2D BubbleUp: Managing Parallel Disks for Media Servers

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

Abstract

In this study we present a scheme called two-dimensional BubbleUp (2DB) for managing parallel disks in a multimedia server. Its goal is to reduce initial latency for interactive multimedia applications, while balancing disk loads to maintain high throughput. The 2DB scheme consists of a data placement and a request scheduling policy. The data placement policy replicates frequently accessed data and places them cyclically throughout the disks. The request scheduling policy attempts to maintain free "service slots" in the immediate future. These slots can then be used to quickly service newly arrived requests. Through examples and simulation, we show that our scheme significantly reduces initial latency and maintains throughput comparable to that of the traditional schemes.
Keywords: multimedia, data replication, initial latency, disk array.

1 Introduction

Media servers are designed to provide large numbers of presentations in the form of audios, movies or news clips. These servers need a large number of disks, not only for storing the data, but also for providing the required high bandwidth for all simultaneous *streams*. In this paper we propose a scheme called two-dimensional BubbleUp (2DB) that manages parallel disks for large media servers.

The objective of 2DB is to minimize initial latency while maintaining high throughput. We define initial latency as the time between the request's arrival and the time when the data is available in the server's main memory. Low initial latency is important for interactive multimedia applications such as video games, since we do not want the user to wait for a long time at scene transitions. Even in movie-on-demand applications, where a few minutes' delay before a new multi-hour movie starts may be acceptable, the response time should be very short when the viewer decides to fast-scan (e.g., fast-forward or rewind) to other segments of the movie. Our 2DB scheme minimizes the initial latency for both newly arrived and fast-scan requests of an ongoing presentation.

Many schemes have been proposed in the literature to manage parallel disks for media servers (see Section 1.1). Most of these schemes try to balance disk load to maximize throughput. However, this maximum throughput is often achieved at the expense of long initial latencies. In particular, a new request is not admitted until it can be *guaranteed* that no one disk will be overloaded in the future. This rigid scheduling causes requests to be delayed for long periods of time, even if the disks containing the initial segments for the new presentation have the capacity to serve the new request. When the server is near its peak load, the initial latencies can be on the order of $O(MN)$, where M is the number of disks and N is the number of IOs performed in one sweep of the disk arm. Data replication can be used to reduce initial latencies, but still, they can be on the order of $O(N)$. This is because these schemes typically use an elevator-like disk scheduling policy: When 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 (servicing up to $N - 1$ requests) and returns to the data segment in the next sweep.

The 2DB scheme services requests in *cycles* (time dimension) on parallel disks (disk dimension). Each cycle is divided into *service slots* of equal duration. An ongoing presentation performs one IO per cycle, using up one of the slots in that cycle on one of the disks. The 2DB *scheduling policy* assigns IO requests onto these two dimensional disk-time slots. It attempts to maintain the maximum number of free slots open in the near future to service new requests. If new requests arrive before the start of a service slot, they are usually scheduled immediately in the free slots, with little delay. If free slots are still available after new requests have been satisfied, the scheduler assigns existing requests that have the closest service deadlines to use the remaining slots. This effectively pushes the deadlines of the most urgent tasks further in time, freeing up service slots in the immediate future, as well as reducing the number of requests that will compete for the free slots with future new requests. We use the name "bubble-up" because free slots bubble up to or near the current time. The simulation results of Section 6.2 show that the 2DB scheme can service new requests in under 0.5 second on the average, even when the requests arrive in a large batch and when the server is near its peak load. Other schemes can take from 5 seconds to several minutes.

A *data placement* policy is used by 2DB to balance loads across disks. Each presentation is split into *chunks* and spread across the disks. Like other traditional schemes, 2DB also replicates popular presentations. However, the replication and data distribution is performed in a way that enhances the run-time bubbling-up of free slots. The results of Section 6.1 show that two copies of the popular presentations are sufficient to balance loads and permit the bubble-up strategy to effectively reduce latencies.

Scheme 2DB can have service disruptions (hiccups), when some the data required by some presentations can only be found at fully loaded disks. However, the probability of a hiccups can be reduced to an insignificant amount by slightly reducing the maximum system throughput. In other words, suppose that our server can support N_{all} streams with a traditional multi-disk scheme. If we reduce this maximum load to $N_{all} - M \times N_b$, for $N_b = 2$ (M is the number of disks), then the hiccups virtually disappears. Typically, this is a reduction of 2 to 6% in overall throughput. This is the price one pays to achieve the very low initial latencies; we believe this price will be acceptable in many interactive applications.

The rest of the paper is organized as follows. Section 2 introduces the parameters used in this study. In Section 3 we review the single disk scheduling policy, Fixed-Stretch, used by 2DB. Sections 4 and 5 present the 2DB scheme through an example and formal specification, respectively. Section 6 describes our simulation results and observations. Finally, we offer our conclusions in Section 7.

1.1 Related Work

The 2DB scheme is an extension of scheme *BubbleUp*, which we proposed in [8] for managing a single disk. We call the new scheme 2DB because the free disk bandwidth can be bubbled up in the schedule not only in the time dimension but also in the disk dimension. (This will become evident from the example we show in Section 4). Adding the disk dimension makes the problem much more challenging since the scheme must minimize initial latency, as well as maintain balanced disks.

Parallel disk management schemes have been widely studied for conventional file and database systems [4, 19]. However, the design of a parallel disk manager for a media server faces at least two additional challenges:

1. The data delivery must meet real-time constraints, and
2. Data prefetching cannot be excessive, since media data is voluminous and may take up too much buffer space, driving down throughput.

Many studies have addressed disk arrays, where IOs have real-time constraints. The study of [6] proposed a staggered striping scheme for media data that requires higher bandwidth than the transfer rate of a disk. The focus of other studies and this one is to devise data placement and scheduling schemes for media data that requires lower bandwidth than the transfer rate of a disk. The proposed schemes for this area can be categorized by their disk striping technique, as either *fine-grained* or *coarse-grained* [27, 28]. Fine-grained striping works as follows: Fine-grained disk striping treats M disks as one storage unit, with each IO unit (which we call *segment*) broken into M subunits (*subsegments*), each stored on a separate disk [5, 32, 35]. With fine-grained striping, all M disks service one request at a time. Studies [9, 27] have shown that this fine-grained approach may not be desirable due to its almost linear growth of required memory with respect to number of disks. Initial latencies are also very high, on the order of $O(MN)$, where N is the number of users serviced per disk.

Coarse-grained striping stores each segment on one disk only. When a segment is accessed, only one of the M disks is involved in that IO. Coarse-grained striping is attractive because of its lower memory cost. A variety of coarse-grained striping schemes have been proposed. The simplest approach stores each presentation entirely on one disk. (A disk may store more than one presentation.) A drawback of this scheme is that it can lead to an unbalanced workload, where disks with "hot" movies become overloaded. This phenomenon is sometimes referred to as *bandwidth fragmentation*: the idle disk bandwidth cannot be used to service a new request because the requested presentation is not at an idle disk. To minimize bandwidth fragmentation, [14, 26, 12, 27, 32] propose placing segments of a presentation on M disks in a round robin fashion. In this way, both cold and hot movies share the bandwidth of all disks. However, since these schemes admit a new request only until the new request does not overload any disk, the initial latency may be very long (on the order of $O(MN)$).

To reduce initial latency, data replication schemes have been proposed [20, 33]. Replication does help contain initial latency. However, these schemes schedule requests one cycle (T) at a time (due to the limitation of the elevator disk scheduling policy), so latencies are still high, on the order of $O(N)$. Our 2DB scheme achieves significantly lower initial latency.

Due to space limitations we cannot discuss the schemes mentioned in this section in more detail. However, Appendix A provides some additional discussion and examples.

2 Analytical Model

To analyze the performance of a media server, we are typically given the following parameters regarding the hardware (i.e., memory and disks), the videos, and the requests:

- Mem_{avail}: Available memory, in MBytes.
- M: Number of disks. In this study, we assume the media server uses homogeneous disks.
- TR: The disk's data transfer rate.
- $\gamma(d)$: A concave function that computes the disk latency given a seek distance d. For convenience, we refer to the combined seek and rotational overhead as the disk latency.
- L: Number of distinct video titles.
- L': Number of videos stored on disks after data replication.
- DR: The display rate of the videos.

The media server has the following tunable parameters, which can be adjusted within certain bounds to optimize system throughput:

- T: The period for servicing a round of requests on each disk. T must be made large enough to accommodate the maximum number of streams we expect to handle.
- S: The segment size, i.e., the number of bytes read for a stream with a contiguous disk IO.
- N: The maximum number of requests a disk allows in T.
- N_b: The cutback in throughput to reduce hiccups. We also call N_b the number of cushion slots in T.
- N_{all}: The total number of requests the server allows. $N_{all} = (N - N_b) \times M$.

To assist the reader, Table 1 summarizes these parameters, together with other parameters that will be introduced later. The first portion of Table 1 lists the basic fixed and tunable parameters. The second portion describes subscripted parameters that are used for the characteristics of individual requests.

3 Fixed-Stretch

Before presenting the 2DB scheme, this section briefly reviews a disk scheduling policy *Fixed-Stretch*, which we presented in detail in [8]. Our proposed scheme, 2DB, presented in the next section, uses a modified version of *Fixed-Stretch*.

To simplify our discussion, we assume in this study that each stream is encoded in a constant bitrate (CBR) and all streams have the same display rate, DR. At the end of this section we present some techniques that can be used to extend Fixed-Stretch (and hence 2DB) to support streams encoded in a variable bitrate (VBR) (i.e., the stream bitrate changes with time) and streams with different display rates (i.e., the bitrates differ from one stream to another).

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

Parameter	Description
Mem_{Avail}	Total available memory, MBytes
M	Number of disks
DR	Data display rate, Mbps
TR	Disk transfer rate, Mbps
CYL	Number of cylinders on disk
$\gamma(d)$	Function computes seek overhead
T	Service time for a round of N requests
Δ	Duration of a service slot
S	Segment size, MBytes
CK	Chunk size, number of segments
CO	Chunk overlap size, number of segments
C	Copies of the popular movies
L	Number of movies
L'	Number of movies after replication
N	Limit on number of requests in T
N_b	Throughput cutback or the number of cushion slots per T
N_{all}	Limit on number of requests in the server per T
R_i	i^{th} request
D_i	i^{th} disk

Table 1: Parameters

segment is read.

In a feasible system, the period T must be large enough so that even in the worst case all *scheduled* IOs can be performed. Thus, we must make T large enough to accommodate N seeks and transfer N segments. Fixed-Stretch achieves this by dividing a service cycle T into N equal service slots. Since the data on disk needed by the requests are not necessarily separated by equal distance, we must add time delays between IOs to make all service slots last 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 cylinders, and cyl_i is the maximum of these, then we must separate each IO by at least the time it takes to seek 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 times Δ. Figure 1 presents an example where $N = 3$. 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), it sets the *end of slot*

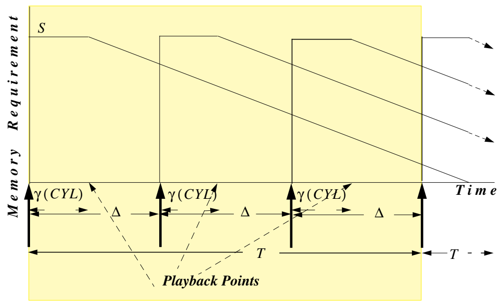


Figure 1: Service Slots of Fixed-Stretch

timer to expire in Δ.

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

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

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

5. When the *IO timer* expires, the playback starts consuming the data in the buffer (indicated by the "playback points" arrows 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 been completed.[2] Fixed-Stretch goes 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 of whether other requests depart or join the system. (As we will see in Section 4, the "fixed" scheduling may be changed by the 2DB scheme.)
- 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.

At first glance, Fixed-Stretch appears to be inefficient since it assumes the worst seek overhead between IOs. However, it uses memory very efficiently because of its very regular IO pattern, and this compensates for the poor seek overhead. In [8, 9] we analyze the memory requirement of Fixed-Stretch and compare its performance with the performance of other disk scheduling policies (e.g., elevator and GSS [34]). We show that Fixed-Stretch achieves throughput comparable to that of the other schemes.

[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 possibility 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 possibility 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 performance of a larger IO. Other possibilities are discussed in [25].

[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.

Note that although Fixed-Stretch guarantees that in each service cycle a disk can service up to N requests, Fixed-Stretch is not guaranteed to be hiccup-free. Suppose that a server has M disks, each using Fixed-Stretch to service up to N requests in a service cycle. Suppose further that the total number of requests in service never exceeds $N \times M$. Server hiccups may occur if in one service cycle more than N requests wish to access data on one disk. In Section 4 we propose techniques to balance disk load so that the hiccups can be virtually eliminated.

3.1 Supporting Different Display Rates and VBR

A simple-minded solution to support VBR encoded streams and streams with different display rates is to use the maximum rate of all streams as the universal display rate. This approach wastes memory and is clearly inefficient. In this section we briefly discuss more efficient techniques.

To support streams of different display rates, one alternative is to store media data that have the similar display rates on the same disk array. This does not require any changes to the Fixed-Stretch (and hence 2DB) scheme. We can also use any combination of the following techniques to minimize the display rate disparity between streams:

1. Spliting a large stream: For a stream that has a display rate significantly larger than the others, we can split the stream and service it in multiple service slots. For example, if a stream needs a 2.5 times rate, we allocate three service slots for the stream. A split stream does not have to be serviced in consecutive slots, as long as the data buffer does not underflow.
2. Grouping small streams: For streams that have display rates significantly smaller than the others, we can bundle these low-rate streams into some groups and service each group in one service slot. For instance, the playback rates of a movie's audio track and closed-caption may be significantly lower than the playback rate of the video. These low bit-rate IOs can be serviced in one sweep of the disk arm in one service slot. The only restriction is that the IOs must be completed in a service slot. Let k denote the number of IOs serviced in a sweep and S_i the segment size of the i^{th} IO in the service slot, where $i = 1$ to k. The duration of the service slot, Δ, must be long enough to cover the worst-case [3] total seek overhead $k \times \gamma(CYL/k)$ and the total data transfer time, or

$$k \times \gamma(CYL/k) + \sum_{i=1}^{k} S_i/TR \leq \Delta.$$

3. Smoothing out rate differences using data placement techniques: When the differences between DRs are not large, one can store the streams with higher DRs on the outer tracks of the disks and the ones with lower DRs on the inner tracks. Since outer tracks enjoy higher transfer rates in modern disks, the transfer time for all requests, i.e., DR_i/TR_i (DR_i denotes the display rate of the i^{th} request and TR_i the data transfer rate), can be normalized into a narrower range.

To support VBR, Fixed-Stretch can reserve extra service slots at the end of each service cycle for streams that may consume more bits in the next service cycle than their regular cycles. We believe that the number of reserved slots needed for supporting VBR is moderate because of the following reasons:

[3]Many studies [21, 30, 31] have shown that the function of computing seek overhead is concave, so the worst total seek overhead occurs when the seeks equally divide the radius of the disk.

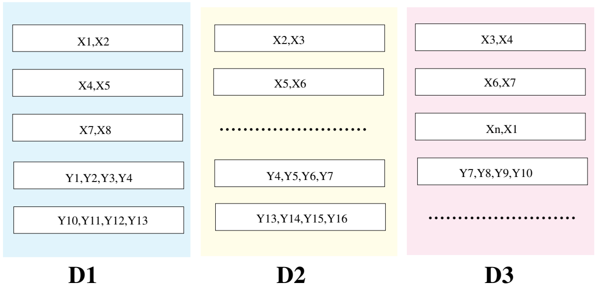


Figure 2: Data Placement Example

1. Not all streams require peak display rate in the same service cycle.
2. The bitrate fluctuation is smoothed out substantially by the large segment size. If we look at the bitrate at the frame level, since the number of bits required to encode anchor frames (e.g., I and P frames) can be significantly more than that required by the frames in between (e.g., B frames), the bitrates may seem to change a lot from frame to frame. However, the IO size is a segment rather than a frame. Since a segment typically contains more than one hundred frames according to our experiments, the segment hides much of the variability. Consequently, at the level we care about, the IO size does not fluctuate as significantly as one may think.

Supporting different display rates and VBR increases memory requirement to cushion the rate variability for all multimedia scheduling policies; Fixed-Stretch does not suffer disproportionately compared to other schemes. On the other hand, supporting different display rates and VBR does not affect initial latency: the focus of this paper. Therefore we deem the display rate and bitrate as orthogonal issues to the focus of this study and use the same display rate and CBR in the remainder of this paper to simplify our discussion. For studies on the display rate and bitrate issues, and other proposed solutions, please consult references [11, 22, 24].

4 2-Dimensional BubbleUp (2DB)

In this section we use an example to illustrate how our scheme, two-dimensional BubbleUp (2DB), works. In the example, we assume that the server uses three disks, D_1, D_2, and D_3, each able to service up to four requests ($N = 4$) in each service period T. We assume two movies, X and Y, are placed on the three disks. We also assume that movie X enjoys higher viewing popularity than movie Y does and hence we place two copies of movie X on the disks. In the remainder of this section we show how scheme 2DB places data and schedules requests. To simplify our discussion, we only show how scheme 2DB minimizes initial latency for the newly arrived requests. Section 6.5 discusses how to extend the scheme to support low latency fast-scan operations.

4.1 Data Placement

Scheme 2DB places data on disks in chunks. The chunk placement follows three rules:

1. Each chunk is physically contiguous and is a minimum of two segments in size.

2. The tailing segments of a chunk are always replicated at the beginning of the next chunk. We call the replication size *chunk overlap*; chunk overlap is a minimum of one segment in size.

3. The first chunk of a movie is placed on a randomly selected disk, and the subsequent chunks are placed in a round-robin fashion throughout the disks.

Figure 2 shows an example of chunk placement. The formal rules of chunk placement are specified in Section 5.1. In the figure, the chunk size and chunk overlap size of movie X are two and one and of movie Y four and one, respectively. Note that we can compute the number of copies of a movie on the disks using the formula $\frac{chunk\ size}{chunk\ size\ -\ chunk\ overlap\ size}$. For instance, movie X (the more popular movie) has $\frac{2}{2-1} = 2$ copies on the disks while movie Y has $1\frac{1}{3}$.

The chunk placement is intended to accomplish the following objectives:

- Minimizing IO overhead: Placing data in chunks ensures that every disk IO less than S in size is physically contiguous (performing only one seek). (We explain in Section 4.3 that scheme 2DB sometimes needs to retrieve a fraction of a segment to conserve memory.)
- Improving scheduling flexibility: The more copies of a movie reside on the disks, the higher the probability that the server can find a disk to schedule the requests for that movie.
- Balancing workload among disks: Placing the first chunks of the movies on randomly selected disks makes it highly probable that the requests are uniformly distributed on the disks [2, 4]. (We discuss the details in Section 6.1.)

4.2 Request Scheduling

To service four requests in T, scheme 2DB uses the disk scheduling policy *Fixed-Stretch*, which divides the period into four equally separated service slots, each lasting time Δ ($T = 4 \times \Delta$). Policy Fixed-Stretch is chosen because its assumption of the worst-case seek overhead between IOs gives the disk arm the freedom to move to any disk cylinder to service any request promptly. Scheme 2DB schedules requests for one Δ at a time. At the start of each Δ, it assigns one request to each disk to retrieve up to S amount of data. This not only minimizes the number of seeks (recall that as long as the data transfer size is $\leq S$, the number of seeks is one), but also keeps the memory requirement under control. For M disks (in our example $M = 3$), scheme 2DB schedules up to M IOs, each on one disk, at the start of each Δ.

The 2DB scheme assigns requests to disks according to their priorities. The priorities are ranked based on the requests' service deadlines, i.e., the earlier the deadline, the higher the priority. For instance, a request that will run out of data in 2Δs enjoys a higher scheduling priority than one that will run out of data in 3Δs. Scheme 2DB assigns requests, starting from the highest priority ones, to M disks until either all disks are used or no more requests can be assigned. Note that not all disks may be used for two reasons: (1) the number of requests in the server is less than M, or (2) the unassigned disks do not have the data needed by the remaining requests.

To minimize initial latency, the 2DB scheme gives newly arrived requests the highest priority and attempts to assign them to disks immediately after their arrival. This, however, may cause a scheduling conflict on the disks to which the newly arrived requests are assigned. For example, suppose disk D_1 in Figure 2 is saturated and a newly arrived request wants to start movie Y on the disk (segment Y_1 resides

Arrive Time	*Request*	*Movie*	*Request*	*Movie*
Before Δ_1	R_1	X	R_2	Y
Before Δ_2	R_3	X	R_4	Y
Before Δ_3	R_5	X	R_6	Y
Before Δ_4	R_7	X		
Before Δ_5	R_8	X		
Before Δ_6	R_9	X		

Table 2: The Arrival Time of the Requests

Deadlines (Δs Away)	0	Δ	$2\Delta s$	$3\Delta s$
D_1			R_2	R_4
D_2				R_1
D_3				R_3

Table 3: 2D BubbleUp Example - At the End of Δ_2

on disk D_1). Assigning the newly arrived request to disk D_1 can cause the existing requests scheduled on disk D_1 to be "bumped" by one Δ. Any bump in the schedule disrupts the continuous data supply to the request and causes display disruption (hiccups). To reduce hiccups, scheme 2DB cuts back the throughput on each disk by N_b. These N_b slots work like a cushion to absorb the unexpected bumps. We show how N_b can be set to minimize (or virtually eliminate) the hiccups in Section 6.1.

4.3 Execution Example

This section shows an execution example. We assume that $N = 4$ and $N_b = 1$. Under this condition, each segment, S, sustains playback for $T = N \times \Delta = 4 \times \Delta$, although each disk services only $N - N_b = 3$ requests per cycle T. Table 2 lists the arrival time of nine requests, R_1 to R_9, and their requested movies. In the following we describe how the disks are assigned at the start of the first seven time slots. Each slot *instance* is labeled Δ_i to remind us that its duration is Δ time units. We use Ψ to denote the schedule that contains the disk assignment (request and disk pairs).

- Δ_1: Requests R_1 and R_2 have arrived requesting movies X and Y, respectively. The only possible schedule for the requests is $\Psi = \{\{D_1, R_2\}, \{D_3, R_1\}\}$.
  Note that without replicating segment X_1 on disk D_3, one of the requests cannot be serviced immediately.
- Δ_2: Requests R_3 and R_4 arrive requesting movies X and Y, respectively. Since the new requests enjoy the highest scheduling priority, we schedule requests R_3 and R_4 immediately. The only possible assignment for the new requests is $\Psi = \{\{D_3, R_3\}, \{D_1, R_4\}\}$. The idle disk D_2 can service R_1 to retrieve segment X_2. The amount of data retrieved for R_1 is $S/4$ since only that amount of data has been consumed since R_1's last IO in Δ_1. Keeping the peak memory required by each request under S caps the server's memory requirement. The schedule for Δ_2 is $\Psi = \{\{D_1, R_4\}, \{D_2, R_1\}, \{D_3, R_3\}\}$. If D_2 were not used to service R_1, the disk would be idle. Using the idle bandwidth to service R_1 in this time slot pushes back the deadline of R_1 by one Δ and hence gives the server more flexibility to schedule the newly arrived request in Δ_5. In other words, the disk bandwidth that was supposed to be allocated in Δ_5 is now freed to service other requests. Essentially, the free disk bandwidth is "bubbled

up" nearer in time and the number of the high priority requests in the future is also reduced.
Table 3 depicts the states of the requests at the end of Δ_2. The rows of the table are the disks and the columns the requests' deadlines, zero, one Δ, two Δs, and three Δs away. Requests R_1, R_3, and R_4, which were just serviced, have a service deadline that is three time slots away at the end of Δ_2. The deadline of R_2 is two Δs away. Note that the empty service slots are all kept near in time (zero, Δ, and 2Δs away). The deadlines of the requests are pushed back in the table as far as possible.

- Δ_3: Requests R_5 and R_6 arrive requesting movies X and Y, respectively. We schedule the newly arrived requests immediately. We assign requests R_5 and R_6 to disks D_3 and D_1, respectively. The idle disk D_2 can service either R_1 or R_3. We assign R_3 to D_2. The schedule for Δ_3 is $\Psi = \{\{D_1, R_6\}, \{D_2, R_3\}, \{D_3, R_5\}\}$. Again, pushing the deadline of R_3 backwards opens up free server bandwidth earlier in time for scheduling other requests.
- Δ_4 to Δ_6: Since the execution steps are similar, we skip the detailed description for these time slots. Table 4 summarizes the deadlines of the requests at the end of Δ_6.

Deadlines (Δs Away)	0	Δ	$2\Delta s$	$3\Delta s$
D_1		R_2	R_4	R_6
D_2		R_1	R_5	R_3
D_3		R_7	R_8	R_9

Table 4: 2D BubbleUp Example - At the End of Δ_6

At the end of Δ_6, the server is fully occupied. Any newly arrived requests will be either turned away or put in a queue until a request leaves the server (e.g., when the playback ends). Again, all empty slots are next in time because of the bubbleup policy.

- Δ_7: In this time slot, we show the use of the cushion slot ($N_b = 1$). We first schedule R_1, R_2, and R_7, the highest priority requests (their deadlines are nearest in time). Since the data that R_2 needs ($1/4$ of segment Y_2 and $1/2$ of Y_3) resides only on disk D_1, we must assign D_1 to R_2. The data that R_1 and R_7 need can only be found on disk D_2 (D_1 has been taken). Thus, we must bump either R_1 or R_7 by one Δ. If the cushion slot were not allocated, one of the requests would suffer a hiccup. Suppose we decide to bump R_1 and assign R_8 to the final disk. Table 5 shows the deadlines of the requests at the end of Δ_7. Note that the bumped request R_1 enjoys the highest scheduling priority in the next time slot and is guaranteed to be serviced in Δ_8. Our simulation results using significantly larger Ms and Ns (discussed in Section 6) show that by replicating popular movies and carefully placing the data chunks, a top priority request will not be bumped more than twice. Therefore, allocating two cushion slots ($N_b = 2$) is sufficient to virtually eliminate hiccups.

Deadlines (Δs Away)	0	Δ	$2\Delta s$	$3\Delta s$
D_1		R_4	R_6	R_2
D_2	R_1	R_5	R_3	R_7
D_3			R_9	R_8

Table 5: 2D BubbleUp Example - At the End of Δ_7

To summarize, the example first illustrates that the data replication (e.g., shown in Δ_1, Δ_2, and Δ_3) and the reduction in throughput (shown in Δ_7) help balancing disk load. We also observe that limiting the data prefetching size to S conserves memory. Furthermore, the bubbleup scheduling policy maintains the open service slots in the nearest future to minimize initial latency. In Section 6 a realistic simulation shows that all these observations hold and scale well with large Ns and Ms.

5 2DB Specification

Given L video titles, we first replicate the videos that enjoy high popularity. Video popularity is commonly modeled using Zipf or geometric distribution [3, 23, 33]. For simplicity, one can say that about 10% to 20% of the presentations enjoy 90% of the popularity. Thus, by replicating only 20% of the videos, we can get $L' = 1.2 \times L$ videos stored on the disks. [4]

Based on the storage requirement and the number of users that the server must support (N_{all}), one can derive the number of disks (M) and amount of memory (M_{avail}) required. In this section we assume that M and M_{avail} are given and we describe how scheme 2DB places data on the disks and schedules requests.

5.1 Data Placement

Let CK and β denote the starting address of and offset into the memory where the temporary chunk is created before being written onto the disk. Let κ denote the offset into the logical sequence of the video file V. Let CS denote the chunk size and CO the chunk overlap size (both in number of segments) of the video. Let $U(1, M)$ denote a random number generator that generates an integer uniformly distributed over the interval (1,M). Figure 3 depicts the procedure *ChunkPlacement* that implements the chunk placement policy. The procedure is as follows:

ChunkPlacement takes a video file (V), chunk size (CS), chunk overlap size (CO), segment size (S), and number of disks (M) as input and places V on M disks in chunks. The procedure first checks if the input observes two rules:

1. $CS > CO$: By definition, the chunk overlap size should be smaller than the chunk size, so the procedure can terminate.
2. $CS/(CS - CO) \leq M$: The copies of a movie must not exceed the number of disks, M, or the disk storage is wasted without any benefit being gained.

After the parameters pass the input validation in step 1, step 2 of the procedure initializes variables. Variable θ is assigned a random disk on which to place the first chunk of the video. Step 3 iterates through seven substeps the create and placement of chunks, one at a time, until the end of the video file is reached. Steps 3(a) and 3(b) initialize the offsets into the video file and the chunk's memory address, respectively. Step 3(c) creates the i^{th} chunk by copying CS number of segments from the video file to the chunk's memory address, one segment at a time. The *Copy* function copies the segment in the video file starting from the α offset into the local memory buffer CK starting from the β offset. If fewer than CS segments of data are left in the video file, the procedure wraps around the video file ($\alpha \leftarrow \alpha + S$ mod

[4] Deciding on the number of copies of each individual movie to be replicated by predicting the movie's access frequency is highly susceptible to prediction errors, especially the access frequency of a movie can change on an hour-by-hour or day-by-day basis [12, 33]. Rather than predicting the popularity of each individual movie, we deem that a group prediction model [13] less susceptible to prediction errors. Analysis of movie replication strategies, however, is beyond the scope of this study.

Procedure ChunkPlacement (in V, CS, CO, S, M)

- Local variables
  - CK /* Local memory buffer */
  - $i, j, \alpha, \beta, \kappa, \theta$

1. Validate Input Parameters:

   if ($CS \leq CO$) or ($\frac{CS}{CS-CO} > M$))

   Return error

2. Initialization:

   $\theta \leftarrow U(1, M)$

   $i \leftarrow 0$ /* i^{th} chunk, starting from zero */

   $\kappa \leftarrow 0$ /* offset into the video file */

3. While ($\kappa < sizeof(V)$)

   (a) $\alpha \leftarrow \kappa$ /* offset into V */

   (b) $\beta \leftarrow 0$ /* offset into the chunk */

   (c) For $j = 1$ to CS Do

   copy(α, V, β, CK, S)

   $\alpha \leftarrow \alpha + S$ mod $sizeof(V)$

   $\beta \leftarrow \beta + S$

   (d) $\theta \leftarrow (\theta + i)$ mod M

   (e) Place CK on disk D_θ

   (f) $\kappa \leftarrow \kappa + (CS - CO) \times S$

   (g) $i \leftarrow i + 1$

Figure 3: Procedure ChunkPlacement

$sizeof(V)$) and copies the initial segments into the chunk. For example, if the chunk size is two and the first segment of the chunk is the last segment of the video, the procedure appends the first segment of the video to its last segment to create a chunk. Step 3(d) decides the destination disk and step 3(e) places the chunk onto the chosen disk. Finally, steps 3(f) and 3(g) update the variables.

5.2 Formalizing the Request Scheduling Problem

At each service slot Δ, the server assigns at most M out of N_{all} requests to M disks. Since some videos are replicated on more than one disk, a request may be assigned to one of many disks. Figure 4 depicts this scheduling problem, where up to M of the N_{all} requests must be assigned to M disks. The lines connecting the requests on the left to the disks on the right are the possible matches between the requests and the disks. For instance, only one line emanating from requests R_1, R_5 and $R_{N_{all}}$ shows that the segments these requests access reside only on one disk. Two lines emanating from requests R_2, R_3, and R_4 shows that these requests can be assigned to either of the two disks at which the lines terminate.

We formalize the problem as follows: Let $G = (V, E)$ be a graph with vertex $V = R \cup D$, where R is the vertex set that contains up to N_{all} requests and D the vertex set that contains M disks. All edges in E go between R and D. An edge between a request vertex and a disk vertex exists if the segment

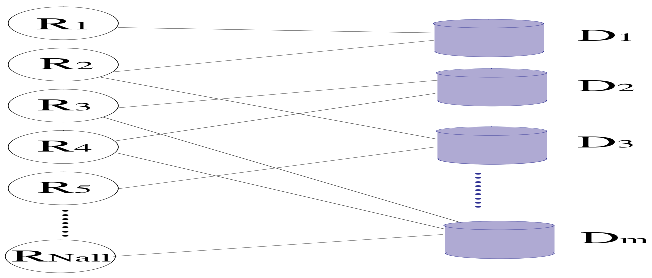


Figure 4: Request Scheduling

that the request next retrieves resides on that disk, and the weight assigned to the edge depends on the scheduling priority of the request. As we have discussed in Section 4, the priorities are ranked based on the requests' service deadlines, i.e., the earlier the deadline, the higher the priority. Since $T = N\Delta$, and each request is supposed to be serviced at least once in each T, N priority levels exist. The priority of a request is calculated by subtracting from N the time to the deadline in terms of the number of Δs.

The optimal schedule can be solved as a bipartite weighted matching problem. Let E_{ij} represent the edge between request R_i and disk D_j. Let w_{ij} denote the weight assigned to E_{ij}. Derigs [17] shows that this bipartite weighted matching problem can be formulated into a special case linear programming problem as follows:

$$Maximize \sum_{i=1}^{N_{all}} \sum_{j=1}^{M} w_{ij} E_{ij}$$

$$subject\ to \sum_{i=1}^{N_{all}} E_{ij} \leq 1,\ and \sum_{j=1}^{M} E_{ij} \leq 1,\ where\ E_{ij} = 0\ or\ 1.$$

$E_{ij} = 1$ means request R_i is assigned to disk D_j, otherwise $E_{ij} = 0$. Derigs shows that this problem can be solved using a generalized Hungarian method in $O(V^3)$ time. For scheme 2DB, since the number of nodes, V, includes $N_{all} = (N - N_b) \times M$ request nodes and M disk nodes, the computation cost is on the order of $O(N^3 M^3)$ (N_b is small with respect to N).

Although solving the linear programming problem yields an optimal schedule, the typically large $N_{all} = (N - N_b) \times M$ can make the computational time very long. Instead, we use a greedy procedure that assigns disks from the highest priority request group down to the lowest group until either all disks are used or no more requests can be assigned. This way, we need to evaluate only a small subset of N_{all} requests. Indeed, our experiment in Section 6 shows that with high probability all M disks are used after the top three (out of N) priority request groups are considered for the schedule. In addition, since the requests in each priority group have the same weight, the complexity of the problem is reduced to that of solving a maximum *unweighted* bipartite matching problem, which the Ford-Fulkerson method can solve in O(VE) time [15]. The number of nodes participating in the Ford-Fulkerson method starts from up to M request nodes in the top priority group and M disk nodes. The number of edges between them is up to $C \times M$. The computational cost for the top priority group is thus on the order of $O(M^2)$. The computational time for evaluating the subsequent request groups is much lower since with high probability the majority of the disk nodes will have been assigned by then and do not participate in the algorithm. The overall computational time is still on the order of $O(M^2)$. Now that M is much smaller than $(N - N_b) \times M$ and the computation is one order more efficient, the computational time can be reduced by more than a thousands fold.

Scheme 2DB

- Initialization:
  $D \leftarrow \{D_1, D_2, ..D_M\}$ /* Init the disk set */
  $R, \Lambda \leftarrow \emptyset$ /* Empty the request set and request queue */
- As new requests arrive, enter them into Λ.
- Before the start of every Δ:
  0. $p = N$; $D' \leftarrow D$; $\Psi \leftarrow \emptyset$
  1. For each $r \in R$:
     If the playback ends, $R \leftarrow R - \{r\}$
  2. While $((|R| < N_{all})$ & $(r = DeQueue(\Lambda) \neq nil))$
     $r.\Delta \leftarrow 0$; $r.new \leftarrow true$
     $r.latency, r.hiccups, r.IO \leftarrow 0$
     $R \leftarrow R \cup \{r\}$
  3. While $((p > 0)$ & $(D' \neq \emptyset))$
     $R' \leftarrow \emptyset$; $E \leftarrow \emptyset$; $\psi \leftarrow \emptyset$
     For each $r \in R$: If $(N - r.\Delta = p)$
     - $R' \leftarrow R' \cup \{r\}$
     - $r.IO.size \leftarrow \frac{(N-r.\Delta)S}{N}$ /* Compute IO size */
     - $r.CK \leftarrow FindChunks(r.IO)$ /* Find chunk numbers */
     - $r.D \leftarrow FindDisks(r.CK, D')$ /* Find disks */
     - For each $d \in r.D$:
       Create edge $E_{r,d}$
       $E \leftarrow E \cup \{E_{r,d}\}$

     $\psi = BipartiteMatching(R', D', E)$
     For each $E_{r,d} \in \psi$: $D' \leftarrow D' - \{d\}$
     $p \leftarrow p - 1$
     $\Psi \leftarrow \Psi \cup \psi$
  4. For each $E_{r,d} \in \Psi$: /* Perform IOs */
     $r.\Delta \leftarrow N$
     $DiskIO(d, r.IO)$
     $r.IO.S \leftarrow r.IO.S + ((r.IO.\kappa + r.IO.size)/S)$
     $r.IO.\kappa \leftarrow (r.IO.\kappa + r.IO.size)\ mod\ S$
  5. For each $r \in R$: /* Update statistics */
     If $(r.\Delta = 0)$
     - If $(r.new)$ $r.delay \leftarrow r.delay + 1$
     - Else $r.hiccup \leftarrow r.hiccup + 1$

     Else /* $r.\Delta > 0$ */
     - $r.\Delta \leftarrow r.\Delta - 1$
     - If $(r.new)$ $r.new \leftarrow false$

Figure 5: The 2DB Scheme

To specify 2DB we define the following parameters. At the start of each time slot (Δ) scheme 2DB assigns up to M requests in the R set (containing all requests) to the disks in the D set (containing M disks). For each request $r \in R$, scheme 2DB maintains five sets of data:

- $r.\Delta$: records the service deadline (number of Δs away).
- $r.IO$: records the information needed for doing IO, including five sub-fields:
  - $r.IO.D$: the disk set where the requested chunk resides,
  - $r.IO.CK$: the chunk that contains the accessed data,
  - $r.IO.S$: the first segment in $r.IO.CK$ that contains the accessed data,
  - $r.IO.\kappa$: the first byte in $r.IO.S$, where the accessed data begins.
  - $r.IO.size$: the size of the data retrieval.
- $r.latency$: records the number of Δs the request waits before its first IO is started.
- $r.hiccup$: records the number of hiccups that the request has suffered due to scheduling conflicts.
- $r.new$: records a newly arrived request.

In addition, we use Λ to represent the server's queue, where the newly arrived requests reside. Sets ψ and Ψ store the temporary and final schedule, respectively. The variable p denotes the priority levels, from N down to 1. Procedure *DeQueue(Λ)* removes the first request, if any, from the queue. Procedure *FindChunks* returns the chunks that contain a given segment. Procedure *FindDisks* returns the disks where the given chunks reside. Procedure *BipartiteMatching* returns ψ given a bipartite graph. Finally, procedure *DiskIO* performs a disk-to-memory data transfer given a chunk and a disk number.

Figure 5 specifies policy 2DB. Before the start of every Δ, the 2DB scheme initializes variables in step zero. After initialization, step 1 removes the requests that have finished their playback, if any, from the R set. Step 2 admits new requests if the server is not fully loaded ($|R| < N_{all}$). Step 3, the core of the procedure, assigns disks to requests, one priority level at a time, until either all disks are used or no more requests can be assigned. To find the candidate disk assignments for a request, this step first computes the data retrieval size by subtracting the amount of data left in the buffer ($(r.\Delta \times S)/N$) from S. Using the information stored in $r.IO$ (including $r.IO.s$, $r.IO.\kappa$, and $r.IO.size$), it then calls procedure *FindChunks* to find out in which chunks the desired data reside. Subsequently, it uses the chunk numbers to call procedure *FindDisks*, which locates the disks where the accessed data resides from among the unassigned disk set, D'. Edges are created between R' and the candidate disks. After the graph is built, procedure *BipartiteMatching* is called to obtain a partial disk assignment in ψ. Repeating these same steps (typically for less than four request groups) yields the final schedule, Ψ. Step 4 then sets the new deadline of the scheduled requests to $N\Delta$ and initiates a disk-to-memory data transfer for each scheduled request on the assigned disk. It also updates the pointer to the data the request will retrieve next time. Finally, step 5 updates the service deadlines for all requests and the applicable statistics.

6 Evaluation

We implemented a simulator to measure the server's performance, including its ability to balance disk load and its initial latency. We used the fraction of hiccups to measure if the disk load is balanced: if the load is balanced, the fraction of hiccups should be zero, otherwise, hiccups occur.

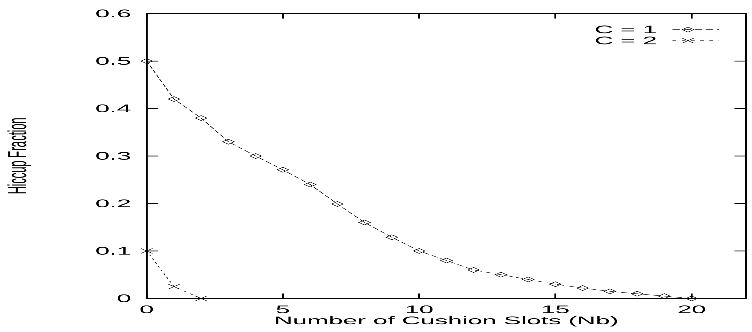


Figure 6: Hiccup Fraction vs. N_b

As we have discussed in Section 4, the 2DB scheme replicates popular movies and reserves cushion slots to balance disk load and to minimize initial latency. However, the parameters (listed below) must be chosen properly to virtually eliminate hiccups, We thus investigated the effects on the server's performance by various parameters including:

- The number of copies (C) of the hot movies (we assume that hot movies are the 20% of the movies that enjoy 90% of the requests),
- The number of cushion slots or the throughput reduction in T (N_b),
- The chunk size (CS) and chunk overlap size (CO), and
- The number of disks (M) and the maximum number of requests serviced per disk (N).

The values of these parameters can be changed through a configuration file.

To conduct the simulation, we used the Seagate Barracuda 4LP disk [1]. We assumed a display rate DR of 1.5 Mbps, which is sufficient to sustain typical video playback, and we used $M = 10$ disks and $N = 40$ requests per disk (and hence $T = 40 \times \Delta$). We show in Section 6.4 how the different values of Ms and Ns affect the performance.

We ran the simulator three times for every experiment, each time for 24 simulation hours. We collected statistics for the average hiccup fraction, the average initial latency, and the worst-case initial latency. We were not interested in the variance of the hiccup fraction since, any hiccups, regardless of their distribution, mean the quality of service is unacceptable. The remainder of this section describes the simulation results and our observations.

6.1 Fraction of Hiccups

We define the fraction of hiccups as the time that a request's buffer underflows over its playback duration. For instance, if the hiccup fraction is 0.1, the playback will be "blacked out" for one out of every ten seconds on average. We measure the fraction of hiccups after the server reaches and stays at the peak load, N_{all}, and measure how C and N_b affect the fraction of hiccups.

Figure 6 plots the average hiccup fraction (on the y-axis) versus N_b (on the x-axis) for two different values of C. When movies are not replicated ($C = 1$), the hiccup fraction is 0.5 at $N_b = 0$. It needs $N_b = 20$ cushion slots, or reducing throughput by 50%, to eliminate the hiccups. When two copies of

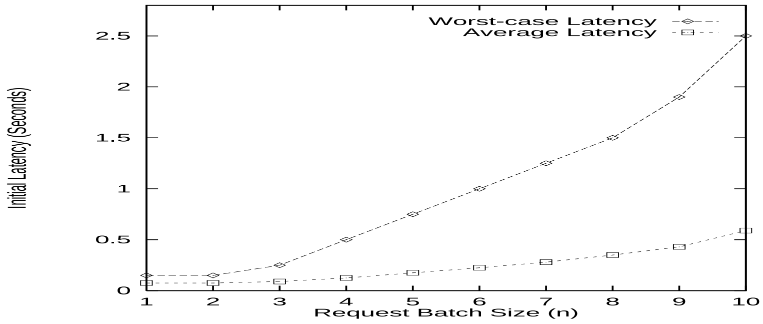


Figure 7: Initial Latency vs. n

the hot movies are available ($C = 2$), the figure shows that the hiccup fraction drops significantly. The fraction of hiccups is 0.1 when $N_b = 0$ and reaches zero when $N_b = 2$.

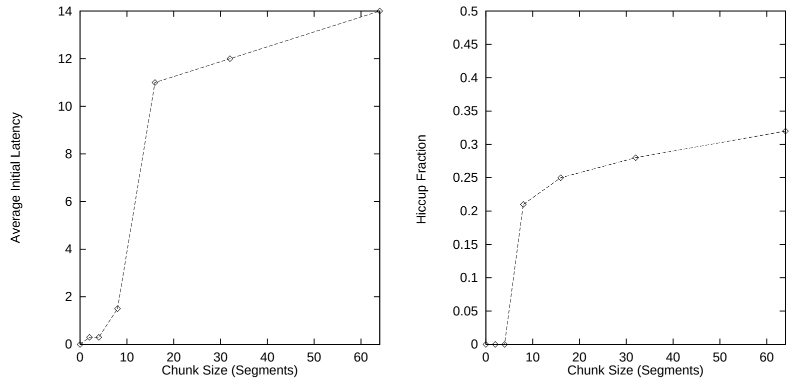
In addition to replicating the hot movies once (having two copies of each segment), we also experiment with having three, four, and five copies of the hot movies. Surprisingly, having more than two copies of the movies ($C > 2$) does not reduce the number of cushion slots ($N_b = 2$) necessary to virtually eliminate hiccups. The results of having more than two copies of the hot movies are similar to those shown in Figure 6 for two copies.

We use a similar problem, placing balls in runs, to explain this surprising result. Suppose that we sequentially place n balls into m urns by putting each ball into a randomly chosen urn. It has been shown by many studies (e.g., [2, 4, 29]) that there is a high probability the balls can be distributed among the urns quite unevenly. However, if for each ball we randomly select two locations and each ball is placed in the least full urn between these two possible locations, the number of balls can be spread out evenly among the urns with high probability. Giving each ball more than two locations for placement does not further smooth out the ball distribution significantly. The balls are analogous to the requests and the d possible destinations of the balls are analogous to C copies of the movies. This helps to explain the intuition behind why the disk scheduling conflict decays rapidly when $C = 2$, and the improvement is not significant when $C > 2$. Moreover, the balls are placed one at a time based on the past state of the urns. The 2DB scheme cannot do worse because it assigns M requests to M disks in each time slot by selecting the requests from the top priority groups: it balances disk load with future knowledge.

6.2 Initial Latency

Measuring initial latency when the server is not near its peak load is uninteresting, since plenty of free slots are available to service the newly arrived requests and hence the initial latency must be low. We, therefore, measured performance when the server was near its full load. We simulated the peak load as follows: after we first loaded up the server with N_{all} requests, at a random time in each service cycle we terminated n requests and at the same time let a batch of n requests arrive at the server. We did this for every service cycle. We simulated batch arrival to stress the server since if new requests arrive one at a time (without disk contention) the performance is certainly good. We tested different values of n from 1 to M.

Figure 7 shows the initial latency (on the y-axis) for $n = 1$ to M (on the x-axis), given $C = 2$ and



(a) Initial Latency (b) Hiccup Fraction

Figure 8: The Effect of the Chunk Size

$N_b = 2$. The figure shows both the average and worst-case initial latency over multiple runs. When $n \leq 3$, the average and worst-case initial latency is minimum: 75 milliseconds and 150 milliseconds, respectively. This is because the server can immediately assign these new requests to disks without encountering disk contention. The worst-case latency happens when the server has just started scheduling the existing requests for the next Δ when the requests arrive. The requests hence must wait until the next Δ to be serviced. When $n \geq 4$, the average delay starts to rise linearly with n while the worst-case delay grows super-linearly. This is because more requests may need to access their first segments on the same disk and as a consequence some must be delayed for extra Δs. Nevertheless, even when $n = 10$, the server is still able to keep the average initial latency under 0.5 second.

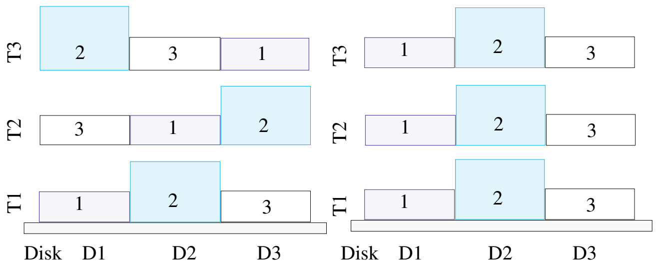
The initial latency achieved by the 2DB scheme is substantially lower than that achieved by any other proposed disk-based schemes (examples shown in Appendix A) for two reasons:

1. Most schemes use the elevator disk scheduling policy, which has an initial latency on the order of $O(N)$ [20] (about three to five seconds using the same configuration of our simulator).
2. Some schemes maximize throughput by delaying admission to newly arrived requests until the disk load can be balanced. These schemes suffer from very long initial latency, on the order of $O(NM)$ (the delay can be in minutes if M is large).

6.3 Chunk and Overlap Sizes

To replicate a movie once and thereby create two copies of it, one seems to be able to choose any chunk and chunk overlap size, as long as the chunk overlap size is 50% of the chunk size. For instance, either $CS = 2$ and $CO = 1$ and $CS = 4$ and $CO = 2$ gives us two copies of the movie. However, our experiment shows that a small chunk size is more desirable.

Figure 8 shows the effect of the chunk size on the initial latency and the hiccup fraction. (We use $C = 2$, $N_b = 2$, and $n = 5$ here to measure performance.) When the chunk size is small, from two to four segments, the initial latency and hiccup fraction remain low. When the chunk size grows beyond four segments, both the initial latency and display quality start suffering from severe degradation. We



(a) Small Chunk Size (b) Large Chunk Size

Figure 9: Disk Load vs. Chunk Size

attribute this to the large chunk size aggravating congestion on some bottleneck disks. When the chunk size is small, a request does not retrieve segments from the same disk for many consecutive service cycles. Thus, if a disk is the bottleneck in one service cycle, that same disk is very unlikely to remain the bottleneck in the next cycle, since the requests on it move quickly to other disks. If the chunk size is large, the congestion caused by a bottleneck disk is prolonged because requests stay on a disk for a larger number of cycles.

Figure 9 illustrates the effect of the chunk size on the bottleneck. The figure uses three blocks, one, two, and three, to represent three batches of requests. Batch two represents the workload that causes bottleneck. These three batches of requests are initially scheduled on disks D_1, D_2, and D_3, respectively (in service cycle T_1). Figure 9(a) shows that when the batches can be shifted from one disk to the next from service cycle to cycle, the average number of requests on each disk can be evenly distributed after three cycles. But when the batches stay on the same disk for some cycles (Figure 9(b)), the bottleneck (on disk D_2) aggravates. Thus, using small chunk size together with the round-robin chunk placement can shift the bottleneck quickly to balance the disk load, and using small chunk size is therefore desirable.

6.4 The Impact of M and N

So far we have used $M = 10$ for the number of disks and $N = 40$ for the number of requests per disk in the simulation. However, for a media server that services a large number of requests, M can be larger (M, however, cannot be too large due to bus bandwidth and contention issues). And as the disk technology advances, N can also grow rapidly. Our simulation shows that the minimum number of cushion slots required to avoid hiccups is insensitive to different values of M and N. The value of N_b remains two, given $C = 2$. This shows that the 2DB scheme is scalable in both the M and N dimensions. The initial latency, however, is sensitive to the value of M. If a group of requests tends to arrive at the same time asking for different movies, the larger the value of M, the more requests the server can attend to in Δ. This is analogous to the reduction in the requests' waiting time achieved by adding servers in a queuing model. Therefore, one may consider choosing M based on the arrival pattern of the new requests.

6.5 Fast-Scan Requests

Since we describe how BubbleUp supports fast-scan operations in detail in [8], we do not repeat the discussion here. In short, scheme 2DB can support fast-scan operations with negligible initial latency if we add more cushion slots. This allows other requests to be bumped in the schedule so that a fast-scan request is given a higher priority to be serviced. The priority of a fast-scan request can be ranked high or low depending on many factors such as the application types, the required quality of service (QoS), and how frequently a user has issued the fast-scan request. Using a priority scheduling scheme, scheme 2DB provides the flexibility to implement different admission control policies.

7 Conclusion

We have presented scheme two-dimensional BubbleUp for a media server to manage parallel disks. Through examples and simulations we have shown how the scheme places replicated data and allocates cushion slots to balance disk load. We have also shown that by bubbling up the free slots in the nearest future the scheme can minimize the initial latency for the newly arrived requests. Our simulation shows that the 2DB scheme reduces the initial latency substantially even when the requests arrive in a large batch and when the server is near its peak load.

It is important to note that the 2DB scheme 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 2DB can be above the device driver layer, since it only needs to queue an IO request after the completion of the previous one. To maintain cushion slots, the 2DB scheme trades a small fraction of throughput. However, we argue that since short response time is necessary for interactive multimedia applications and desirable for any others, the 2DB scheme is an attractive scheme to manage parallel disks.

References

[1] Seagate barracuda 9lp family product specification. *URL: http://www.seagate.com*, 1997.

[2] Y. Azar, A. Broader, A. Karlin, and E. Upfal. Balanced allocations. *ACM Symposium on Theory of Computing*, pages 593–602, December 1994.

[3] M. Bar, C. Griwodz, and L. Wolf. Long-term movie popularity models in video-on-demand systems. *Proceedings of ACM Multimedia Conference*, pages 349 – 357, November 1997.

[4] R. Barve, E. Grove, and J. Vitter. Simple ramdomized mergesort on parallel disks. *Parallel Computing*, 23(4-5):601–31, June 1997.

[5] S. Berson and S. Ghandeharizadeh. Staggered striping: A flexible technique to display continuous media. *Multimedia Tools And Applications*, 1(2):127–148, June 1995.

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

[7] S. Berson, R. Muntz, and W. Wong. Randomized data allocation for real-time disk i/o. *Proceedings of IEEE 41th COMPCON*, pages 286–90, 1996.

[8] E. Chang and H. Garcia-Molina. Bubbleup - low latency fast-scan for media servers. *Proceedings of the 5th ACM Multimedia Conference*, pages 87–98, November 1997.

[9] E. Chang and H. Garcia-Molina. Effective memory use in a media server. *Proceedings of the 23rd VLDB Conference*, pages 496–505, August 1997.

[10] E. Chang and H. Garcia-Molina. Reducing initial latency in media servers. *IEEE Multimedia*, 4(3):50–61, July-September 1997.

[11] E. Chang and A. Zaknor. Cost analysis for vbr video servers. *SPIE International Symposium on Electronic Imaging*, February 1996.

[12] M.-S. Chen, H.-I. Hsiao, C.-S. Li, and P. Yu. Using rotational mirrored declustering for replica placement in a disk-array-based video server. *ACM Multimedia Systems*, December 1997.

[13] S. Christodoulakis and F. Zioga. Data base design principles for striping and placement of delay-sensitive data on disks. *Proc. of PODS (to appear)*, 1998.

[14] 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.

[15] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. McGraw-Hill, 1990.

[16] A. Dan, D. Sitaram, and P. Shahabuddin. Scheduling policies for an on-demand video server with batching. *Proceedings of ACM Multimedia Conference*, pages 15–23, 1994.

[17] U. Derigs. A generalized hungarian method for solving minimum weight perfect matching problame with algebraic objective. *Discrete Applied Mathematics*, 1(2):167–80, 1979.

[18] M. N. Garofalakis, B. Ozden, and A. Silberschatz. Resource scheduling in enhanced pay-per-view continuous media databases. *Proc. 23rd VLDB*, pages 516–25, August 1997.

[19] G. Gibson, J. Vitter, and J. Wilkes. Strategic directions in storage io issues in large-scale computing. *ACM Computing Survey*, 28(4):779–93, December 1996.

[20] J. Korst. Random duplication assignment: An alternative to striping in video servers. *Proceedings of the 5th ACM Multimedia Conference*, pages 219–226, November 1997.

[21] 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.

[22] J. Krost, V. Pronk, and O. Coumans. Disk scheduling for variable-rate data streams. *Proceedings of the 4th IDMS*, September 1997.

[23] T. D. C. Little and D. Venkatesh. Popularity-based assignment of movies to storage devices in a video-on-demand system. *ACM Multimedia Systems*, 1994.

[24] D. Makaroff, G. Neufeld, and N. Hutchinson. An evaluation of vbr disk admission algorithms for continuous media file. *Proceedings of the 5th ACM Multimedia Conference*, pages 143–154, 1997.

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

[26] Y.-J. Oyang and C.-H. Wen. A multimedia storage system for on-demand playback. *ACM Multimedia*, 1997.

[27] B. Ozden, R. Rastogi, and A. Silberschatz. Disk striping in video server environment. *Proc. IEEE Multimedia Computing and Systems*, pages 17–23, June 1996.

[28] B. Ozden, R. Rastogi, and A. Silberschatz. Multimedia support for databases. *PODS*, pages 1–11, May 1997.

[29] A. Papoulis. *Probability, Random Variables, and Stochastic Processes, Second Edition*. McGraw-Hill, 1984.

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

[31] 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.

[32] Y. Wang, J. Liu, D. Du, and J. Hsieh. Efficient video file allocation schemes for vod services. *ACM Multimedia Systems*, 5, September 1997.

[33] J. Wolf, , P. Yu, and H. Shachnai. Disk load balancing for video-on-demand systems. *ACM Multimedia Systems*, December 1997.

[34] 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.

[35] R. Zimmerman and S. Ghandeharizadeh. Continuous display using heterogenous disk subsystems. *Proceedings of the 5th ACM Multimedia Conference*, pages 227–238, 1997.

Appendix A: Scheme Survey

In this section we review through examples various parallel disk management schemes proposed in the literature for multimedia servers. The schemes we present in this section include:

- Single Disk Placement (SD),
- Disk Striping (DS),
- Round-Robin Placement (RR), and
- Replication and Random Placement (RRP).

We study and compare these schemes' initial latencies.

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 the request's 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_{all} streams), as this time could be unbounded no matter what scheduling policy is in place. In other words, our focus is on evaluating the initial delay when both disk bandwidth and memory resources are available to service a newly arrived request.

7.1 Single Disk Placement (SD)

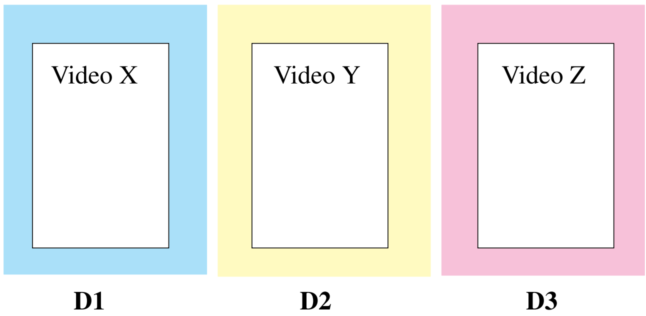


Figure 10: Single Disk Placement

The single disk data placement (SD) scheme places each movie entirely on one disk. (A disk can contain more than one movie.) Suppose we have three movies X, Y, and Z and three disks D_1, D_2, and D_3. Figure 10 shows that scheme SD can place movies X, Y, and Z on disks D_1, D_2, and D_3, respectively. Suppose each disk can service up to three requests and nine requests arrive consecutively, all asking for movie X. (We assume that the requests arrive a few seconds apart so batching [16] the requests is not feasible.) Scheme SD can service only three requests using disk D_1 and leaves disks D_2 and D_3 idle. Some of the literature defines this situation as *bandwidth fragmentation*: disk bandwidth is available but cannot be used to service requests. Bandwidth fragmentation can cause very long initial latency. For example, while three requests are viewing movie X, the fourth request that arrives and asks for movie X may have to wait as long as the duration of the movie. The SD scheme thus may not be desirable.

7.2 Disk Striping (DS)

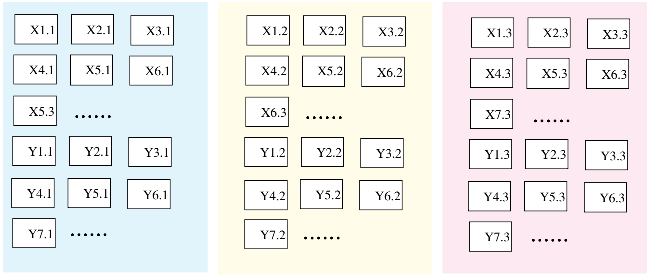


Figure 11: Disk Striping Placement

In this discussion of disk striping we refer to the fine-grained striping discussed in Section 1.1. Figure 11 shows two movies, X and Y, striped on three disks. Each segment of the movie is divided into three subsegments, each stored on one disk. For instance, the first segment of movie X, X_1, is divided into subsegments $X_{1,1}$, $X_{1,2}$, and $X_{1,3}$ and stored on disks D_1, D_2, and D_3, respectively. When a segment is retrieved, all three disks must operate at the same time transferring the subsegments.

Scheme DS does not suffer from bandwidth fragmentation since all disks share the workload at the same time. However, it suffers from long initial latency. Under the elevator-like disk scheduling policy, a newly arrived request, in the worst case, has to wait for the disks to service $(N \times M) - 1$ requests before being serviced. This delay can be minutes when M is large.

7.3 Round-Robin Placement (RR)

The studies of [26, 18] propose placing successive segments of a movie on a group of disks in a cyclic fashion. We call this policy scheme *RR*. Suppose we have M disks. Scheme RR places the i^{th} segment of the movie, S_i, on disk $D_{\theta+i-1 \ mod \ M}$ if the first segment of the movie is placed on disk D_θ.

Figure 12 shows an example with two movies, X and Y, each being placed on $M = 3$ disks. The j^{th} $(j = 1, 2, 3)$ disk stores segments $X_{(i \times M)+j}$ and $Y_{(i \times M)+j}$ for movies X and Y, respectively (i starts at zero). When a segment is retrieved, only one disk is involved in doing the IO. Thus, the disks operate independently from each other.

Scheme RR does not suffer from the bandwidth fragmentation problem, since a request uses the disks in a cyclic fashion following the placement of the segments (illustrated shortly in an example). It can, however, suffer from long initial latency. Suppose each disk can service up to three requests ($N = 3$). Suppose disks D_1 and D_3 are servicing three requests and disk D_2 two. If a new request arrives asking for either movie X or Y, the new request cannot be serviced immediately since the bandwidth of disk D_1, where the first segments of the movies reside, is saturated. Since the consecutive segments are placed in a cyclic fashion on the disks, the requests too are transferred in a cyclic fashion from disk to disk after each service period. For instance, if a request is serviced by disk D_1 in the current service cycle, the request will be serviced by disk D_2 in the next service cycle, by disk D_3 in the cycle after the next, and so on. Therefore, for the new request to be serviced, the request must wait until the available bandwidth, currently on disk D_2, is shifted to disk D_3 and then D_1. In general, the worst initial latency

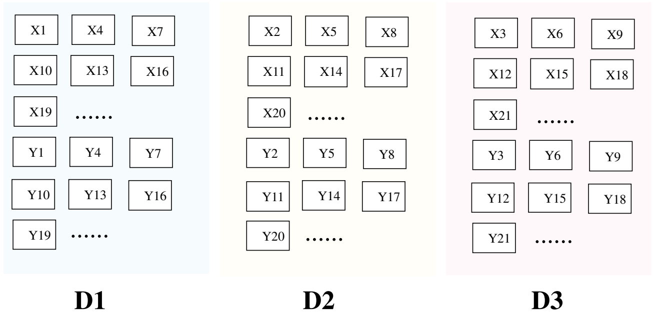


Figure 12: Round-Robin Placement

under scheme RR is on the order of $O(MN)$. For a large M, the initial latency reported in the literature can be minutes, which is clearly unacceptable for interactive multimedia applications.

7.4 Replication and Random Placement (RRP)

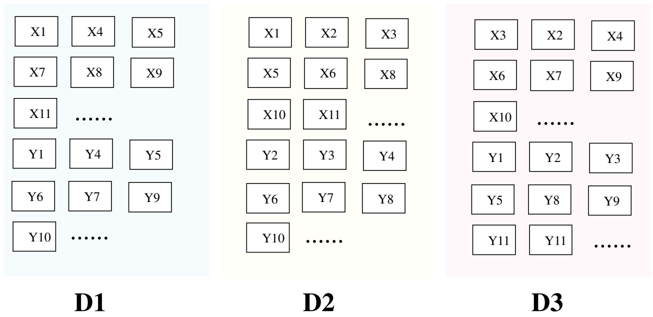


Figure 13: Replication and Random Placement

Randomized data placement was first proposed by Berson, Muntz, and Wong [7] to take advantage of the parity disk to balance disk load. Barve, Grove, and Vitter [4] proposed a random data placement polocy to improve the performance of merge-sort on parallel disks. The Replication and Random Placement (RRP) scheme proposed by Krost [20] adopts the random placement idea and adds data replication to it. Since each segment has multiple copies, a newly arrived request is more likely to be serviced in this scheme than in a scheme without replication. This approach statistically reduces the initial latency.

Figure 13 shows how movies X and Y are replicated and randomly placed on three disks. The example is the same as that used for explaining scheme RR. If a request arrives asking for movie X when both disks D_1 and D_3 are saturated, scheme RRP can service the request on disk D_2 since the disk holds the second copy of X_1. The initial latency is worst when the disk arm of D_2 has just passed X_1 and has to service the rest of the requests on disk D_2 first. As explained in [9, 10], this latency is $2 \times T$ or on the order of $O(N)$ (since T depends on N). In general, scheme RRP schedules requests for one period T at a time. If a request arrives after the scheduler has decided the schedule for the next T, the request must wait until the next period to be serviced.