Stream Combination: Adaptive IO Scheduling for Streaming Servers*

Bin Liu **Raju Rangaswami**
School of Computing and Information Sciences
Florida International University
Miami FL 33199
{bliu001,raju}@cis.fiu.edu

Zoran Dimitrijević†
Google, Inc.
1600 Amphitheater Pkwy
Mountain View CA 94043
zorand@gmail.com

Abstract

Cycle-based IO schedulers use statically configured time-cycle durations. As a result, they are unable to avoid the formation of virtual bottlenecks. We term a bottleneck as *virtual* when it occurs within a single resource subsystem, and it is possible to use a secondary under-utilized resource to thwart the bottleneck. The primary reason for virtual bottlenecks in streaming servers is static allocation of memory and disk-bandwidth resources using fixed time-cycle durations. As a result, shifting request workload can cause a virtual bottleneck either in the memory or disk subsystem. We present *stream combination*, an adaptive IO scheduling technique that addresses this problem in a comprehensive fashion. Stream combination predicts the formation of virtual bottlenecks and proactively alters the IO schedule to avoid them. A simulation study suggests significant performance gains compared to the current state-of-the-art fixed time-cycle IO scheduler.

1 Introduction

The web-based sharing and distribution of audio and video content is gaining ground with recent push from industry (e.g., Apple [1], Google [6]) combined with the widespread adoption of file-sharing tools (e.g., BitTorrent [3], KaZaa [8]). Support for streaming digital audio and video content during distribution provides flexibility by freeing up disk buffer-space at the client-side and reducing client wait-time before playback. In the past, researchers have investigated the problem of streaming media, mainly addressing the network-bandwidth bottleneck issue. In this article, we address server-side requirements such as *guaranteed-rate IO* and *high IO throughput* when serving multiple client requests.

The design goals of *guaranteed-rate IO* and *high throughput* within a streaming server requires establishing a trade-off between memory-use and disk-bandwidth utilization; this has been long recognized by designers of streaming multimedia systems [12, 13]. The underlying mechanism that determines this trade-off is the disk IO scheduling algorithm. Prior approaches to scheduling in real-time systems can be classified into two basic categories: *deadline-based priority scheduling* [7, 10, 13, 14] and *time-cycle-based scheduling* [2, 11, 12]. Deadline-based priority scheduling works excellently for CPU scheduling with provable guarantees for task completion. However, guaranteeing IO rate and performing disk admission control under this paradigm requires constant-overhead resource preemptibility [9], not feasible for disk-based systems, or at least not completely [5]. As a result, deadline-based priority scheduling in its current form, as proposed for CPU resources, cannot support guaranteed-rate IO delivery or a provably correct admission control mechanism, critical requirements for streaming servers.

The time-cycle-based IO scheduling technique, originally proposed as *quality proportional multi-subscriber servicing* (QPMS) by Rangan et al. [12], is a simpler and more popular model for streaming media servers. This is due to the fact that it supports guaranteed-rate IO and a provably correct admission control mechanism [2]. In this model, each stream is serviced exactly one IO per time-cycle and the retrieved data is written to a display buffer. The size of each IO is such that the display buffer does not underflow before the completion of the next IO for that stream. In a multi-bitrate streaming server, the buffer sizes for different streams could vary significantly, implying that the corresponding IO sizes could also be vastly different. Intuitively, the disk utilization depends on the average IO size, since this metric directly dictates the overhead component. Lesser the average IO size, greater the fraction of per-unit time spent on access overheads, and lower the disk utilization. In the time-cycle model, the disk utilization therefore depends on the bitrate of the streams serviced in each time-cycle. If the average bitrate of streams serviced in a time-cycle is low, the average IO size and the achieved disk throughput are low, potentially resulting in a *virtual disk-bandwidth bottleneck*. We call this a virtual bottleneck because this bottleneck is a result of a misconfigured time-cycle and may be avoided. One way to avoid this bottleneck and increase disk throughput would be to increase the duration of the time-cycle. However, increasing the time-cycle suddenly would result in display buffer underflow. Second, the server memory requirement would also increase as a result, increasing faster than the achieved disk utilization. Chang et al. analyze memory requirements in streaming servers extensively in [2]. A solution which can increase the average request size, without severely impacting the memory use would eliminate

*A preliminary version of this paper appears in the Proceedings of the 26th IEEE Real-Time Systems Symposium Work in Progress Session.

†This work was performed while the author was at the University of California, Santa Barbara.

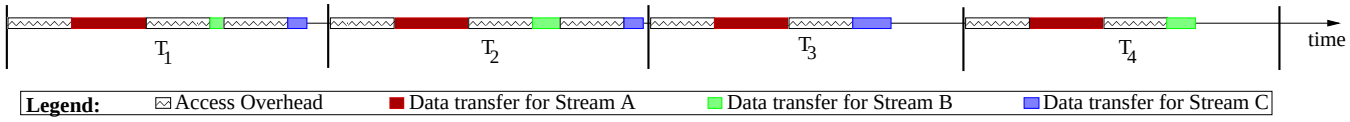


Figure 1: The Stream Combination Technique.

this virtual disk-bandwidth bottleneck.

Virtual memory-bottlenecks can occur as a result of a high average bitrate of streams. Higher the average bitrate, larger are the display buffer sizes, and consequently, greater the total memory requirement. In such situations, time-cycle duration reduction can be used to potentially avoid this virtual bottleneck. However, this reduction cannot occur after the bottleneck has been established. Proactive and dynamic reduction of time-cycle duration has not been extensively studied before.

In this paper, we propose *stream combination*, a variant of the time-cycle-based scheduling algorithm that dynamically adapts to changing system bottlenecks brought upon by shifting workloads. Stream combination provides guaranteed-rate IO and a provably correct admission control. Using a technique of combining and splitting IO streams and a technique for dynamic time-cycle alteration, it accounts for and avoids virtual disk- and memory- subsystem bottlenecks until these system resources are fully utilized.

The technique of stream combination recognizes that multi-bitrate streaming servers can encounter bottlenecks in either the memory or disk subsystem at any time. Stream-combination uses the time-cycle-based scheduling model as the underlying framework to allow deterministic admission control, but allows for variable per-stream time-cycles using stream combination as well as dynamic time-cycle reduction. Using these techniques, stream combination dynamically avoids subsystem bottlenecks by trading off memory use and disk utilization until all system resources are fully utilized. We make the following contributions in this paper:

1. We introduce the problem of virtual bottlenecks in multi-bitrate streaming servers.

2. We propose stream combination as a possible solution to the problem.

3. We evaluate the proposed technique and show that it can offer significant performance improvement over fixed time-cycle schedulers.

The rest of this paper is organized as follows. Section 2 describes stream combination, our proposed solution for avoiding virtual bottlenecks in streaming servers. In Section 3, using simulation, we evaluate the performance of a multi-bitrate streaming server to determine the effectiveness of the stream combination technique. Section 4 raises some interesting issues and avenues for future work with the Stream Combination IO scheduling technique. We make concluding remarks in Section 5.

2 Stream Combination

Stream combination is a variant of the time-cycle model that alters the IO schedule dynamically to avoid the formation of virtual bottlenecks in streaming servers. In this section, we present the rationale behind stream combination and the algorithm that drives this technique.

2.1 Rationale

Virtual bottlenecks can occur when servicing a dynamic streaming workload in either the memory or disk subsystem. Earlier, we noted that for virtual disk-bandwidth bottlenecks, simply increasing the time-cycle duration is not an acceptable solution. We investigate further to determine the root cause of disk IO inefficiency. For a stream with bitrate R serviced in a time-cycle of duration T, the amount of data retrieved in each IO is $R \times T$ and the amount of time spent to perform this IO is the sum of an (overhead) access time, T_{access}, and a data retrieval time, $\frac{R \times T}{R_{disk}}$, where R_{disk} is the data transfer rate from the disk medium. Therefore, the efficiency of the IO for the stream is:

$$e = \frac{R \times T}{R_{disk} \times T_{access} + R \times T} \quad (1)$$

Based on Equation 1, we note that a stream with high bitrate may have fair efficiency while a stream with low bitrate has poor efficiency. This raises the question: *Can we combine two or more low bitrate streams to obtain a single higher bitrate stream and improve IO efficiency?*

Figure 1 presents one possible combination technique. T_i denote time-cycle durations along a time axis. Streams A, B, and C are currently being serviced by the system. The bitrate of A is relatively high compared to B and C. In time-cycle T_1 (prior to combination), the IO scheduler performs one IO each per time-cycle per stream, retrieving S_A, S_B, and S_C amount of data respectively. The scheduler starts the stream combination process in time-cycle T_2 by retrieving twice the amount of data for stream B ($= 2 \times S_B$). In time-cycle T_3, the scheduler does not perform IO for Stream B, but retrieves twice the amount of data for stream C ($= 2 \times S_C$). Starting from time cycle T_3, in any given IO cycle, only one of streams B or C are serviced, reducing the number of access overheads by one, increasing the average IO size, and consequently improving disk utilization.

Although such a technique improves disk utilization as a result, several issues must be considered in a combination strategy: (i) the state of the system; combination makes sense only if disk-bandwidth is the bottleneck, (ii) combination must be proactive and must not allow the system to reach a bottleneck state before taking effect, (iii) how many streams must be combined to avoid the virtual bottleneck? (iv) combination increases memory requirement, and a wrong combination decision may potentially result in a virtual memory-bottleneck, (v) the combination operation incurs a transitory data transfer overhead during the time-cycle in which combination is initiated, and (vi) after combination, if there is a virtual memory-bottleneck at some later time due to shift in the workload, is *uncombining* or *splitting* combined streams straightforward?

The second virtual bottleneck is memory consumption. Assume that the first K out of N streams served by the system are in the combined state. If R_i is the bitrate of stream i and T denotes the time-cycle duration, the total

memory requirement for N streams is the sum of the display buffer sizes of all streams and is given by:

$$M = \sum_{i=1}^{N} T \times R_i + \sum_{i=1}^{K} T \times R_i \quad (2)$$

This equation follows from the observation that combined streams require buffering for two time-cycle durations as opposed to one time-cycle duration for uncombined streams. When the system approaches a potential virtual memory-bottleneck, it may be in one of two states: (a) there exist combined streams in the system, and (b) all streams are uncombined. In case (a), combined streams can be split to reclaim memory. In case (b), reducing time-cycle duration can reduce total memory requirement. However, three issues must be considered: (i) if several combined streams exist, which stream must be chosen to split first? (ii) how many combined streams should be split to avoid the bottleneck? (iii) by how much must the duration of the time-cycle be reduced to avoid the bottleneck? The answer to the question of which combined streams should be split first is straightforward. Splitting should be performed first on the high bitrate streams because they allow reclaiming the maximum amount of memory. However, the other issues need further investigation.

2.2 Mechanism

The basic idea of stream combination is to thwart virtual bottlenecks in streaming servers by proactively balancing memory and disk resource consumption under shifting stream workload. This balancing act is performed until both memory and disk resources are fully utilized. To balance these resources, we use two parameters, the memory utilization (u_m) and the time-cycle utilization (u_t). Memory utilization is the ratio of the utilized memory to the available memory, while time-cycle utilization is the ratio of the time spent performing IO during a time-cycle to the time-cycle duration. These parameters capture the relative availability of memory and disk-bandwidth resources.

A simplistic stream combination mechanism requires keeping track of u_m and u_t; when $u\_m < u\_t$, it combines the two lowest bitrate un-combined streams; when $u\_m > u\_t$, it un-combines or splits the highest bitrate combined stream. However, this straightforward strategy has several problems: (i) when choosing to combine, there may be no uncombined streams, (ii) when choosing to split, there may be no combined streams, (iii) this simplistic strategy would typically result in frequent combinations and splits, and (iv) several combination operations in a short duration can lead to a significant transitory disk-bandwidth overhead for transferring additional data for combined streams.

To avoid these problems, the stream combination IO scheduler uses four heuristics:

1. When combination is required and no uncombined streams exist, the scheduler doubles the duration of the time-cycle, effectively un-combining all streams. Notice that this increase in time-cycle duration incurs no overhead.

2. When splitting is required and no combined streams exist, the scheduler decreases the time-cycle by a UNIT percentage value, thereby reducing memory requirement. However, the disk utilization degrades due to a reduced average IO size. Here, we trade disk-bandwidth to conserve memory.

```
Input:     Current Workload (W),
           Current Schedule (CS)
Output:    New Schedule (NS)

Procedure: CheckSchedule {
  Compute {u_m,u_t} from {W,CS} ;
  If(((u_m>u_mT || u_t>u_tT) && abs(u_m-u_t)<u_dT)
      || SFLAG) { Call Reschedule ; }
}

Procedure: Reschedule {
  SFLAG = false ;
  If(u_m>u_t) {
    If(combinedStreamsExist) {
      Split highest bitrate combined streams ;
      Modify schedule to NS ;
    } Else { Decrease Time-cycle by UNIT ; }
    Recalculate {u_m,u_t} from {W,NS} ;
    If(abs(u_m-u_t)>u_dT) { SFLAG = true ; }
    return NS ;
  } Else {
    If(uncombinedStreamsExist) {
      Combine lowest bitrate uncombined streams ;
      Modify schedule to NS ;
    } Else { Double Time-cycle duration ; }
    Recalculate {u_m,u_t} from {W,NS} ;
    If(abs(u_m-u_t)>u_dT) { SFLAG = true ; }
    return NS ;
  }
}
```

Figure 2: Stream Combination Scheduler.

3. It makes provision for three constants, the memory utilization threshold (u_mT), the time-cycle utilization threshold (u_tT), and the difference threshold (u_dT). The decision to reschedule is made only in case either memory or time-cycle utilizations exceed their threshold and their difference is greater than the difference threshold.

4. When a decision to combine or split is made, the scheduler spreads out multiple required combine or split operations, allowing only one operation per time-cycle, thereby minimizing the transitory disk-bandwidth overhead. This is achieved using a scheduling flag (SFLAG).

The detailed IO scheduling algorithm is presented in Figure 2. The procedure CheckSchedule is invoked at the beginning of each time-cycle, which in turn invokes the Reschedule procedure if required.

2.3 Extension: N-way Stream Combination

So far, we have proposed combining two streams in each stream combination operation. Indeed, it is possible, and even sometimes desirable, to rather combine several streams at once. Allowing the combination of more than two streams when appropriate would make this technique more flexible by allowing for streams to be retrieved only once in $n > 2$ time-cycle durations. IO for each combined stream in a group of n combined streams increases n-fold, improving IO efficiency drastically. This flexibility also allows for significant reduction in the IO request-size variability, and consequently the IO bandwidth variability, over time. For a generic n-way stream combination, let us assume that the streams to be combined are numbered $1, 2, \cdots n$. In the final schedule, exactly one of streams $1 \leq i \leq n$ would be retrieved in each time-cycle. To transition the system from the initial state (each stream IO performed in each time-cycle) to the final schedule, stream combination could occur as follows. In the first time-cycle of the combination operation, for each stream i, the IO scheduler would retrieve data for i time-cycles. This completes the stream combination

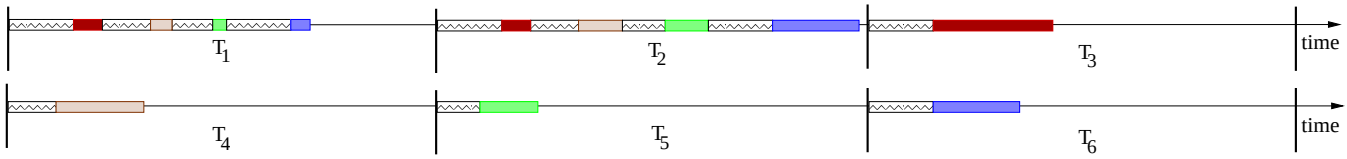


Figure 3: The N-way Basic Stream Combination Technique.

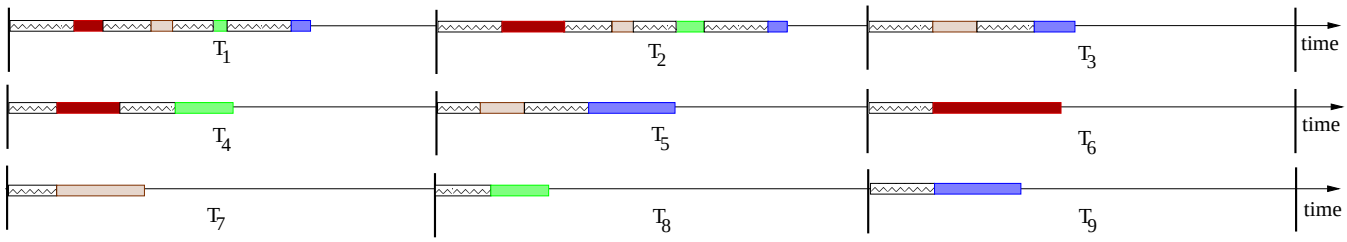


Figure 4: The N-way Pyramid Stream Combination Technique.

operation. In subsequent time-cycles, only one of streams $1 \leq i \leq n$ are retrieved, starting from stream 1 and increasing sequentially through n. The size of each IO henceforth is sufficient to last the stream n time-cycle durations. Of course, this is only one way of performing the stream combination operation and it essentially minimizes the duration of the stream combination operation to lead to the final IO schedule quickly, within one time-cycle. We term this technique as *n-way basic stream combination*, depicted in Figure 3. Time-cycle T_1 shows the original schedule. Combination of the four streams occurs during T_2.

In the n-way basic stream combination technique described above, although the duration of the stream combination operation is minimized (requires a single time-cycle to complete the n-way combination), it incurs significant data transfer overhead during the first time-cycle. This overhead may lead to deadline misses and make the technique unusable. Another generic technique to achieve the same schedule with lesser data transfer overhead could use the basic technique for combining two streams, and further combining the combined streams till the final schedule is reached. We call this the *n-way pyramid stream combination* technique. The pyramid technique staggers the data-transfer overhead of the combination operation over multiple time-cycles. Both techniques lead to the same final schedule. However, with the pyramid technique, as depicted in Figure 4., the stream combination operation takes longer to complete. The original schedule is shown in T_1. Pyramid combination occurs during time-cycles T_2, T_3, T_4, and T_5.

It is possible to combine two sets of combined streams and thereby further extend the stream combination technique. For instance combining a 3-way combined group and a 4-way combined group would lead to a 7-way combined group. Furthermore, note that splitting an n-way combined group is similar to splitting a 2-way combined group.

3 Experimental Evaluation

To evaluate the performance of the stream combination IO scheduler, we built a simulator to compare it with a fixed time-cycle scheduler. The system was configured to have 128MB of total available memory to buffer stream data. The maximum disk transfer-rate was 50MB/s and the average disk access time (including seek, rotational, and settle overheads) was 10ms. The base-line IO scheduler chosen was Fixed-Stretch [2], a state-of-the-art fixed time-cycle scheduler that balances disk-bandwidth and memory use. We evaluated the basic 2-way stream combination IO scheduler against the base-line Fixed-Stretch scheduler. Figure 5 tracks the following metrics during a simulation run of 20 minutes for a workload with uniformly distributed stream bitrates between 128 and 1024 kbps and with uniformly distributed request inter-arrival times between 2-7 seconds: (a) memory consumption (in MB), (b) time-cycle consumption (in milliseconds), (c) cumulative number of streams admitted over time, and (d) number of streams in service at any instant. The initial time-cycle duration for the stream combination scheduler was the same as that of the fixed time-cycle scheduler: 500 milliseconds. Initially, as streams arrive, the two scheduling strategies performed similarly. At approximately 200 seconds, the fixed time-cycle scheduler encountered a virtual disk-bandwidth bottleneck due to an under-estimated time-cycle duration. The stream combination scheduler detected the future formation of a virtual disk-bandwidth bottleneck and proactively started combining streams at approximately 100 seconds. As a result, it successfully thwarted the bottleneck. Beyond 200 seconds, the fixed time-cycle scheduler was unable to increase the number of streams in a time-cycle. Our scheduler was able to continue servicing greater number of streams in each time-cycle, delivering as much as 55% more throughput than the fixed time-cycle scheduler. The time consumption graph shows an increase beyond 500ms for stream combination because the time-cycle effectively doubles each time all streams in service have been combined. In this particular experiment, the time-cycle doubled twice to reach a maximum of 2000 milliseconds.

Figure 6 demonstrates the case where the initial time-cycle duration for both schedulers was set to 5 seconds. The generated workload was the same as for the previous experiment. At around 200 seconds into the simulation, the fixed time-cycle scheduler encountered a virtual memory-bottleneck that limited its throughput. Our scheduler proactively started reducing the duration of the time-cycle (and

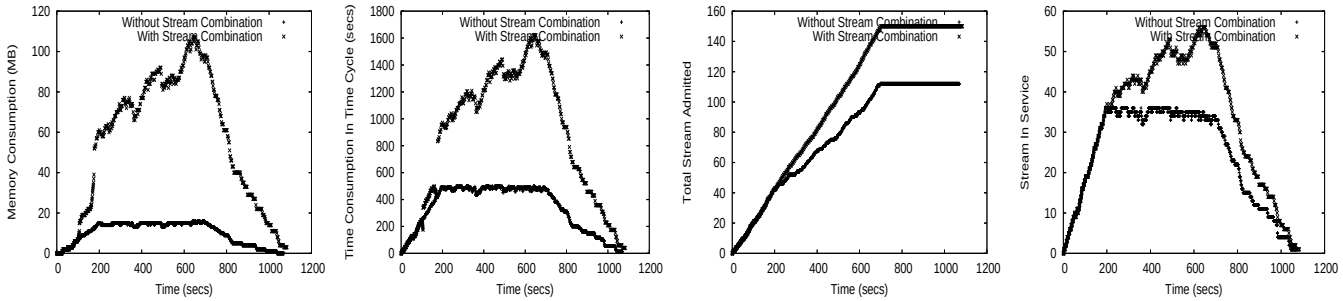


Figure 5: Comparison for time-cycle=500ms.

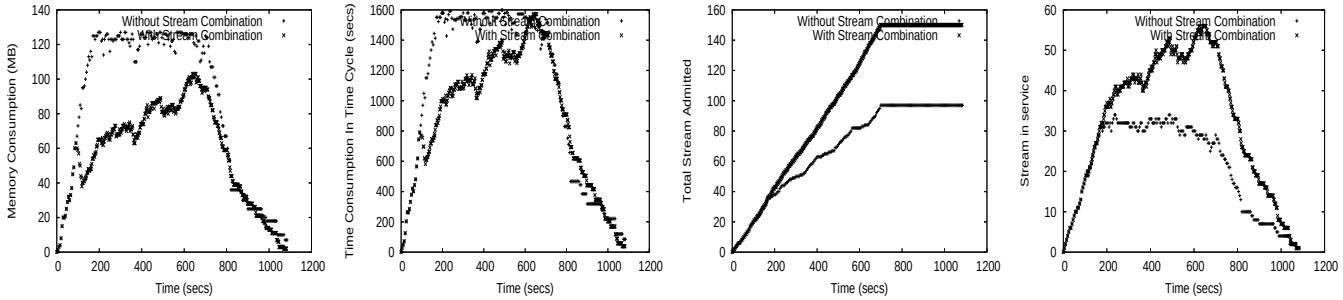
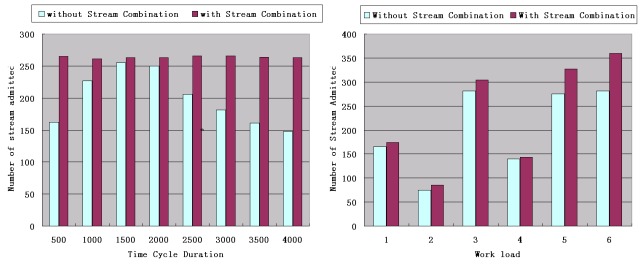


Figure 6: Comparison for time-cycle=5000ms.

the memory consumption as a result) at around 100 seconds (See Figure 6(b)) to successfully thwart the virtual bottleneck. More time-cycle reductions occurred beyond 200 seconds, dynamically adapting to the increased workload and delivering as much as 100% more throughput than the fixed time-cycle scheduler.

The above experiments demonstrated the inadequacy of a statically chosen time-cycle duration. We now determine how the throughput of the streaming server (in terms of the maximum number of streams admitted) depends on the time-cycle duration. Figure 7(a) compares against a fixed time-cycle scheduler with different time-cycle durations in each experiment, shown along the X-axis. The workload used was the same as for the previous experiments. As the initial time-cycle duration is changed, the fixed-time cycle scheduler admitted different number of streams, achieving its maximum for a time-cycle duration of 1.5 seconds. With stream combination, regardless of the initial time-cycle duration, the scheduler dynamically altered both its schedule as well as time-cycle duration to always provide the maximum throughput. It is important to note that, in case of the fixed time-cycle scheduler, determining the optimal time-cycle duration requires prior knowledge of the workload. Second, for real-world streaming servers, the natural shift in the workload over time precludes the existence of an optimal time-cycle duration. In such real-world scenarios, the stream combination scheduler dynamically adapts to deliver the maximum possible throughput.

Our final experimental result, depicted in Figure 7(b), compares the relative performance for six different workloads. These workloads were generated by varying both the distribution of stream bitrates as well as the arrival rates. Workloads #1-3 used stream bitrates generated from



(a) Varying time-cycle duration. (b) Varying workload.

Figure 7: Throughput comparison.

a uniform distribution. The time-cycle scheduler picked the time-cycle duration based on the average duration (assuming prior knowledge) and performed within 8% of the stream combination scheduler. Workloads #4-5 used a non-uniform distribution for stream bitrates; #4 favored high bitrates and #5 favored low bitrates. With workload #4, the primary bottleneck is memory and for #5, it is disk-bandwidth, with no virtual bottlenecks formed during these simulations. Even so, the stream combination scheduler was able to fine-tune the time-cycle duration to deliver as much as 15% more throughput for workload #5. Finally workload #6 varied the distribution over time to initially favor low bitrates and then high bitrates. The fixed time-cycle scheduler did not have a clear choice for the time-cycle duration and used the average bitrate as the basis. The stream combination scheduler dynamically varied the time-cycle duration over

time to better match the request traffic and delivered as much as 30% more throughput. It is important to note that real-world streaming workloads behave relatively more like workload #6 (probably with greater variations) than like workloads #1-5, underscoring the importance of stream combination.

4 Discussion

As demonstrated in our simulation-based evaluation, stream combination is a generic and powerful IO scheduling technique that dynamically avoids virtual bottlenecks. To recap, to avoid a virtual memory-bottleneck, the stream combination technique may employ time-cycle reduction. However, a future request workload with a low average bitrate may move the bottleneck to disk. In such a case, combination is employed. Using the basic 2-way combination technique, after all streams have been combined, the time-cycle is doubled. This increase in time-cycle comes at absolutely no cost. This completes a scheduling cycle that dynamically adapted to changing request workload, successfully avoiding virtual memory- and disk-bandwidth- bottlenecks.

4.1 Choosing Initial Time-cycle Duration

The stream combination technique requires choosing a time-cycle duration to begin with. With fixed time-cycle schedulers, the choice of this initial time-cycle duration is critical, since it statically determines the memory disk-bandwidth trade-off point. Stream combination, on the other hand, dynamically adapts the time-cycle duration to best match request workload. As a result, it is not necessary to choose the initial value of the time-cycle perfectly. In fact, as we have discussed earlier, for real-world workloads, there may not exist an ideal static time-cycle duration. Although this is subject to further study, we believe that for the stream combination scheduler, a time-cycle duration chosen based on the average bitrate of the streams stored at the server should work well.

4.2 Stream Combination and Deadline Scheduling

We have only investigated the stream combination technique for the family of cycle-based real-time IO schedulers. It would be interesting to investigate the appropriateness of stream combination, or the ideas contained therein, to the family of deadline-based real-time schedulers. However, we must first investigate the appropriateness of the deadline-scheduling family for real-time disk IO. Developing a deadline-scheduling theory for disk systems under the assumptions of non-preemptive tasks with large and non-uniform task-switching and task-preemption overheads is an interesting direction for future work.

5 Conclusion

We have presented stream combination, an IO scheduling technique that avoids virtual bottlenecks in streaming servers. This technique predicts subsystem bottlenecks and proactively alters the IO schedule to successfully thwart them until all system resources are fully utilized. Stream combination achieves its goal using the dynamic techniques of combining low-bitrate streams, splitting high-bitrate combined streams, and changing the time-cycle duration, as required. A simulation study suggests that this technique can offer significant performance improvement over fixed time-cycle schedulers. An implementation of the stream combination technique is currently being incorporated into Xtream [4], a real-time streaming multimedia system.

References

[1] Apple, Inc. Quicktime Streaming Server. *http://www.apple.com/quicktime/streamingserver/*.

[2] Edward Chang and Hector Garcia-Molina. Effective Memory Use in a Media Server. *Proceedings of the 23rd VLDB Conference*, pages 496–505, August 1997.

[3] B. Cohen. Incentives Build Robustness in Bittorrent. *Proceedings of the Workshop on Economics of Peer-to-Peer Systems*, May 2003.

[4] Zoran Dimitrijevic, Raju Rangaswami, and Edward Chang. The Xtream Multimedia System. *Proceedings of the IEEE Conference on Multimedia and Expo*, August 2002.

[5] Zoran Dimitrijevic, Raju Rangaswami, and Edward Chang. Design and Implementation of Semi-preemptible IO. *Proceeding of the Second Usenix FAST*, March 2003.

[6] Google, Inc. Google Video Search. *http://video.google.com/*.

[7] Kevin Jeffay, Donald F. Stanat, and Charles U. Martel. On Non-Preemptive Scheduling of Periodic and Sporadic Tasks. *Proceedings of the Twelfth IEEE Real-Time Systems Symposium*, December 1991.

[8] Nathaniel Leibowitz, Matei Ripeanu, and Adam Wierzbicki. Deconstructing the Kazaa Network . *Proceedings of The Third IEEE Workshop on Internet Applications*, page 112, June 2003.

[9] C Liu and J Layland. Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment. *ACM Journal*, January 1973.

[10] A. Molano, K. Juvva, and R. Rajkumar. Guaranteeing Timing Constraints for Disk Accesses in RT-Mach. *Proceedings of the IEEE Real Time Systems Symposium*, 1997.

[11] B. Ozden, A. Biliris, R. Rastogi, and A. Silberschatz. A Low-cost Storage Server for Movie On Demand Databases. *Proc. VLDB*, September 1994.

[12] P. Venkat Rangan, Harrick M. Vin, and Srinivas Ramanathan. Designing and On-Demand Multimedia Service. *IEEE Communications Magazine*, 30(7):56–65, July 1992.

[13] A L Reddy and J Wyllie. Disk Scheduling in a Multimedia I/O System. *Proceedings of the ACM Conference on Multimedia*, pages 225–233, 1993.

[14] Joel C. Wu and Scott A. Brandt. Storage Access Support for Soft Real-Time Applications. *Proceedings of the 10th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 164–173, May 2004.