Chapter 1

MANAGING PARALLEL DISKS FOR CONTINUOUS MEDIA DATA

Edward Chang
University of California, Santa Barbara
echang@ece.ucsb.edu

Chen Li and Hector Garcia-Molina
Stanford University
chenli@stanford.edu,hector@db.stanford.edu

Abstract In this study we present a scheme called two-dimensional BubbleUp (2DB) to manage parallel disks for continuous media data. 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 and fast-scan 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. 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 4.2 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 4.1 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 of the data required by some presentations can only be found at fully loaded disks. However, the frequency of hiccups can be reduced significantly 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. Our results (Section 4.) show that typically the reduction is a small percentage (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. As the disk speed increases with the improvement of the disk technology, this tradeoff will become even less significant in the near future.

2. FIXED-STRETCH

Before presenting the 2DB scheme, this section briefly reviews a disk scheduling policy *Fixed-Stretch*, which we presented in detail in Chang and Garcia-Molina, 1997a. Our proposed scheme, 2DB, presented in the next section, uses a modified version of *Fixed-Stretch*.

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 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

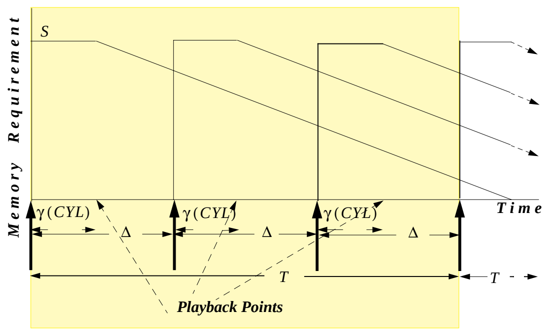

Figure 1.1 Service Slots of Fixed-Stretch

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 1.1 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 1.1 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 1.1), it sets the *end of slot 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.

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 1.1), 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. 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

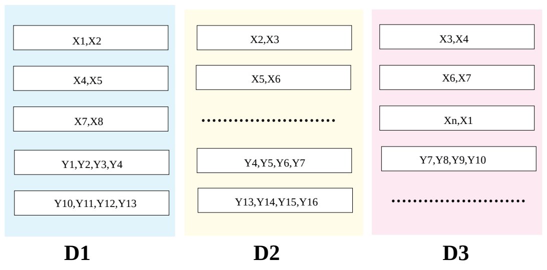

Figure 1.2 Data Placement Example

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 3., 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 Chang and Garcia-Molina, 1997a; Chang and Garcia-Molina, 1997b, 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 Yu et al., 1993). We show that Fixed-Stretch achieves throughput comparable to that of the other schemes, even though it has longer seek overhead. However, Fixed-Stretch conserves memory, which is the critical resource, and this compensates for the high seek overhead.

3. 2-DIMENSIONAL BUBBLEUP (2DB)

We now 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.

3.1 DATA PLACEMENT

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

1. Each chunk is 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 1.2 shows an example of chunk placement. 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 3.3 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 Azar et al., 1994; Barve et al., 1997. (We discuss the details in Section 4.1.)

3.2 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

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 1.1 The Arrival Time of the Requests

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 1.2 is saturated and a newly arrived request wants to start movie Y on the disk (segment Y_1 resides 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 4.1.

3.3 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 1.1 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

Deadlines (#Δ Away)	0	Δ	2Δ	3Δ
D_1			R_2	R_4
D_2				R_1
D_3				R_3

Table 1.2 2D BubbleUp Example - At the End of Δ_2

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 1.2 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 to Δ_6: Since the execution steps are similar, we skip the detailed description for these time slots. Table 1.3 summarizes the deadlines of the requests 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.,

Deadlines (#Δ Away)	0	Δ	2Δ	3Δ
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 1.3 2D BubbleUp Example - At the End of Δ_6

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 1.4 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 4.) 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 (#Δ Away)	0	Δ	2Δ	3Δ
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 1.4 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 4. a realistic simulation shows that all these observations hold and scale well with large Ns and Ms.

4. 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.

As we have discussed in Section 3, 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).

Due to the space limitation, we report only the first two experiments in this chapter. The results of the last two experiments are reported in Chang et al., 1998.

To conduct the simulation, we used the Seagate Barracuda 4LP disk. 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 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.

4.1 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 1.3 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 the hot movies are available ($C = 2$), the figure shows that the hiccup fraction

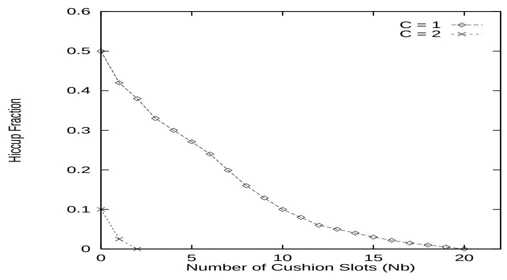

Figure 1.3 Hiccup Fraction vs. N_b

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 1.3 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., Azar et al., 1994; Barve et al., 1997; Berson et al., 1994; Papoulis, 1984) 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.

4.2 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

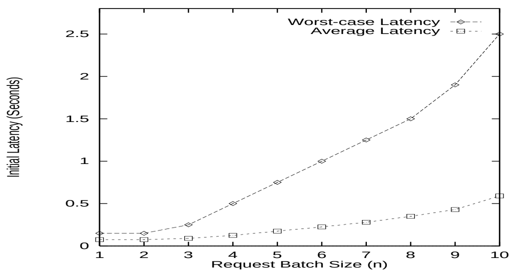

Figure 1.4 Initial Latency vs. n

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 1.4 shows the initial latency (on the y-axis) for $n = 1$ to M (on the x-axis), given $C = 2$ and $N_b = 2$. 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 for two reasons:

1. Most schemes use the elevator disk scheduling policy, which has an initial latency on the order of $O(N)$ (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).

5. 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

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

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

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

Chang, E. and Garcia-Molina, H. (1997a). Bubbleup - Low latency fast-scan for media servers. *Proceedings of the 5th ACM Multimedia Conference*, pages 87–98.

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

Chang, E., Garcia-Molina, H., and Li, C. (1998). 2d BubbleUp - Managing parallel disks for media servers. *Stanford Technical Report*.

Papoulis, A. (1984). *Probability, Random Variables, and Stochastic Processes, Second Edition*. McGraw-Hill.

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