linearly.

Note that it is cheaper to support fast-scans (i.e., to increase M_{Limit}) than to support the same number of additional streams (i.e., to increase N_{Limit}). For instance, Figure 9 shows that supporting $N_{Limit} = 35$ plus $M_{Limit} = 6$ requires 19.5 MBytes of memory (the first line from the top); however, supporting $N_{Limit} = 40$ (the bottom line at $N_{Limit} = 40$) also needs the same amount of memory. In this case, a reduction in throughput of five requests supports at least six fast-scan operations (M_{Limit} becomes the lower bound if we make the enhancement discussed in the end of Section 3.4). This illustrates that supporting fast-scan operations incurs only a sub-linear tradeoff with throughput. Which of the two operations to favor in a media server, new requests (which increases throughput) or fast-scan, is a policy decision.

In closing this section we make one important point. The results of this section are for a specific hardware scenario. However, we believe that our general conclusions hold even under different disk parameters. Reference [4] presents results that support this claim, but due to space limitations they cannot be given here.

6 Conclusion

In this paper we proposed a disk and memory management scheme, BubbleUp, that minimizes initial latency for servicing new and fast-scan requests. To minimize initial latency, BubbleUp, always makes the available disk bandwidth and memory resource ready for servicing a new request. In addition, BubbleUp can prefetch additional data for each request so that when a fast-scan request arrives, it can delay service to the scheduled requests in order to service the fast-scan promptly. We showed that BubbleUp typically keeps the initial latency under a quarter of a second even when the media server is heavily loaded. We believe that this low latency is critical in interactive multimedia applications.

To implement BubbleUp, the disk arm must be given freedom to move to any location on the disk so that the scheduling order of the requests can be changed. This freedom is made possible by our underlying Fixed-Stretch policy. Intuitively, increasing disk latency between IOs may lead to throughput degradation. However, we showed that spacing out IOs conserves memory, and BubbleUp (without fast-scan support) is able to achieve throughput near the optimal of the GSS scheme, and outperform a conventional elevator disk arm scheduling policy like Sweep. In short, BubbleUp minimizes initial latency without adversely affecting throughput.

It is important to note that BubbleUp can be used with any commercial, off-the-shelf disks since it does not require any modification to the device drivers. The entire implementation of BubbleUp can be above the device driver layer, since BubbleUp only needs to queue an IO request after the completion of the previous one. Another benefit of BubbleUp is that even when the system is fully loaded, the disk arm usually has slack between IOs. This slack could be used to service short duration conventional file IOs, such as for accessing closed captions for the playing videos. (To efficiently use the slack, however, may require implementing BubbleUp at the device driver level.) Thus, BubbleUp may be a good resource manager for multimedia file systems. We also believe that BubbleUp can be used in a multi-disk environment, and we are currently studying its implementation and performance in such a case.

References

[1] Seagate barracuda 4lp family product specification. *URL: http://www.seagate.com*, 1996.

[2] S. Berson, S. Ghandeharizadeh, R. Muntz, and X. Ju. Staggered striping in multimedia information systems. *Proceedings of the ACM Sigmod*, pages 79–89, 1994.

[3] E. Chang and Y.-Y. Chen. Minimizing memory requirements in a multimedia storage system. *Stanford Technical Report SIDL-WP-1996-0045 URL: http://www-diglib.stanford.edu*, 1996.

[4] E. Chang and H. Garcia-Molina. Effective memory use in a media server (extended version). *Stanford Technical Report SIDL-WP-1996-0050 URL: http://www-diglib.stanford.edu*, 1996.

[5] E. Chang and H. Garcia-Molina. Effective memory use in a media server. *Proceedings of the 23rd VLDB Conference*, August 1997.

[6] E. Chang and H. Garcia-Molina. Reducing initial latency in media servers. *IEEE Multimedia*, Fall 1997.

[7] T. Chua, J. Li, B. Ooi, and K.-L. Tan. Disk striping strategies for large video-on-demand servers. *ACM Multimedia*, pages 297–306, November 1996.

[8] S. Ghandeharizadeh, S. Kim, and C. Shahabi. On configuring a single disk continuous media server. *Sigmetrics Performance Evaluation*, 23(1):37–46, May 1995.

[9] S. Ghandeharizadeh, S. Kim, and C. Shahabi. On disk scheduling and data placement for video servers. *USC Technical Report USC-CS-TR97-650*, December 1995.

[10] D. Kotz, S. B. Toh, and S. Radhakrishnan. A detailed simulation model of the hp 97560 disk drive. *Dartmouth College Technical Report PCS-TR94-220*, 1994.

[11] D. Makaroff and R. Ng. Schemes for implementing buffer sharing in continuous-media systems. *Information Systems*, 20(6):445–464, 1995.

[12] R. Ng and J. Yang. Maximizing buffer and disk utilizations for news on-demand. *Proceedings of the 20th VLDB Conference*, pages 451–462, 1994.

[13] B. Ozden, A. Biliris, R. Rastogi, and A. Silberschatz. A low-cost storage server for movie on demand databases. *Proc. VLDB*, September 1994.

[14] B. Ozden, R. Rastogi, and A. Silberschaz. *The Storage and Retrieval of Continuous Media Data, a Chapter in Multimedia Database Systems*. Springer, 1996.

[15] A. Reddy and J. Wyllie. I/o issues in a multimedia system. *Computer*, 2:69–74, March 1994.

[16] C. Ruemmler and J. Wilkes. An intro to disk drive modeling. *Computer*, 2:17–28, March 1994.

[17] R. Steinmetz. Multimedia file systems survey: approaches for continuous media disk scheduling. *Computer Communications*, pages 133–44, March 1995.

[18] S. R. Thuel and J. P. Lehoczky. Algorithms for scheduling hard aperiodic tasks in fixed-priority systems using slack stealing. *Proceedings of Real-Time Systems Symposium*, pages 22–33, December 1994.

[19] F. Tobagi, J. Pang, R. Baird, and M. Gang. Streaming raid-a disk array management system for video files. *First ACM Conference on Multimedia*, August 1993.

[20] P. Yu, M.-S. Chen, and D. Kandlur. Grouped sweeping scheduling for DASD-based multimedia storage management. *Multimedia Systems*, 1(1):99–109, January 1993.