MediaGuard: a Model-Based Framework for Building QoS-aware Streaming Media Services

Ludmila Cherkasova
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94303, USA
cherkasova@hpl.hp.com

Wenting Tang
Hewlett-Packard Laboratories
1501 Page Mill Road
Palo Alto, CA 94303, USA
wenting.tang@hp.com

Amin Vahdat
Dept of CS& E, UCSD
9500 Gilman Drive La Jolla,
CA 92093, USA
vahdat@cs.ucsd.edu

Abstract *A number of technology and workload trends motivate us to consider the appropriate resource allocation mechanisms and policies for streaming media services in shared cluster environments. First, workload measurements of existing media services indicate a "peak-to-mean" workload variance of more than one order of magnitude. It is difficult to overprovision service resources for such a highly variable workload, making the adaptive resource allocation and economies of scale of a shared hosting environment attractive for streaming media services. Second, in emerging workloads based on enterprise, news, and music content, a significant portion of the content is short and encoded at low bit rates. Additionally, media workloads display a strong temporal and spatial locality. This makes modern servers with gigabytes of main memory well suited to deliver most of the accesses to popular files from memory. Finally, end-point admission control for streaming services is more important than for traditional web services because a streaming media object delivered in the face of insufficient server resources is doubly bad, with wasted work at the server often resulting in aborted connection at the client.*

We present MediaGuard *– a model-based infrastructure for building QoS-aware streaming media services – that can efficiently determine the fraction of server resources required to support a particular client request over its expected lifetime. The proposed solution is based on a unified* cost function *that uses a single value to reflect overall resource requirements such as the CPU, disk, memory, and bandwidth necessary to support a particular media stream based on its bit rate and whether it is likely to be served from memory or disk. We design a novel, segment-based memory model of a media server to efficiently determine whether a request will incur memory or disk access when given the history of previous accesses and the behavior of the server's main memory file buffer cache. Using the MediaGuard framework, we design a novel, more accurate admission control policy for streaming media servers that accounts for the impact of the server's main memory file buffer cache. Our evaluation shows that, relative to a pessimistic admission control policy that assumes that all content must be served from disk, MediaGuard delivers a factor of two improvement in server throughput.*

1. INTRODUCTION

The Internet is becoming an increasingly viable vehicle for delivery of real-time multimedia content. The measurements of realistic streaming media workloads reveals that the "peak-to-mean" ratio of offered load varies by at least one order of magnitude [5]. To satisfy client requests under a variety of conditions in this environment would require similar overprovisioning of the service delivery infrastructure, daunting from both an economic and management perspective. In this paper, we consider the applicability of a shared utility computing infrastructures [4, 21, 26, 29, 30] to a streaming media service. The utility environment, where streaming media services may automatically request the necessary resources from the infrastructure to adapt to dynamically changing workload characteristics, becomes an attractive solution. It can satisfy the requirements of a wide variety of workloads with significantly less resources than would be required to satisfy the peak demands of all services simultaneously.

Traditionally, network bandwidth has been the target of optimizations for streaming media services because of the belief that other system resources, such as CPU, memory, and storage are relatively cheap to acquire. For our work, we take a more wholistic approach to resource management. While network bandwidth usage can be considered as a primary component in the service billing, the cost to manage, operate, and power more traditional resources makes CPU, memory, and storage important targets of resource management and allocation, certainly within a shared hosting environment.

A set of characteristics of emerging streaming media workloads further motivates our work. Earlier analysis [5] shows that emerging streaming workloads (e.g., for enterprise settings, news servers, sports events, and music clips) exhibit a high degree of temporal and spatial reference locality. Additionally, a significant portion of media content is represented by short and medium videos (2 min-15 min), and the encoding bit rates, targeting the current Internet population, are typically 56 Kb/s - 256 Kb/s CBR. These popular streaming objects have footprints on the order of 10 MB. At the same time, modern servers have up to 4 GB of main memory, meaning that most of the accesses to popular media objects can be served from memory, even when a media server relies on traditional file system and memory support and does not have additional application level caching. Thus, the locality available in a particular workload will have a significant impact on the behavior of the system because serving content from memory will incur much lower overhead than serving the same content from disk.

A final motivation for our work is the observation that the adequate resource allocation or the admission control at the service endpoint is more important for streaming services than for traditional Internet services. A request for a web page to a busy web site may result in a long delay before the content is returned. However, once the object is retrieved its quality is typically indistinguishable to an end user from the same object that may have been returned more quickly. However, for an interactive streaming media service, individual frames that are returned after a deadline lead to an inadequate service, interfering with QoS requirements and the client's ability to provide continuous playback, and resulting in the aborted connection at the client. Thus, a streaming media server must ensure that sufficient resources are available to serve a request (ideally for the duration of the entire request). If not, the request should be rejected rather than allow it to consume resources that deliver little value to the end user, while simultaneously degrading the QoS to all clients already receiving content.

One of the difficult problems for real-time applications including streaming media servers is that monitoring a single system resource cannot be used to evaluate the currently available server capacity: the CPU/disk utilization and the achievable server network bandwidth highly depend on the type of workload. In this paper, we present *MediaGuard* – a model-based framework for building QoS-aware streaming media services – that can efficiently determine the fraction of server resources required to support a particular client request over its lifetime and, as a result, to evaluate currently available server capacity. In this context, *MediaGuard* promotes a new unified framework to:

- Measure media service capacity via a set of basic benchmarks.
- Derive the *cost* function that uses a single value to reflect the combined resource requirements (e.g., CPU, disk, memory, and bandwidth) necessary to support a particular media stream. The *cost* function is derived from the set of basic benchmark measurements and is based on the stream bit rate and the file access type: memory vs disk.
- Determine the type of file access, i.e., whether a request (or its fraction) can be served from memory (or disk), using a novel, *segment-based memory model* of the media server, based on the history of previous accesses and the behavior of the server's main memory file buffer cache.
- Calculate the level of available system resources as a function of time to provide QoS guarantees.

We believe that the *MediaGuard* framework provides a foundation for building QoS-aware streaming media services. For example, consider the case where a shared media service simultaneously hosts a number of distinct media services. The services share the same physical media server, and each service pays for a specified fraction of the server resources. For such a shared media hosting service, the ability to guarantee a specified share of server resources to a particular hosted service is very important.

The problem of allocating $X_s\%$ of system capacity to a designated media service s is inherently similar to an admission control problem: we must admit a new request to service s when the utilized server capacity by service s is below a threshold $X_s\%$ and reject the request otherwise. Commercial media server solutions do not have "built-in" admission control to prevent server overload or to allocate a predefined fraction of server resources to a particular service.

Using *MediaGuard* framework, we design a novel, more accurate admission control policy for streaming media server (called *ac-MediaGuard*) that accounts for the impact of the server's main memory file buffer cache. Our performance evaluation reveals that *ac-MediaGuard* can achieve a factor of two improvement relative to an admission control policy that pessimistically assumes all accesses must go to disk. Our simulation results show that more than 70% of client requests can be served out of memory, and these requests account for more than 50% of all the bytes delivered by the server even for the media server with relatively small size of file buffer cache. Our workloads are based on large-scale traces of existing media services.

The rest of this paper is organized as follows. Section 2 briefly reviews the related work. Section 3 outlines our approach on media server benchmarking. Section 4 introduces the segment-based memory model. Section 5 further presents the main components of *MediaGuard* framework and develops the *MediaGuard* admission controller. We present the performance evaluation results in Section 6. Finally, we conclude with a summary and directions for future work in Section 7.

2. RELATED WORK

A large-scale multimedia server has to service a large number of clients simultaneously. Given the real-time requirements of each client, a multimedia server has to employ admission control algorithms to decide whether a new client request can be admitted without violating the quality of service requirements of the already accepted requests. There has been a large body of research work on admission control algorithms. Existing admission control schemes were mostly designed for disk subsystems and can be classified by the level of QoS provided to the clients [27].

Deterministic admission control schemes provide strict QoS guarantees, i.e. the continuous playback requirements should never be violated for the entire service duration. The corresponding admission control algorithms are characterized by the worst case assumptions regarding the service time from disk [11, 22, 25, 28, 31, 6]. Most of the deterministic admission control schemes conservatively constrain the number of clients that can be serviced simultaneously, and hence lead to severe underutilization of server resources.

In [27, 14, 2, 12], *statistical* admission control algorithms are designed which provide probabilistic QoS guarantees instead of deterministic ones, resulting in higher resource utilization due to statistical multiplexing gain. Statistical admission control uses statistics of the stored data to ensure that the probability of the overload does not exceed a predefined threshold. Most of the papers are devoted to storage and retrieval of variable bit rate (VBR) data. Good comparison and analysis of deterministic versus statistical admission control strategies is provided in [15, 3, 17].

Continuous media file servers require that several system resources be reserved in order to guarantee timely delivery of the data to clients. These resources include disk, network and processor bandwidth. A key component of determining the amount of a resource to reserve is characterizing each streams' bandwidth. In [15, 19, 18] the admission control is examined from the point of *network bandwidth allocation* on the server side.

A large portion of multimedia storage research has focused on optimizing disk bandwidth via scheduling policies or data placement schemes. However, there is another critical resource that has not received as much attention: the main memory that holds data coming off the disk. In media servers, requests from different clients arrive independently. Commercial systems may contain hundreds to thousands of clients. Providing an individual stream for each client may require very high disk bandwidth in the server, and therefore, it can become a bottleneck resource, restricting the number of concurrently supported clients in the system. There have been several studies on *buffer sharing* techniques to overcome the disk bandwidth restriction [7, 13, 20, 23]. The basic idea behind buffer sharing is as follows: if two clients request the same video at different points in time, the server may service the latter one by using the data which is read into the buffer pool on behalf of the former one. Therefore the requested video is read from disk only once, while the application will be supporting two different client requests to this file. Closely related to this technique is an *interval caching* scheme proposed in [8] which selects data blocks to be cached by media server based on the interval between two consecutive streams referencing the same object. Our work proceeds further in the same direction. We design a high-level model of a traditional memory system with LRU replacement strategy as used in today's commodity systems to reflect and quantify the impact of system level caching in delivering media applications for typical streaming media workloads.

The current trend of outsourcing network services to third parties has brought a set of new challenging problems to the architecture and design of automatic resource management in Internet Data Centers [1, 4, 21, 10, 26, 9]. In [4, 21, 9], the authors design a framework and architecture for resource management in IDCs using web server workloads. In [1], the authors propose resource containers as a new OS abstraction that enables fine-grained resource management in network servers. In our work we take a different approach by building a framework that maps the application demands into the resource requirements and provides the application-aware wrapper for managing the server resources. In [10], the authors propose a scheduling and admission control algorithm optimizing streaming media server performance in an IDC environment when network bandwidth is a resource bottleneck. In our work, we address the resource allocation/admission control problem in a more general setting, where under the different workloads, the different system resources may limit a media server performance. In [26, 9], the authors promote the necessity of application profiling and adequate system/workload/application models, facilitating a utility service design. Our work follows similar approach and motivation and proposes corresponding models for an efficient streaming media utility design.

3. MEASURING MEDIA SERVER CAPACITY

Commercial media servers are typically characterized by the number of concurrent streams a server can support without loosing a stream quality, i.e. until the real-time constraint of each stream can be met. A standard commercial stress test measures a maximum number of concurrent streams delivered by the server when all the clients are accessing the same file encoded at a certain bit rate, e.g. 500 Kb/s. However, a multimedia content is typically encoded at different bit rates depending on a type of content and a targeted population of clients and their connection bandwidth to the Internet. What are *the scaling rules for server capacity* when delivered media content encoded at different bit rates? For example, if a media server is capable of delivering N concurrent streams encoded at 500 Kb/s, will this server be capable of supporting $2 \times N$ concurrent streams encoded at 250 Kb/s? The other issue with a standard commercial stress test is that all the clients are accessing the same file. Thus another question to answer is: how the media server performance is impacted when different clients retrieve different (unique) files of a media content?

We use two basic benchmarks for establishing the scaling rules for server capacity when multiple media streams are encoded at different bit rates:

- *Single File Benchmark* – measuring a media server capacity when *all the clients* in the test are accessing the *same file*;
- *Unique Files Benchmark* – measuring a media server capacity when *each client* in the test is accessing a *different file*.

Each of these benchmarks consists of a set of sub-benchmarks with media content encoded at a different bit rate (in our study, we used six bit rates representing the typical Internet audience: 28 Kb/s, 56 Kb/s, 112 Kb/s, 256 Kb/s, 350 Kb/s, and 500 Kb/s). Using an experimental testbed with standard components and a proposed set of basic benchmarks, we measured capacity and scaling rules of a media server running RealServer 8.0 from RealNetworks. The configuration and the system parameters of our experimental setup are specially chosen to avoid some trivial bottlenecks when delivering multimedia applications such as limiting I/O bandwidth between the server and the storage system, or limiting network bandwidth between the server and the clients.

Under the *Single File Benchmark*, the media server is CPU bounded: CPU reaches 100% of utilization, and it is the main resource which limits server performance. In essence, under the *Single File Benchmark* only one stream reads a file from the disk, while all the other streams read the corresponding bytes from the file buffer cache. Thus, practically, this benchmark measures a streaming server capacity when the media content is delivered from memory. For *Unique File Benchmark*, the CPU utilization is much lower than for *Single File Benchmark*. For all the tests in this study, it is below 45% and it is not a resource which limits server performance. Under the *Unique Files Benchmark*, the server performance is disk-bound: this particular bottleneck is hard to measure with the usual performance tools. The maximum bandwidth delivered by a disk depends on the number of concurrent streams it can support without violating on-time delivery constraints.

Figure 1 a) shows the normalized graph reflecting the scaling rules for the media server capacity under the *Single File Benchmark* and different encoding bit rates. In this figure, point (1,1) presents the maximum capacity achievable by a server when all the clients in the test are accessing the same file encoded at a 500 Kb/s bit rate. Each absolute value for the other encoding bit rates is normalized with respect to it. Figure 1 b) shows the similar normalized graph for the *Unique File Benchmark*.

The measurement results show that the scaling rules for server capacity when multiple media streams are encoded at different bit rates are non-linear. For example, the difference between the highest and lowest bit rates of media streams used in our experiments is 18 times. However, the difference in the maximum number of concurrent streams a server is capable of supporting for corresponding bit rates is only around 9 times for the *Single File Benchmark* as shown in Figure 1 a), and 10 times for the *Unique Files Benchmark* as shown in Figure 1 b).

In our study, the media server performance is 2.5-3 times higher under the *Single File Benchmark* than under the *Unique Files Benchmark* as shown in Figure 2 a). This quantifies the performance benefits for multimedia applications when media streams are delivered from memory. Figure 2 b) shows the corresponding maximum bandwidth in Mb/s delivered by the media server for the *Single* and *Unique File Benchmarks*. These results are sensitive to a disk/file subsystem used in the media

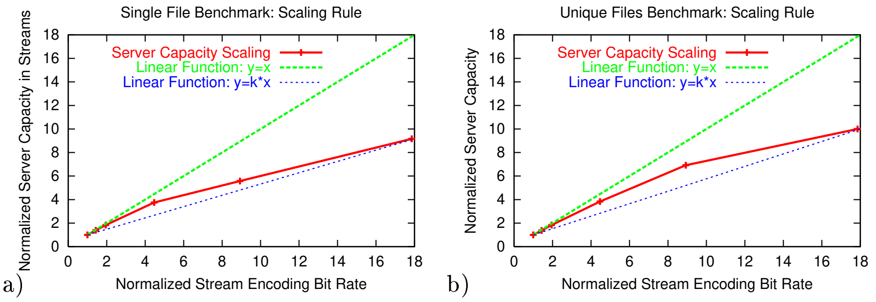

Figure 1: a) Server capacity scaling rules under: a) *Single File Benchmark*; b) *Unique File Benchmark*.

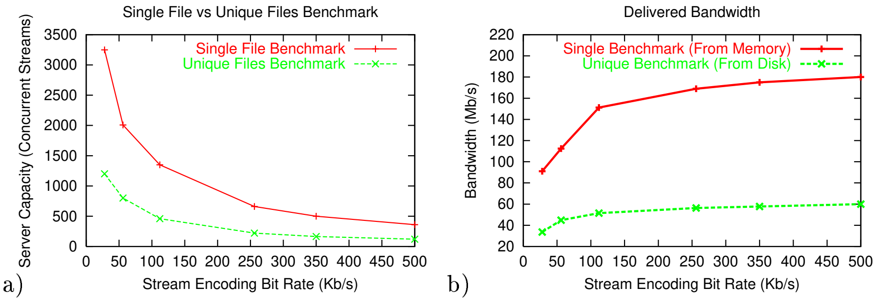

Figure 2: Comparison of server capacity for *Single File* vs *Unique Files Benchmark*: a) the number of requests, b) delivered bandwidth.

server configuration. In our benchmark experiments, we evaluated different file systems and disk configurations: it is not unusual, when a media server throughput is 3-7 times lower under the *Unique Files Benchmark* compared to achievable throughput under the *Single File Benchmark*, i.e. the maximum number of media streams delivered from memory can be 3-7 times higher than the maximum number of media streams delivered from disk.

The measurement results lead to an interesting conclusion: the bottleneck resource that limits server performance as well as the maximum bandwidth delivered by the media server are highly workload dependent. Different system resources are limiting a media server performance under the different basic benchmarks. Under the *Single File Benchmark*, the server performance is CPU bounded, while under the *Unique Files Benchmark*, the server performance is disk-bound. A typical media server traffic presents a combination of some clients accessing the same files (during the same period of time) and some clients retrieving the "unique", different files. Under the "mixed" workload, neither a CPU nor a disk are the limiting performance system resources, and hence, they can not be used for monitoring of the available server capacity.

The next question to answer is: how to compute the expected media server capacity for a realistic workload if the measured capacities of a streaming media server under the basic benchmarks are given. First, we introduce a *cost* function derived from the set of basic benchmark measurements. Intuitively, the cost function defines the *fraction* of system resources needed to support a particular stream depending on the file encoding bit rate and the access type of the corresponding file.

With each stream delivered by a server we associate an access type:

- *memory access* if a stream retrieves a corresponding file (or the corresponding bytes of the file) from memory;
- *disk access* if a stream retrieves a corresponding file from disk.

Additionally, we use the following notations:

- $X = X_1, ..., X_k$ - a set of encoding bit rates of the files used in basic benchmarks,
- $N^{single}_{X_i}$ - the maximum measured server capacity in concurrent streams under the *Single File Benchmark* for a file encoded at X_i Kb/s,
- $N^{unique}_{X_i}$ - the maximum measured server capacity in concurrent streams under the *Unique Files Benchmark* for a file encoded at X_i Kb/s,
- $cost^{memory}_{X_i}$ - a value of cost function for a stream with memory access to a file encoded at X_i Kb/s,
- $cost^{disk}_{X_i}$ - a value of cost function for a stream with disk access to a file encoded at X_i Kb/s.

Under the *Unique Files Benchmark*, all the streams have a disk access type. Hence each stream requires a fraction of system resources defined by the $cost^{disk}_{X_i}$ value.

Under the *Single File Benchmark*, the initial stream reads the corresponding file from disk, while the rest of the streams retrieve the corresponding bytes from memory and, therefore require a fraction of system resources defined by the $cost^{memory}_{X_i}$.

Let 1 be the media server capacity. The following *capacity equations* describe the maximum server capacity measured under a set of basic benchmarks for each encoding bit rate $X_i \in X$:

$$N^{unique}_{X_i} \times cost^{disk}_{X_i} = 1$$

$$1 \times cost^{disk}_{X_i} + (N^{single}_{X_i} - 1) \times cost^{memory}_{X_i} = 1$$

By solving the equations above, we can derive the corresponding cost function values:

$$cost^{disk}_{X_i} = \frac{1}{N^{unique}_{X_i}}$$

$$cost^{memory}_{X_i} = \frac{N^{unique}_{X_i} - 1}{N^{unique}_{X_i} \times (N^{single}_{X_i} - 1)}$$

Let W be the current workload processed by a media server, where

- $X_w = X_1, ...X_{k_w}$ - a set of encoding bit rates of the files used in W ($X_w \subseteq X$),
- $N^{memory}_{X_{w_i}}$- a number of streams having a memory access type for a subset of files encoded at X_{w_i} Kb/s,
- $N^{disk}_{X_{w_i}}$ - a number of streams having a disk access type for a subset of files encoded at X_{w_i} Kb/s.

Then the applied $Load$ to a media server under workload W can be computed by a formula:

$$Load = \sum_{i=1}^{k_w} N_{X_{w_i}}^{memory} \times cost_{X_{w_i}}^{memory} + \sum_{i=1}^{k_w} N_{X_{w_i}}^{disk} \times cost_{X_{w_i}}^{disk} \quad (1)$$

If $Load \leq 1$ then the media server operates within its capacity, and the difference $1 - Load$ defines the amount of available server capacity. We validated this performance model by comparing the predicted (computed) and measured media server capacities for a set of different synthetic workloads (with statically defined request mix). The measured server capacity matches the expected server capacity very well for studied workloads (with the error 1%-8%).

Introduced *cost* function uses a single value to reflect the combined resource requirement such as CPU, disk, memory, and bandwidth necessary to support a particular media stream depending on the stream bit rate and type of the file access: memory or disk access. The proposed framework provides a convenient mapping of a service demand (client requests) into the corresponding system resource requirements. If there is an additional constraint on the deployed server network bandwidth it can be easily incorporated in the equation of the server capacity. For a given constant bit rate media request, it is straightforward to determine whether sufficient network bandwidth is available at the server.

4. SEGMENT-BASED MEMORY MODEL

In order to assign a *cost* to a media request, we need to evaluate whether a new request will be streaming data from memory or will be accessing data from disk. Note, that memory access does not assume or require that the whole file resides in memory: if there is a sequence of accesses to the same file, issued closely to each other on a time scale, then the first access may read a file from disk, while the subsequent requests may be accessing the corresponding file prefix from memory. Thus, in order to accurately assign a *cost* to a media request, a model reflecting which file segments are currently residing in memory is needed. Taking into account the real-time nature of streaming media applications and the sequential access to file content, we design a novel, *segment-based memory model* reflecting data stored in memory as a result of media file accesses. This model closely approximates the media server behavior when the media server operates over a native OS file buffer cache with LRU replacement policy.

4.1 Basic Definitions and Notations

For each request r, we define the following notations.

- $file(r)$ – the media file requested by r.
- $duration(r)$ – the duration of $file(r)$ in seconds.
- $bitRate(r)$ – the encoding bit rate of the media file requested by r. In this paper, we assume that files are encoded at constant bit rates.
- $t^{start}(r)$ – the time when a stream corresponding to request r starts (once r is accepted).
- $t^{end}(r)$ – the time when a stream initiated by request r terminates. In this work, we assume non-interactive client sessions [1] which continue for a designated file duration: i.e. once a request is accepted, it will proceed until the end: $duration(r) = t^{end}(r) - t^{start}(r)$.

[1]Proposed approach, models, and algorithms can be extended with some modifications for the general case.

The real-time nature of streaming media applications suggests the following high-level memory abstraction. Let request r be a sequential access to file f from the beginning of the file. For simplicity, let it be a disk access. Then after $10\ sec$ of access r, the content, transferred by a server, corresponds to the initial $10\ sec$ of the file. The duration of transferred file prefix defines the number of bytes [2] transferred from disk to memory and further to the client: in our example, it is $10\ sec \times bitRate(r)$. Moreover, the real-time nature of file access defines the relative time ordering of streamed file segments in memory. It means that the time elapsed from the beginning of the file (we use $0\ sec$ to denote the file beginning) can be used to describe both the streamed file segment and the relative timestamps of this file segment in memory.

For illustration, let us consider the following simple example. Let a media server have a 100 MB memory, and the media files stored at the media server be $600\ sec(10\ min)$ long and encoded at 100 KB/s. Let us consider the following sequence of request arrivals as shown in Figure 3:

- request r_1 for a file f_1 arrives at time $t_1 = 0\ sec$;
- request r_2 for a file f_2 arrives at time $t_2 = 100\ sec$;
- request r_3 for a file f_3 arrives at time $t_3 = 200\ sec$.

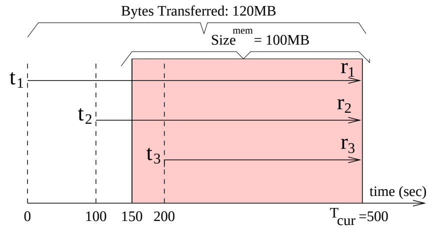

Figure 3: Simple example.

Let us evaluate the memory state at time point $T_{cur} = 500\ sec$. At this time point, request r_1 has transferred $500\ sec \times 100\ KB/s = 50\ MB$, request r_2 has transferred $400\ sec \times 100\ KB/s = 40\ MB$, and request r_3 has transferred $300\ sec \times 100\ KB/s = 30\ MB$. While the overall number of bytes transferred by three requests is 120 MB, the memory can hold only 100 MB of the latest (most recent) portions of transferred files which are represented by the following file segments:

- a segment of file f_1 between $150\ sec$ and $500\ sec$ of its duration. We use a denotation $<150, 500>(150)$ to describe this segment, where numbers in " $<>$" describe the beginning and the end of segment, and a number in "()" defines a relative timestamp in memory corresponding to the beginning of the segment.
- a segment of the file f_2: $<50, 400>(150)$;
- a segment of the file f_3: $<0, 300>(200)$.

This new abstraction provides a close approximation of file segments stored in memory and their relative time ordering (time stamps) in memory. This new memory representation can be used in determining whether a new media request will be served from memory or disk. For example, if a new request $r_{new}^{f_1}$ arrives at time $T_{cur} = 500\ sec$ it will be served from disk because the initial prefix of file f_1 is already evicted from memory. However, if a new request $r_{new}^{f_3}$ arrives at time $T_{cur} = 500\ sec$ it will be served

[2]To unify the measurement units between the memory size and the encoding bit rates of media files, we compute everything in bytes. In examples, while we use denotation $bitRate(r)$, the file encoding bit rates in computations are converted to bytes/sec.

from memory because the initial prefix of the corresponding file is present in memory.

If there are multiple concurrent accesses to the *same* file f then requests with a later arrival time might find the corresponding file segments already in memory (similar to the example described above). In the next section, we develop a set of basic operations to compute the *unique segments* of file f with the most recent timestamps which correspond to a sequence of accesses to f.

For each file f, we use the *time stamp ordering* of its segments. When the time ordering representation is used with respect to segments of all the files that are currently stored in memory it leads to a *segment-based memory model* (rather than a traditional block-based memory representation), which is actively used in *MediaGuard* framework.

4.2 Basic Operations for Computing File Segments in Memory

A file segment transferred by request r^f during time interval $[T,T']$ is defined as follows:

$$segm(r^f,T,T') = <x,y>(\hat{T}) \tag{2}$$

where
$$x = max\{T, t^{start}(r^f)\} - t^{start}(r^f),$$
$$y = min\{t^{end}(r^f), T'\} - t^{start}(r^f),$$
$$\hat{T} = max\{T, t^{start}(r^f)\}.$$

In computation of a current memory state, we need to be able to compute the **unique** file segments currently present in memory. This means that in case of multiple requests to the same file, we need to be able to identify the accesses and the corresponding file segments with the latest access time, and must avoid the repeatable counting of the same bytes accessed by different requests at different time points.

To explain this situation in more detail, let us consider the following example, graphically depicted in Figure 4. Let r_1^f, r_2^f, r_3^f be a sequence of requests accessing the same file f (with duration of 300 sec) in the following arrival order: $t^{start}(r_1^f) = 0$, $t^{start}(r_2^f) = 10$ *sec*, and $t^{start}(r_3^f) = 20$ *sec*.

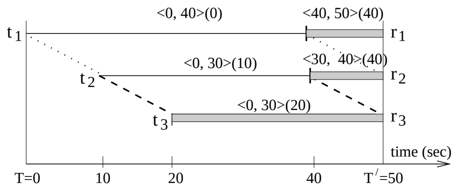

Figure 4: Multiple concurrent accesses to the same file.

While the first request r_1^f by the time $T' = 50$ *sec* had transferred segment $<0,50>(0)$, the initial part of this segment $<0,40>(0)$ was again accessed and transferred at a later time by the second request r_2^f. Thus segment $<40,50>(40)$ is the only unique segment of file f accessed by r_1^f most recently. Similarly, segment $<30,40>(40)$ represents the only unique segment of file f, which was accessed most recently by r_2^f. Finally, the latest request r_3^f is accountable for the most recent access to the initial segment $<0,30>(20)$ of file f. Thus overall, the unique segments of file f with the most recent timestamps in $[0,50]sec$ interval are the following:

$$segm(f,0,50) = \{<0,30>(20), <30,40>(40), <40,50>(40)\}$$

To determine the unique, most recent segments of file f accessed by subsequent requests $r_{i_1}^f$ and $r_{i_2}^f$ in $[T,T']$ time interval, we introduce a new operation, called *segment subtraction* and denoted as " \ ". Let $r_{i_1}^f$ and $r_{i_2}^f$ be two subsequent requests accessing the same file f such that $t^{start}(r_{i_1}^f) \le t^{start}(r_{i_2}^f)$, i.e. $r_{i_2}^f$ is a more recent access than $r_{i_1}^f$. Let $segm_{i_1} = segm(r_{i_1}^f, T, T') = <x_{i_1}, y_{i_1}>(T_{i_1})$ and $segm_{i_2} = segm(r_{i_2}^f, T, T') = <x_{i_2}, y_{i_2}>(T_{i_2})$. Then

$$segm_{i_1} \setminus segm_{i_2} = \begin{cases} <x_{i_1}, y_{i_1}>(T_{i_1}) & \text{if } y_{i_2} \le x_{i_1} \\ <y_{i_2}, y_{i_1}>(T'_{i_1}) & \text{otherwise} \end{cases} \tag{3}$$

where $T'_{i_1} = T_{i_1} + (y_{i_2} - x_{i_1})$.

Intuitively, the operation $(segm_{i_1} \setminus segm_{i_2})$ defines a part of the older segment $segm_{i_1}$ which does not coincide with any part of more recent segment $segm_{i_2}$. For illustration, let us consider the example depicted in Figure 4. Let us first compute the segments of file f accessed by r_1^f, r_2^f, r_3^f in $[0,50]$ *sec* interval:

$$segm_{i_1} = <0,50>(0)$$
$$segm_{i_2} = <0,40>(10)$$
$$segm_{i_3} = <0,30>(20).$$

Then,
$$segm_{i_1} \setminus segm_{i_2} = <40,50>(40)$$
$$segm_{i_2} \setminus segm_{i_3} = <30,40>(40)$$

Let $r_1^f, r_2^f, ..., r_n^f$ be a sequence of requests accessing the same file f during $[T,T']$ interval, where $t^{start}(r_1^f) \le t^{start}(r_2^f) \le ... \le t^{start}(r_n^f)$, i.e. r_1^f is the oldest access and r_n^f is the most recent access to file f in $[T,T']$ interval.

Our goal is to compute the unique segments of file f with the most recent timestamps which correspond to requests $r_1^f, r_2^f, ..., r_n^f$ during time interval $[T,T']$. The general formula to compute such file segments is defined in the following way:

$$segm(f,T,T') = segm(r_n^f,T,T') \cup \bigcup_{i=1}^{n-1} (segm(r_i^f,T,T') \setminus segm(r_{i+1}^f,T,T')) \tag{4}$$

Since function $segm(f,T,T')$ represents the unique segments of file f accessed in $[T,T']$ interval, we also can compute the total amount of unique bytes of file f accessed and stored in memory between $[T,T']$ interval and denoted as $UniqueBytes(f,T,T')$.

Let us again consider the example depicted in Figure 4. Now, let we need to compute the unique segments of file f in $[0,70]sec$ interval. This situation is represented in Figure 5.

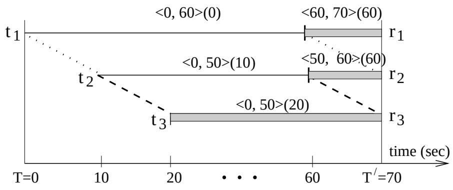

Figure 5: Example: the time advancement operation.

We can compute the segments by using Formula (4), but in practice, we use an optimized operation, called *segment shift by* Δt. This operation can be formally defined from Formulas (2) and (4). Here, we only give an intuition behind this operation. If there are already computed segments of file f in $[0,50]sec$ interval then it is easy to compute the updated segments of file f in $[0,70]sec$ interval: we need to advance time by $\Delta t = 20$ *sec* or *shift by* $\Delta t = 20$ *sec* the existing segments:

$$segm(f,0,50) = \{<0,30>(20), <30,40>(40), <40,50>(40)\}$$
$$segm(f,0,70) = \{<0,50>(20), <50,60>(60), <60,70>(60)\}$$

The outlined operation "shifts" in a uniform manner the segments that correspond to all the requests, except the latest, most recent one, by Δt:

$$<x,y>(\hat{T}) \text{ becomes } <x+\Delta t, y+\Delta t>(\hat{T}+\Delta t).$$

Let $<x, y>(\hat{T})$ be the latest (most recent) request. Then the *shift by* Δt operation leads to the following result:

$$<x, y>(\hat{T}) \text{ becomes } <x, y+\Delta t>(\hat{T}).$$

For the rest of the paper, when we use $segm(f, T, T')$ denotation we mean the set of segments of file f in time interval $[T, T']$ with *time stamp ordering* between the segments. This representation is a foundation of the *segment-based memory model*. Note, that when the *shift by* Δt operation is applied to the already ordered set of segments, only the segment that corresponds to the latest (most recent) request needs to be (potentially) reordered. The ordering between all the other segments is not impacted by the *shift* operation, and hence is preserved by this operation. Thus, the **complexity** of segments re-ordering after the *shift* operation is **linear** with respect to the number of requests for corresponding file f.

5. MODEL-BASED ADMISSION CONTROL

The main goal of an admission control is to prevent a media server from becoming overloaded. [3] The overload for a media server typically results in the violation of the real-time properties of the media application. The overloaded media server continues to serve all the accepted streams but the quality of service degrades: the packets for accepted streams are sent with a violation of "on-time delivery", and in such a way that the quality of the stream received by a client is compromised.

On a time scale, we distinguish two types of events, where the amount of available server resources needs to be reevaluated:

- *acceptance* of new requests;
- *termination* of currently accepted requests;

Multiple events can happen at the same time point. We define the following notations associated with such time points:

- *ActReqs(t)*: the set of requests that are currently in progress, i.e. "active" at time t;
- *TermReqs(t)*: the set of requests that are supposed to terminate at event time t;
- *Cap*: the absolute server capacity. (We set *Cap = 1* and the *cost function* for requests is derived using this setting as defined in Section 3.)
- *ACap(t)*: the available server capacity at time t. In order to allocate a specific share of server resources, say 50%, to a media service s, the available server capacity in the initial time T_{init} is preallocated to a designated share: i.e. $ACap_s(T_{init}) = 0.5 \times Cap$.

Let the current event time be T_{cur}. If there are events of both types, i.e. the *termination* of some already accepted requests and the *acceptance* of new requests, then the termination events are performed first (in order to release the corresponding resources), and the acceptance events are performed after that.

The *MediaGuard* admission controller (also denoted as *ac-MediaGuard*) performs the following actions depending on the event type:

1). Termination of currently accepted requests. In the time points corresponding to termination events, the following actions are performed:

- $ActReqs(T_{cur}) = ActReqs(T_{cur}) \setminus TermReqs(T_{cur})$.
- The server capacity is increased by the cost of the terminated requests:

$$ACap(T_{cur}) = ACap(T_{cur}) + \sum_{r \in TermReqs(T_{cur})} cost(r).$$

2). Acceptance of new request. To evaluate whether a new request r^f_{new} can be accepted at time T_{cur}, the *MediaGuard* admission controller performs the following two procedures:

- *Resource Availability Check*: during this procedure, the cost of a new request r^f_{new} is evaluated. To achieve this goal, the memory state of a media server at time T_{cur} is computed using the new segment-based memory model. From the memory state, we can identify whether a prefix of requested file f is residing in memory, and whether request r^f_{new} will have a cost of accessing memory or disk correspondingly. Then, *ac-MediaGuard* checks whether in the current time, the media server has enough available resources (capacity) to accommodate the resource requirements of new request r^f_{new}. In case of the positive outcome, *ac-MediaGuard* moves on to the QoS validation step.
- *QoS Guarantees Check:* during this procedure, the admission controller verifies that the acceptance of request r^f_{new} will not violate the QoS guarantees of already accepted requests at any point in the future, i.e. the media server will not enter an overloaded state at any point in the future. If the outcome of QoS validation process is positive then new request r^f_{new} is **accepted** and the following actions are performed:

  1) the available server capacity is decreased by the $cost(r^f_{new})$: $ACap(T_{cur}) = ACap(T_{cur}) - cost(r^f_{new})$.

  2) $ActReqs(T_{cur}) = ActReqs(T_{cur}) \cup r^f_{new}$.

  3) Let $T' = T_{cur} + duration(r^f_{new})$. Then $TermReqs(T') = TermReqs(T') \cup r^f_{new}$.

  Otherwise, request r^f_{new} is **rejected**.

This outlines the overall flow of *ac-MediaGuard* activities.

5.1 Computing Memory State and Estimating Cost of Request

The basic idea of computing the current memory state is as follows. Let $Size^{mem}$ be the size of memory [4] in bytes. Let $r_1(t_1), r_2(t_2), ..., r_k(t_k)$ be a recorded sequence of requests to a media server. Given the current time T, we need to compute some past time T^{mem} such that the sum of the bytes accessed by requests and stored in memory between T^{mem} and T is equal to $Size^{mem}$ as shown in Figure 6. This way, the files' segments streamed by the media server in $[T^{mem}, T]$ will be in memory.

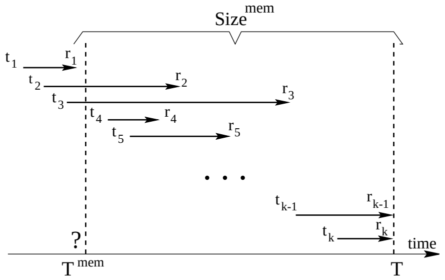

Figure 6: Memory state computation example.

[3] An alternative goal of admission controller can be stated as allocating the predefined share of media server capacity according to a *Service Level Agreement (SLA)*.

[4] Here, a memory size means an estimate of what the system may use for a file buffer cache.

To realize this idea in an efficient way, we design an induction-based algorithm for computing the memory state at any given time. Let T_{cur} be the current time corresponding to a new request r^f_{new} arrival, and the admission controller needs to decide whether to accept or reject request r^f_{new} for processing. Let T_{prev} denote the time of the previous arrival event, and let T^{mem}_{prev} be a previously computed time such that the sum of bytes accessed by requests and stored in memory between T^{mem}_{prev} and T_{prev} is equal to $Size^{mem}$ as shown in Figure 7.

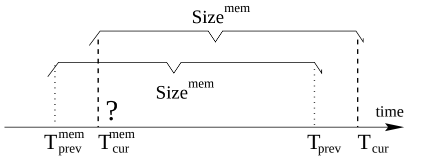

Figure 7: Induction-based memory state computation.

At time point T_{cur}, we need to compute

- an updated time T^{mem}_{cur} such that the sum of bytes stored in memory between T^{mem}_{cur} and T_{cur} is equal to $Size^{mem}$;
- an updated information about the memory state (i.e. file segments stored in memory) in order to determine the cost of new request r^f_{new}.

Let $Files(T_{prev}) = \{f_{i_1}, ..., f_{i_N}\}$ be a set of files accessed during $[T^{mem}_{prev}, T_{prev}]$ interval. The main data structure that we use to represent the unique file segments accessed during a given time interval $[T^{mem}_{prev}, T_{prev}]$ is a hash table, called $FileTable(T^{mem}_{prev}, T_{prev})$ and defined as follows:

f_{i_1} : $segm(f_{i_1}, T^{mem}_{prev}, T_{prev})$
...
f_{i_N} : $segm(f_{i_k}, T^{mem}_{prev}, T_{prev})$

By assumption, the sum of unique bytes accessed by client requests between the time points T^{mem}_{prev} and T_{prev} is equal to $Size^{mem}$. Hence the segments represented above are all stored in memory at time T_{prev}.

The file set $Files(T_{prev})$ consists of the files with active requests from small $ActReqs(T_{prev})$ (let N_a be the number of those files and n be the number of active requests) and the files with terminated requests (let N_t be the number of the corresponding files). The files in the latter subset are represented by a single continuous segment of the latest terminated request.

Let $\Delta t = T_{cur} - T_{prev}$.

For any file $f \in Files(T_{prev})$ with active requests, we perform the *shift by* Δt operation in order to advance time for active requests and to compute the unique file segments accessed during $[T^{mem}_{prev}, T_{cur}]$ interval. The **complexity** of this operation is **linear** with respect to the number of current requests in the system as it was discussed in Section 4.2.

Using $FileTable(T^{mem}_{prev}, T_{cur})$, we compute the total amount of unique bytes accessed during this time interval. The difference $\Delta B = UniqueBytes(T^{mem}_{prev}, T_{cur}) - Size^{mem}$ defines by "how far" time T^{mem}_{prev} should be advanced to a new time point T^{mem}_{cur}.

Since the segments in $FileTable(T^{mem}_{prev}, T_{cur})$ are ordered for each file entry, we can completely order all the segments in the $FileTable(T^{mem}_{prev}, T_{cur})$ in $\mathbf{O(N_t \log N_t) + O(n \log N_a)}$ **time**.

Using the completely ordered list of segments and the information about their file encoding bit rates, we compute time duration Δt during which the "oldest" segments will transfer the amount of bytes equal to ΔB. Then $T^{mem}_{cur} = T^{mem}_{prev} + \Delta t$. At the same time, the corresponding data structures are updated:

- $FileTable(T^{mem}_{cur}, T_{cur})$ contains only file segments starting at time T^{mem}_{cur} and later on;
- $Files(T_{cur})$ consists of only files with segments that were accessed at time T^{mem}_{cur} and later.

From $FileTable(T^{mem}_{cur}, T_{cur})$ that represents the current memory state, we can identify whether a prefix of the requested file f is residing in memory or not, and whether the request r^f_{new} will have a cost of access to memory or disk correspondingly. If the media server has enough currently available capacity to accommodate the resource requirements of new request r^f_{new} then request r^f_{new} is **conditionally accepted**. Otherwise, request r^f_{new} is **rejected**.

In summary, the **complexity** of computing the cost of a new request is **linear** with respect to the number of active requests in the system. In Section 6, we will provide an additional analysis of the number of files in memory (i.e. N_t and N_a) when processing a typical enterprise media workload.

5.2 QoS Validation Process

For long-lasting streaming media requests, an additional complexity consists in determining the level of available system resources as a function of time. When there is enough currently available server capacity to admit a new request r^f_{new}, the *MediaGuard* admission controller still needs to ensure that the acceptance of request r^f_{new} will not violate the QoS guarantees of already accepted requests over their lifetime and that the media server will not enter an overloaded state at any point in the future. [5]

- Let a new request r^f_{new} be a disk access. In this case, there is a continuous stream of new, additional bytes transferred from disk to memory (the amount of new bytes is defined by the file f encoding bit rate). It may result in replacement (eviction) of some "old" file segments in memory. For example, let some segments of file $\hat{f}$ be evicted. If there is an active request $r^{\hat{f}}$ which reads the corresponding file segments from memory (and has a cost of memory access) then once the corresponding segments of file $\hat{f}$ are evicted (replaced) from memory, the request $r^{\hat{f}}$ will read the corresponding segments of file $\hat{f}$ from disk with an increased cost of disk access. We will call that request $r^{\hat{f}}$ is *downgraded*, i.e. the acceptance of new request r^f_{new} will lead to an increased cost of request $r^{\hat{f}}$ in the future.
- Let a new request r^f_{new} be a memory access. Then we need to assess whether request r^f_{new} has the "memory" cost during its life or the corresponding segments of file f may be evicted in some future time points by already accepted active disk requests, and request t r^f_{new} will read the corresponding segments of file f from disk with the increased cost of disk access.

We need to assess such situations whenever they may occur in the future for accepted "memory" requests and evaluate whether the increased cost of downgraded requests can be offset by the *overall available capacity* of the server in the corresponding time points.

The main idea of our algorithm on QoS validation is as follows. We partition all the active requests in two groups:

- *active memory requests*, i.e. the requests which have a cost of memory access, and
- *active disk requests*, i.e. the requests which have a cost of disk access.

Active memory requests access their file segments in memory. Thus, they do not bring new bytes to memory, they only refresh the accessed file segments' time stamps with the current

[5] When *ac-MediaGuard* is used for resource allocation, the QoS validation stage ensures that the allocated share of server resources will not be exceeded over future time.

time. Only active disk requests bring "new" bytes from disk to memory and evict the corresponding amount of "oldest" bytes from memory. The *MediaGuard* admission controller identifies the bytes in memory with the oldest timestamp (let it be $T_{cur}^{act\_m}$) which are read by some of the active memory requests. Thus, all the bytes stored in memory prior to $T_{cur}^{act\_m}$ can be safely replaced in memory (as depicted in Figure 8) without impacting any active memory requests.

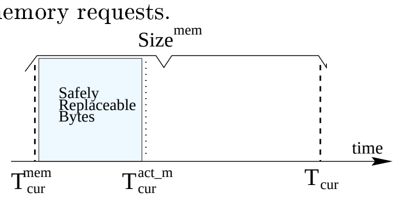

Figure 8: Safely Replaceable Bytes in Memory.

Using the information about file encoding bit rates as well as the future termination times for active disk requests we compute a time duration during which the active disk requests will either transfer from disk to memory the amount of bytes equal to $SafelyReplBytes(T_{cur})$ or all of them will terminate. In order to make the QoS validation process terminate within a limited number of steps, we attempt to advance the clock at each step beyond the designated *ClockAdvanceTime*. In the simulation model that we built for performance evaluation of *ac-MediaGuard* and in the results reported in Section 6, we set *ClockAdvanceTime = 1 sec.*

By repeating this process in the corresponding future points, we identify whether active disk requests are always evicting only "safely replaceable bytes" in memory or some of the active memory requests have to be downgraded. In latter case, *ac-MediaGuard* evaluates whether the increased cost of downgraded requests can be offset by the available server capacity at these time points.

In summary, the **complexity** of QoS validation procedure is **linear** with respect to the number of active requests in the system: the QoS validation procedure is guaranteed to terminate in a fixed number of steps, where at each step, the computation of an updated memory state is performed in a linear time. In Section 6, we will provide an additional analysis of the number of steps (iterations over future time points) in QoS validation procedure when processing a typical enterprise media workload.

6. PERFORMANCE EVALUATION

In order to evaluate the performance benefits of a new model-based and memory-aware strategy *ac-MediaGuard*, we built a simulation model. We compare *ac-MediaGuard* performance against the *disk-based* admission control policy that pessimistically assumes that all accesses must go to disk.

For workload generation, we use the publicly available, synthetic media workload generator *MediSyn* [24]. We performed a sensitivity study using two workloads $W1$ and $W2$ closely imitating parameters of real enterprise media server workloads [5]. The overall statistics for workloads used in the study, are summarized in Table 1:

	$W1$	$W2$
Number of Files	800	800
Zipf α	1.34	1.22
Storage Requirement	41 GB	41 GB
Number of Requests	41,703	24,159

Table 1: Workload parameters used in simulation study.

Both synthetic workloads have the same media file duration distribution, which can be briefly summarized via following six classes: 20% of the files represent short videos 0-2min, 10% of the videos are 2-5min, 13% of the videos are 5-10min, 23% are 10-30min, 21% are 30-60min, and 13% of the videos are longer than 60 min. This distribution represent a media file duration mix that is typical for enterprise media workloads [5], where along with the short and medium videos (demos, news, and promotional materials) there is a representative set of long videos (training materials, lectures, and business events).

The file bit rates are defined by the following discrete distribution: 5% of the files are encoded at 56Kb/s, 20% - at 112Kb/s, 50% - at 256Kb/s, 20% - at 350Kb/s, and 5% - at 500Kb/s.

Request arrivals are modeled by a Poisson process with arrival rate of 1 req/sec. This rate kept the media server under a consistent overload.

The file popularity is defined by a Zipf-like distribution with α shown in Table 1: $\alpha = 1.34$ for workload $W1$, and $\alpha = 1.22$ for workload $W2$. In summary, $W1$ has a higher locality of references than $W2$: 90% of the requests target 10% of the files in $W1$ compared to 90% of the requests targeting 20% of the files in $W2$. Correspondingly, 10% of the most popular files in $W1$ have an overall combined size of 3.8 GB, while 20% of the most popular files in $W2$ use 7.5 GB of the storage space.

We performed a set of simulations for a media server with different memory sizes of 0.5 GB, 1 GB, 2 GB, and 4 GB. While 4 GB might be an unrealistic parameter for file buffer cache size, we are interested to see the dependence of a performance gain due to increased memory size. We define the server capacity and the cost functions similar to those measured using the experimental testbed described in Section 3. We use $cost_{X_i}^{disk}/cost_{X_i}^{memory} = 3$, i.e. the cost of disk access is 3 times higher than the cost of the corresponding memory access.

The first set of performance results for both workloads is shown in Figures 9 a), b). They represent the normalized throughput improvements under *ac-MediaGuard* compared to the *disk-based* admission control strategy using two metrics: the number of accepted requests and the total number of transferred bytes. The *ac-MediaGuard* policy significantly outperforms the *disk-based* strategy for both workloads. For instance, for workload $W1$ and file buffer cache of 2 GB, *ac-MediaGuard* shows a factor of two improvement in throughput for both metrics. It reveals that the media server performance can be significantly improved via main memory support even for media server with relatively small size of file buffer cache. [6] The memory increase does not result in a "linear" performance gain as shown in Figures 9 a), b). The memory increase from 2 GB to 4 GB results in less than 10% of additional performance gain for both workloads.

Despite the fact that $W2$ has much less reference locality and its popular files occupy twice as much space compared to $W1$, the performance improvements under *ac-MediaGuard* for $W2$ are only slightly lower than for $W1$.

Figures 10 a), b) show the overall hit and byte hit ratio for the requests accepted by the media server and served from memory for workloads $W1$ and $W2$ correspondingly. Even for a relatively small file buffer cache (such as 0.5 GB), 79% of the sessions for $W1$ workload and 73% of the sessions for $W2$ workload are served from memory. These sessions are responsible for 55% of bytes for $W1$ workload and 50% of bytes for $W2$ workload transferred by the media server. For a file buffer cache of 2 GB, the file hit ratio increases up to 90% for $W1$ workload and 85% for $W2$ workload, that result in 79% of bytes for $W1$ workload and 74% of bytes for $W2$ workload transferred by the media

[6] In practice, a service provider may use a conservative estimate for a file buffer cache size, while still obtaining a significant performance gain.

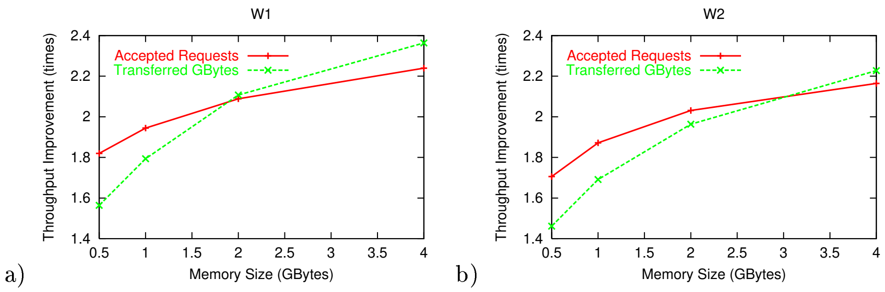

Figure 9: Normalized throughput improvements under *ac-MediaGuard* compared to the *disk-based* admission control strategy: a) $W1$ and b) $W2$.

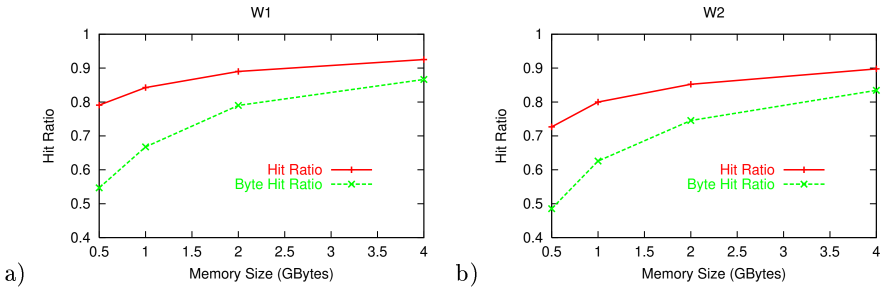

Figure 10: Hit and byte hit ratios under *ac-MediaGuard*: a) $W1$ and b) $W2$.

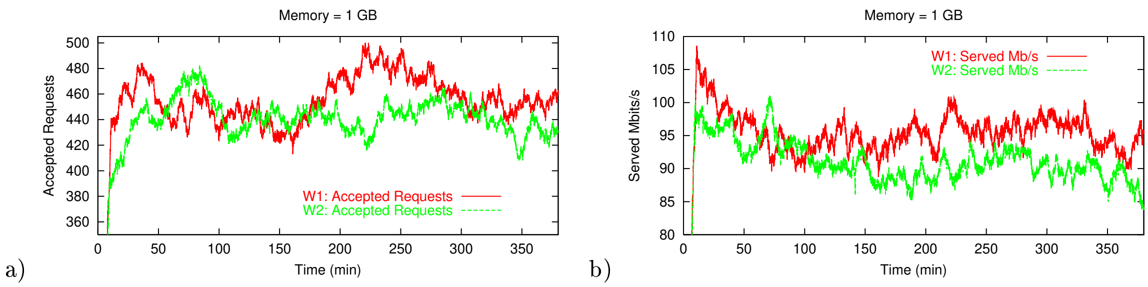

Figure 11: Media server throughput for $W1$ and $W2$ under *ac-MediaGuard* strategy over time: a) the number of accepted sessions over time; b) the number of Mbits/s transferred over time.

server.

Figure 11 a) demonstrates the number of processed clients sessions for both workloads $W1$ and $W2$ over time. While both workloads are utilizing the same server capacity, we can see that the number of accepted and processed sessions over time is far from being fixed: it varies within 25% for each of the considered workload. It can be explained by two reasons: *i)* first, the different requests may have a different *cost* in terms of required media server resources (just compare the bandwidth requirements of 56 Kb/s stream and 256 Kb/s stream); *ii)* second, most of the accesses to popular media files can be served from memory, even when a media server relies on traditional file system and memory support and does not have additional application level caching. Thus, the locality available in typical media workload has a significant impact on the behavior of the system because serving content from memory incurs much lower overhead than serving the same content from disk. Figure 11 b) demonstrates the maximum bandwidth delivered by the media server (in Mbits/s) over time for both workloads $W1$ and $W2$ correspondingly. Similarly, it is variable for each workload, because of the varying number of accepted clients requests as well as a broad variety of encoding bit rates for corresponding media content.

Now, we present some statistics related to the *ac-MediaGuard* algorithm performance. The designed *ac-MediaGuard* algorithm performs two main procedures when evaluating whether a new request can be accepted. At a first stage, it estimates the cost of a new request via computing an updated memory state that corresponds to a current time. The computation of an updated memory state involves recalculating the file segments in memory for all the currently active requests. We showed that the complexity of the algorithm is linear with respect to the number of active requests. The constant in the algorithm depends on the number of files that have their segments stored in memory: let us call these files as a *memory file set*. Tables 2 and 3 show the memory file set profile (averaged over time) for both workloads $W1$ and $W2$ correspondingly.

Memory Size	*Number of files in memory*					
	Overall	Terminated	Single access	2-5 accesses	6-10 accesses	$\geq$ 10 accesses
0.5 GB	93	4	50	21	6	10
1 GB	101	8	50	23	6	13
2 GB	117	18	50	27	6	16
4 GB	153	38	45	41	9	21

Table 2: Workload $W1$: a profile of a memory file set.

Memory Size	*Number of files in memory*					
	Overall	Terminated	Single access	2-5 accesses	6-10 accesses	$\geq$ 10 accesses
0.5 GB	105	5	60	24	6	10
1 GB	112	9	60	25	7	12
2 GB	130	19	58	30	7	16
4 GB	165	38	54	44	9	20

Table 3: Workload $W2$: a profile of a memory file set.

The analysis shows that while each workload has overall 800 files, only 12%-20% of them (93 to 165 files) are in memory at the same time. Clearly, a larger size memory holds a higher number of files. The further profile of those files is interesting. Typically, a small size memory has a very few files with seg-

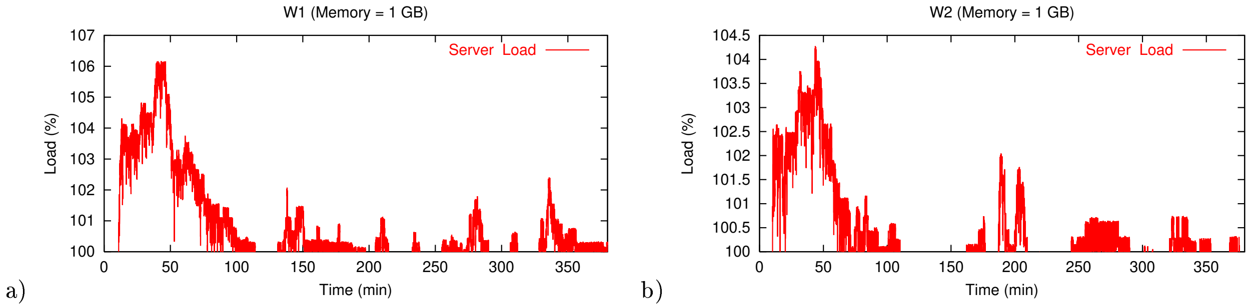

Figure 12: A modified *ac-MediaGuard* without the QoS validation procedure: a server load over time for a) $W1$ workload, $MemSize = 1\ GB$; b) $W2$ workload, $MemSize = 1\ GB$.

ments which correspond to the terminated requests: these files are evicted from memory very fast. However, larger size memory has a higher number of files corresponding to terminated requests: number of these files doubles with the corresponding memory size increase. Additionally, there is a steady percentage of files with a single access: these files are represented in memory by a single continuous segment. Finally, only a small number of files (4% to 9%) has multiple outstanding requests. The computation of the current memory state involves recalculating the file segments of exactly those files: they account for only 37 to 71 files (on average) in our workloads. While the computation of the current memory state is linear with respect to the number of active requests in the system, the number of files in the memory file set determines the constant of proportionality in the computation time. For typical media workloads and current media servers, this constant is small, and hence the *ac-MediaGuard* implementation can be very efficient.

The second procedure, performed by *ac-MediaGuard* algorithm, computes the level of available system resources as a function of time to provide QoS guarantees for accepted client requests. First of all, how important is this step? To answer this question, we have performed the simulations with a modified version of *ac-MediaGuard* that admits client requests based on the resource availability at the time of request arrival (i.e. it does not perform the QoS validation procedure over future time). Figure 12 a), b) shows the server load over time when the media server is running a modified admission controller without the QoS validation procedure.

The simulation results confirm that when the admission decision is based only on the resource availability at the time of request arrival, it may lead to a server overload in the future. In our simulations, we can observe a server overload of 4%-6% over almost one hour time period. The explanation of why the server overload is more pronounced in the beginning of our simulations is due to the fact that in the beginning, memory has not yet reached a "steady" state and a few high bit rate disk requests accepted on a basis of current resource availability may cause a downgrade of a significant number of already accepted memory requests. Thus, the QoS validation procedure may be especially important for shared media service design, where different media services might have different workload characteristics, which might severe interfere with each other (especially over time) in resource consumptions of a shared file buffer cache and main memory.

Finally, Figure 13 shows the *CDF* of the number of steps (the number of iterations over the future time points) in the QoS validation procedure for $W1$ workload (results for $W2$ workload are very similar). For 90% of all the requests (both accepted and rejected), the QoS validation procedure will terminate after the 2 steps, implying a very efficient computation time. Since

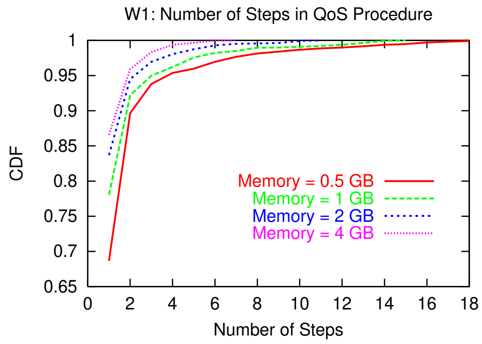

Figure 13: $W1$: CDF of number of steps in QoS procedure.

for long-lasting media requests, it is important to guarantee the allocation of sufficient server resources over time, we view the QoS validation procedure as an important integral component in *MediaGuard* framework.

7. CONCLUSIONS AND FUTURE WORK

In this paper, we present *MediaGuard* – a model-based infrastructure for building QoS-aware streaming media services. Using the following techniques, *MediaGuard*can efficiently determine the fraction of server resources required to support a particular client request over its expected lifetime:

- A unified approach for measuring media server capacity via a set of basic benchmarks to accurately interpolate media server capacity for processing realistic workloads. Using a set of basic benchmark measurements, we derive the *cost* function that uses a single value to reflect the combined resource requirements (e.g., CPU, disk, memory, and bandwidth) necessary to support a particular media stream.
- A novel, segment-based memory model of the media server that provides a close approximation of the operating system's memory occupancy at any point in time based on dynamically changing workload characteristics and an understanding of the operating system's LRU-based page replacement policy. We show that this can be done with no assumptions about the operating system's scheduling policy.

Using the *MediaGuard* framework, we designed an admission control infrastructure for a streaming media service that accounts for the impact of the server's main memory file buffer cache, and as a corrolary is managing a server resources in a more efficient way. A performance comparison of the proposed new strategy relative to a disk-based policy reveals a factor of two improvement in throughput.

While our design and evaluation is limited to a particular set of workloads for a streaming media service, we believe that our approach is general to a variety of infrastructures, including shared hosting environments. In general, it is difficult to determine the level of available system resources as a function of time. For long-lasting real-time requests, it is insufficient to simply consider the current level of, for instance, available memory or idle CPU. Rather, complex interactions with memory caching and with the resource requirements of the requests that are currently processed by a server must be considered in evaluating whether a particular request can be satisfied with particular QoS characteristics for the request duration. Beyond real-time services, consider a request to a complex multi-stage Internet service that must consult a web application server, a database system, and secondary storage before it returns its results to the end user. Once again, the current state of main memory caches, among other considerations, can greatly impact the performance of the request.

Delivering performance isolation to competing media services while at the same time leveraging available multiplexing via a shared system memory is an interesting problem that can be addressed by the type of memory and media server models that we have developed in this work.

MediaGuard can be used to improve resource planning for dynamically changing application workloads in the Utility Data Center [29] environment. For example, a service provider may set two thresholds for server capacity: low - 70% and high - 95%. *MediaGuard* can then perform the double task of: *i)* admitting new requests only when the server capacity is below 95%, and *ii)* collecting a set of alarms when the server capacity crosses the 70% threshold. These alarms may be used by a service administrator to determine when to deploy additional server resources to accommodate growing user demand or changing access characteristics.

Another interesting direction for future work is the design of statistical models of media server capacity where workload properties such as file frequency, sharing patterns, burstiness, etc., can be accounted for estimating disk and memory usage for a given workload. The *MediaGuard* model can be used in design of the efficient workload-aware load balancing and request routing policies in a media server cluster.

8. REFERENCES

[1] G. Banga, P. Druschel, J. Mogul. Resource containers: A new facility for resource management in server systems. In the Proceedings of the Third Symposium on Operating System Design and Implementation (OSDI), February 1999.

[2] E. Biersack, F. Thiesse. Statistical Admission Control in Video Servers with Constant Data Length Retrieval of VBR Streams. Proc. of the 3d Intl. Conference on Multimedia Modeling, France, 1996.

[3] E. Chang, A. Zakhor. Cost Analyses for VBR Video Servers. Proc. of IST/SPIE Multimedia Computing and Networking, San Jose, 1996.

[4] J. Chase, D. Anderson, P. Thakar, A. Vahdat, and R. Doyle. Managing Energy and Server Resources in Hosting Centers. Proc. of the 18-th ACM Symposium on Operating Systems Principles (SOSP), October, 2001.

[5] L. Cherkasova, M. Gupta. Characterizing Locality, Evolution, and Life Span of Accesses in Enterprise Media Server Workloads. Proc. of ACM NOSSDAV, 2002.

[6] J. Dengler, C. Bernhardt, E. Biersack. Deterministic Admission Control Strategies in Video Servers with Variable Bit Rate. Proc. of European Workshop on Interactive Distributed Multimedia Systems and Services (IDMS), Germany, 1996.

[7] A. Dan, D. Dias R. Mukherjee, D. Sitaram, R. Tewari. Buffering and Caching in Large-Scale Video Servers. Proc. of COMPCON, 1995.

[8] A. Dan, D. Sitaram. A Generalized Interval Caching Policy for Mixed Interactive and Long Video Workloads . Proc. of IST/SPIE Multimedia Computing and Networking, 1996.

[9] R. Doyle, J. Chase, O. Asad, W. Jen, A. Vahdat. Model-Based Resource Provisioning in a Web Service Utility. Proc. of the USENIX Symposium on Internet Technologies and Systems (USITS), 2003.

[10] Y. Fu, A. Vahdat. SLA-Based Distributed Resource Allocation for Streaming Hosting Systems. Proc. of the 7th Intl Workshop on Web Content Caching and Distribution (WCW-7), 2002.

[11] J. Gemmell, S. Christodoulakis. Principles of Delay Sensitive Multimedia Data Storage and Retrieval. ACM Transactions on Information Systems, 10(1), 1992.

[12] X. Jiang, P. Mohapatra. Efficient admission control algorithms for multimedia servers. J. ACM Multimedia Systems, vol. 7(4), July, 1999.

[13] M. Kamath, K. Ramamritham, D. Towsley. Continuous Media Sharing in Multimedia Database Systems. Proc. of the 4th Intl. Conference on Database Systems for Advanced Applications, 1995.

[14] E. Knightly, H. Zhang. Traffic Characterization and Switch Utilization Using Deterministic Bounding Interval Dependent Traffic Models. Proc. of IEEE INFOCOM'95, Boston 1995.

[15] E. Knightly, D. Wrege, J. Lieberherr, H. Zhang. Fundamental Limits and Tradeoffs of Providing Deterministic Guarantees to VBR Video Traffic. Proc. of ACM SIGMETRICS'95.

[16] D. Makaroff, R. Ng. Schemes For Implementing Buffer Sharing in Continuous-Media Systems. Information Systems, Vol. 20, No. 6. 1995.

[17] D. Makaroff, G.Neufeld, N. Hutchinson. An Evaluation of VBR Disk Admission Algorithms for Continuous Media File Servers. Proc. of ACM Multimedia'97, Seattle, 1997.

[18] D. Makaroff, G.Neufeld, N. Hutchinson. Network Bandwidth Allocation and Admission Control for a Continuous Media File Server. Proc. of the 6th Intl. Workshop on Interactive Distributed Multimedia Systems and Telecommunications Services, Toulouse, France, 1999.

[19] G. Neufeld, D. Makaroff, N. Hutchinson. Design of a Variable Bit Rate Continuous Media File Server for an ATM Network. Proc. of IST/SPIE Multimedia Computing and Networking, 1996.

[20] B. Ozden, R. Rastogi, A. Silberschatz. Buffer Replacement Algorithms for Multimedia Databases. In S. M. Chung, editor, Multimedia Information Storage and Management, chapter 7. Kluwer Academic Publishers, 1996.

[21] S. Ranjan, J. Rolia, H. Fu, E. Knightly. QoS-Driven Server Migration for Internet Data Centers. Proc. of ACM/IEEE Int'l Workshop on Quality of Service (IWQoS), 2002.

[22] N. Reddy, J. Willie. Disk Scheduling in Multimedia I/O System. Proc. of ACM Multimedia, Anaheim, 1993.

[23] D. Rotem, J. Zhao. Buffer management for Video Database Systems. Proc. of Intl. Conf. on Database Engineering, 1995.

[24] W. Tang, Y. Fu, L. Cherkasova, A. Vahdat. MediSyn: A Synthetic Streaming Media Service Workload Generator. Proc. of ACM NOSSDAV, 2003.

[25] F. Tobagi, J. Pang, R. Baird, M.Gang. Streaming RAID: A Disk Storage for Video and Audio Files. Proc. of ACM Multimedia, Anaheim, 1993.

[26] B. Urgaonkar, P. Shenoy, T. Roscoe. Resource Overbooking and Application Profiling in Shared Hosting Platforms. Proc. of 5th Symposium on Operating Systems Design and Implementation (OSDI), 2002

[27] H. Vin, P. Goyal, A. Goyal, A. Goyal. A Statistical Admission Control Algorithm for Multimedia Servers. In Proc. of ACM Multimedia, San Francisco, 1994.

[28] H. Vin, P. Rangan. Designing a Multi-User HDTV Storage Server. IEEE J. on Selected Areas in Communications, 11(1), Jan., 1993.

[29] Utility Data Center: Solutions. http://www.hp.com/solutions1/infrastructure/solutions/utilitydata/index.html

[30] Management solutions across the utility computing continuum. Strategic White Paper. http://www.hp.com/large/infrastructure/management/strategic_wp_062102.pdf

[31] P. Yu, M. Chen, D. Kandlur. Design and Analysis of a Grouped Sweeping Scheme for Mutimedia Storage Management. Proc. of ACM NOSSDAV, San Diego, 1992.