MediaGuard: a model-based framework for building streaming media services

Ludmila Cherkasova, Wenting Tang
Hewlett-Packard Laboratories
1501 Page Mill Road, Palo Alto, CA 94303, USA
{lucy.cherkasova, wenting.tang}@hp.com

Amin Vahdat
Dept of Computer Science and Engineering,
University of California, San Diego,
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. We present MediaGuard *– a model-based infrastructure for building 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 in liner time 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.*

Keywords: Streaming media servers,benchmarking,modeling,admission control,system performance,analysis.

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.[3] 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. 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[3] 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 by default 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 4.1 introduces the segment-based memory model. Section 4 further presents the main components of *MediaGuard* framework and develops the *MediaGuard* admission controller. We present the performance evaluation results in Section 5. Finally, we conclude with a summary and directions for future work in Section 6.

2. RELATED WORK

Continuous media file servers require that several system resources be reserved in order to guarantee timely delivery of the data to clients. As a part of QoS-aware resource management for distributed multimedia applications[17] the resource broker performs admission control using client QoS profile in which the amount of required resources is specified. Such a profile either is prebuilt off-line or created in on-line manner using testing service.[16] Our work proceeds in a similar direction and proposes a unified framework for measuring performance and allocating resources of *commercial media servers*. The performance model designed in the paper allows one to estimate the necessary system resources required for support of a commodity media service without QoS degradation.

Commercial media systems may contain hundreds to thousands of clients. 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.[20] Most of the papers are devoted to storage and retrieval of variable bit rate data. *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 algorithms are characterized by the worst case assumptions regarding the service time from disk.[7, 11, 19] *Statistical* ac-algorithms are designed[2, 12, 20] to provide probabilistic QoS guarantees instead of deterministic ones, resulting in higher resource utilization due to statistical multiplexing gain. In papers,[10, 14, 15] the admission control is examined from the point of *network bandwidth allocation* on the server side.

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. 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. There have been several studies on *buffer sharing* and *interval caching* techniques to overcome the disk bandwidth restriction.[8, 9]

In our work, we design a high-level model of a traditional memory system with LRU replacement strategy as used in today's **commodity systems** to reflect, quantify, and take advantage 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. In paper,[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 paper,[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 media workloads, the different system resources may limit a media server performance. To solve this problem we propose corresponding models for an efficient streaming media utility design.

3. MEASURING MEDIA SERVER CAPACITY

In this section, we provide a background on current state of measuring media server capacity and our approach to the problem.

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.[13] In paper,[4] two basic benchmarks were introduced that can establish 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, and
- *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 performance 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. Clearly, the set of benchmarked encoding bit rates can be customized according to targeted workload profile). Using an experimental testbed and a proposed set of basic benchmarks, we measured capacity and scaling rules of a media server running 3RealServer 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.

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 rate of media streams used in our experiments is 18 times. However, the difference in maximum number of concurrent streams a server is capable of supporting for corresponding bit rates is only around 9 times for a *Single File Benchmark*, and 10 times for a *Unique Files Benchmark*. The media server performance is 3 times

higher (for some disk/file subsystem up to 7 times higher) under the *Single File Benchmark* than under the *Unique Files Benchmark*. This quantifies the performance benefits for multimedia applications when media streams are delivered from memory.

Using a set of basic benchmark measurements, we derive a *cost* function which defines a *fraction* of system resources needed to support a particular media stream depending on the stream bit rate and type of access (memory file access or disk file access):

- $cost_{X_i}^{disk}$ - a value of cost function for a stream with disk access to a file encoded at X_i Kb/s. If we define the media server capacity being equal to 1, the cost function is computed as $cost_{X_i}^{disk} = 1/N_{X_i}^{Unique}$, where $N_{X_i}^{unique}$ - the maximum measured server capacity in concurrent streams under the *Unique File Benchmark* for X_i Kb/s encoding,
- $cost_{X_i}^{memory}$ - a value of cost function for a stream with memory access to a file encoded at X_i Kb/s. Let $N_{X_i}^{single}$ - the maximum measured server capacity in concurrent streams under the *Single File Benchmark* for a file encoded at X_i Kb/s. Then the cost function is computed as $cost_{X_i}^{memory} = (N_{X_i}^{unique} - 1)/(N_{X_i}^{unique} \times (N_{X_i}^{single} - 1))$.

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

- $X_w = X_1, ...X_{k_w}$ - a set of distinct encoding bit rates of the files appearing in W,
- $N_{X_{w_i}}^{memory}$- a number of streams having a memory access type for a subset of files encoded at X_{w_i} Kb/s,
- $N_{X_{w_i}}^{disk}$ - a number of streams having a disk access type for a subset of files encoded at X_{w_i} Kb/s.

Then the media service demand under workload W can be computed by the following capacity equation

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

If $Demand \leq 1$ then the media server operates within its capacity, and the difference $1 - Demand$ defines the amount of available server capacity. We validated this performance model[5] 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. MODEL-BASED ADMISSION CONTROL

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. 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. Thus, the main goal of the designed admission control mechanism, called *ac-MediaGuard*, is to keep the allocated fraction of a media server resources to the corresponding service s below or equal to its specified threshold X_s%. *ac-MediaGuard* performs two main procedures when evaluating whether a new request r_{new}^f to media service s can be accepted:

- *Resource Availability Check*: during this procedure, a cost of a new request r_{new}^f is evaluated. To achieve this goal we evaluate the memory (file buffer cache) state to identify whether a prefix of requested file f is residing in memory, and whether request r_{new}^f will have a cost of accessing memory or disk correspondingly. Then, *ac-MediaGuard* checks whether in the current time, the media service s has enough available capacity to accommodate the resource requirements of new request r_{new}^f.

- *QoS Guarantees Check:* When there is enough currently available server capacity to admit a new request r^f_{new} the *ac-MediaGuard* mechanism still needs to ensure that the acceptance of request r^f_{new} will not violate the QoS guarantees of already accepted requests in the system over their lifetime and that the media server will not enter an overloaded state at any point in the future.

4.1. Resource Availability Check Using a Novel 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.

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 * which continue for a designated file duration: i.e. once a request is accepted, it will proceed until the end: $duration(r) = t^{end}(r) - t^{start}(r)$.

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 † 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 1:

- 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 $t_3 = 200\ sec$.

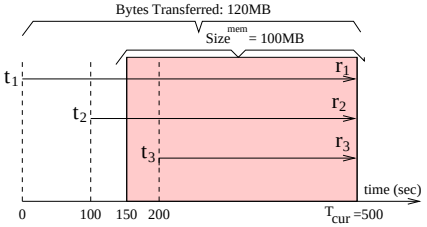


Figure 1. Simple example.

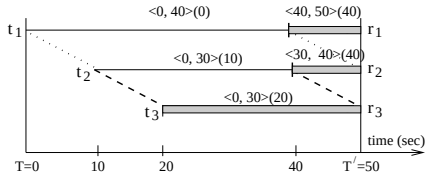


Figure 2. Multiple concurrent accesses to the same file.

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

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

†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.

$300\ sec \times 100\ KB/s = 30\ MB$. 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 from memory because the initial prefix of the corresponding file is present in memory.

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 2. 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 $t^{start}(r_3^f) = 20\ sec$.

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:

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

We developed 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 in $[T,T']$ time interval. Due to the space limitation, we only explain the intuition behind these operations, more formal definitions can be found in our HPL report.[6]

For each file f, we use *time stamp ordering* of its segments. When the time ordering 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 our media-QoS framework.

The basic idea of computing the current memory state is as follows. Let $Size^{mem}$ be the size of memory [‡] 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 3. This way, the files' segments streamed by the media server in $[T^{mem}, T]$ will be in memory.

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_{new}^f arrival, and the admission controller needs to decide whether to accept or reject request r_{new}^f for processing. Let T_{prev} denote the time of the previous arrival event, and let T_{prev}^{mem} be a previously computed time such that the sum of bytes accessed by requests and stored in memory between T_{prev}^{mem} and T_{prev} is equal to $Size^{mem}$ as shown in Figure 4.

At time point T_{cur}, we compute

- an updated time T_{cur}^{mem} such that the sum of bytes stored in memory between T_{cur}^{mem} and T_{cur} is equal to $Size^{mem}$;

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

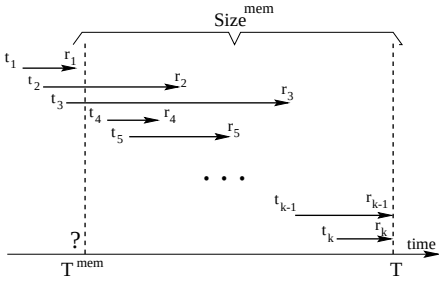


Figure 3. Memory state computation example.

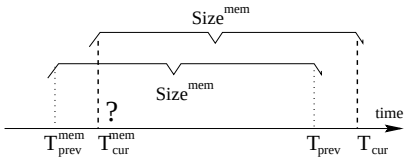


Figure 4. Induction-based memory state computation.

- 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}.

The important feature of the designed algorithm is that the **complexity** of computing the cost of a new request is **linear** with respect to the number of active requests in the system (see for details[6]). The constant in the algorithm depends on the number of overall files that have their segments stored in memory. In order to show the computational efficiency of the proposed approach, we will provide an additional analysis of the number of files simultaneously present in memory when processing a typical enterprise media workload in Section 5.

4.2. 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 *ac-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. [§]

- 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 time. Only active disk requests bring "new" bytes from disk to memory and evict the corresponding amount of "oldest" bytes from memory. The *ac-MediaGuard* admission controller identifies the bytes in memory with the oldest timestamp (let it be $T^{act\_m}_{cur}$) which are read by some of the active memory requests. Thus, all the bytes stored in memory prior to $T^{act\_m}_{cur}$ can be safely replaced in memory (as depicted in Figure 5) without impacting any active memory requests.

[§]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.

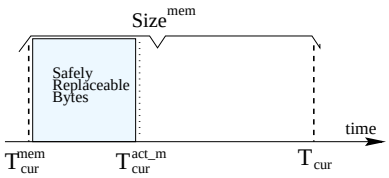


Figure 5. Safely Replaceable Bytes in Memory.

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 advance the clock at each step at least by the designated parameter *ClockAdvanceTime* (we use *ClockAdvanceTime = 1 sec* in our simulation model presented in Section 5).

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.

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 5, 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.

5. PERFORMANCE EVALUATION

In this section, using a simulation model, we evaluate the efficiency of the admission control strategy *ac-MediaGuard* designed using our QoS-aware framework.

5.1. Workload Used in Simulation Study.

For workload generation, we use the publicly available, synthetic media workload generator *MediSyn*.[18] For the sensitivity study, we use two workloads $W1$ and $W2$ closely imitating parameters of real enterprise media server workloads.[3] 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,[3] 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.

5.2. Media Server Used in Simulation Study

In the design process, we assume that a given media server configuration is benchmarked using the set of basic benchmarks described in Section 3. In our simulations, we define the server capacity and the cost functions for memory accesses similar to those measured in Section 3, while the cost of disk access is 3 times higher than the cost of the corresponding memory access, i.e. $cost_{X_i}^{disk}/cost_{X_i}^{memory} = 3$ (that is again similar to those measured in Section 3).

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.

5.3. Performance Results

We compare *ac-MediaGuard* performance against the *disk-based* admission control policy that by default, assumes that all the accesses are served from disk.

The first set of performance results for both workloads is shown in Figures 6 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 admission controller performance can be significantly improved when taking into account the impact of main memory support even for media server with relatively small size of file buffer cache. ¶ The server memory increase does not result in a "linear" performance gain as shown in Figures 6 a), b). The memory increase from 2 GB to 4 GB results in less than 10% of additional performance gain for both workloads.

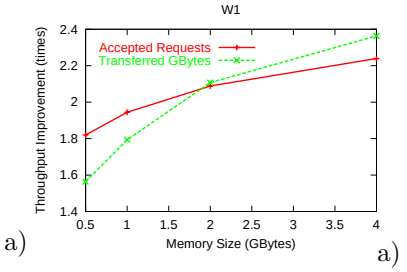


Figure 6. Throughput improvements under *ac-MediaGuard* compared to the *disk-based* admission control: a) $W1$ and b) $W2$.

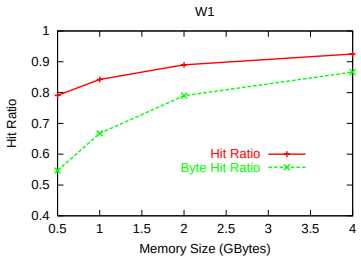


Figure 7. Hit and byte hit ratios under *ac-MediaGuard*: a) $W1$ and b) $W2$.

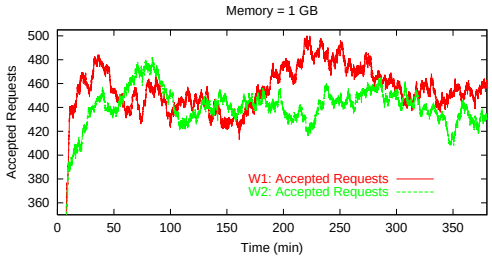


Figure 8. Media server throughput for $W1$ and $W2$ under *ac-MediaGuard* strategy over time: a) the number of accepted sessions over time; b) the number of Mbits/s transferred over time.

Despite the fact that $W2$ has 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 7 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

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

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 server.

Figure 8 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 8 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. 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.

	Number of files in memory					
Memory Size	Overall	Termi-nated	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

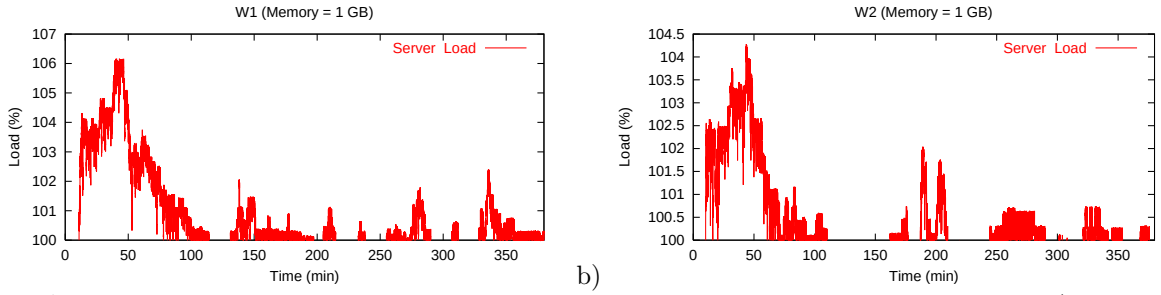
Table 2. Workload $W1$: a profile of a memory file set.

	Number of files in memory					
Memory Size	Overall	Termi-nated	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 segments 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 9 a), b) shows the server load over time when the media server is running a modified admission controller without the QoS validation procedure.



a) b)

Figure 9. A modified *ac-MediaGuard* without the QoS validation procedure: a server load over time for a) $W1$ workload, $MemSize = 1\ GB$; b) $W2$ workload, $MemSize = 1\ GB$.

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.

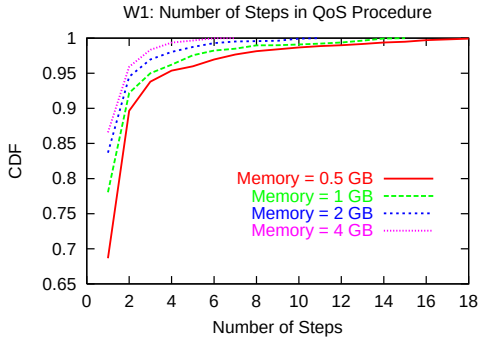


Figure 10. $W1$: CDF of number of steps in QoS procedure.

Finally, Figure 10 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 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 QoS-aware framework.

6. CONCLUSION AND FUTURE WORK

In this paper, we present a model-based framework for building QoS-aware services running on the top of commodity media servers. The core components of our framework are:

- Measuring media server capacity via a set of basic benchmarks that enable an accurate interpolation of the 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. Significantly, our model makes no assumptions about the operating system's scheduling policy.

- The memory model and the *cost* function form the basis of the infrastructure that can efficiently determine in liner time the fraction of server resources required to support a particular client request over its expected lifetime.

Using proposed framework, we designed an admission control infrastructure for a streaming media service that can efficiently determine whether sufficient CPU, memory, and disk resources will be available for the lifetime of a particular request as a function of the requests already being delivered from the server. A performance comparison of *ac-MediaGuard* relative to a pessimistic policy that assumes all requests must be served from disk reveals a factor of two improvement in throughput.

A service provider can extend the proposed framework and infrastructure for monitoring the dynamically changing media workloads in the hosting media clusters. For example, a service provider may set two thresholds for server capacity: low - 70% and high - 95%. The admission controller *ac-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.

Delivering performance isolation to competing services while at the same time leveraging available sharing to the benefit of multiple services is an interesting and difficult consideration that may be partially addressed by the type of memory and CPU models that we have developed in this work.

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.

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. L. Cherkasova, M. Gupta. Characterizing Locality, Evolution, and Life Span of Accesses in Enterprise Media Server Workloads. Proc. of ACM NOSSDAV, 2002.
4. L. Cherkasova, L. Staley. Measuring the Capacity of a Streaming Media Server in a Utility Data Center Environment. Proc. of 10th ACM Multimedia, 2002.
5. L. Cherkasova, L. Staley. Building a Performance Model of Streaming Media Applications in Utility Data Center Environment. Proc. of ACM/IEEE Conference on Cluster Computing and the Grid (CCGrid), May, 2003.
6. L. Cherkasova, W. Tang, , A. Vahdat. MediaGuard: a Model-Based Framework for Building QoS-aware Streaming Media Services. HP Labs Report No. HPL-2004-25, 2004.
7. 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.
8. A. Dan, D. Dias R. Mukherjee, D. Sitaram, R. Tewari. Buffering and Caching in Large-Scale Video Servers. Proc. of COMPCON, 1995.
9. 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.
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. Helix Universal Server from RealNetworks Comparative Load Test. Test Final Report, KeyLabs, 2002. http://www.keylabs.com/results/realnetworks/helixcomparativeload.shtml
14. 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.
15. 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.
16. K. Nahrstedt, A. Hossain, S.-M. Kang. A Probe-based Algorithm for QoS Specification and Adaptation, in Proc. of 4th IFIP Intl. Workshop on QoS (IWQoS), 1996.
17. K. Nahrstedt, H. Chu, S. Narayan. QoS-aware Resource Management for Distributed Multimedia Applications, J. on High-Speed Networking, Special Issue on Multimedia Networking, vol.8, num.3-4, IOS Press, 1998.
18. W. Tang, Y. Fu, L. Cherkasova, A. Vahdat. MediSyn: A Synthetic Streaming Media Service Workload Generator. Proc. of ACM NOSSDAV, 2003.
19. F. Tobagi, J. Pang, R. Baird, M.Gang. Streaming RAID: A Disk Storage for Video and Audio Files. Proc. of ACM Multimedia, Anaheim, 1993.
20. H. Vin, P. Goyal, A. Goyal, A. Goyal. A Statistical Admission Control Algorithm for Multimedia Servers. In Proc. of ACM Multimedia, San Francisco, 1994.