Online Scheduling in Modular Multimedia Systems with Stream Reuse*

Michael K. Bradshaw, Jim Kurose, Prashant Shenoy, Don Towsley
Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003
{bradshaw, kurose, shenoy,towsley}@cs.umass.edu

ABSTRACT

When properly constructed, a modular multimedia system can satisfy a client's request in multiple ways by using different sequences of modules and by reusing existing streams within the system. Such flexibility in a multimedia server or proxy can provide a rich set of services to clients while efficiently utilizing system resources by choosing the best way to schedule (i.e. allocate resources for) new clients. However, it is difficult to optimally schedule clients in an online fashion as the problem is NP-complete. In this paper, we provide an efficient online algorithm to schedule client requests using a modular multimedia platform.

Categories and Subject Descriptors: C.4: Design Studies

General Terms: Algorithms, Design, Measurement

Keywords: Modular, Multimedia, Resource-Conserving

1. INTRODUCTION

The growing popularity of streaming media places an increasing strain on both network and server resources. Streaming media requires large amounts of network and disk bandwidth to successfully transmit streams from a server or proxy to the client. The most common solution is to simply purchase more resources. For instance, replicating servers and proxies in the Internet can increase both network and disk bandwidth [1, 6]. For every system there exists some workload that will completely consume one of the resources. In many of these cases, one set of resources can be substituted for another set of resources. For example, a multimedia proxy caches streams in memory to conserve network bandwidth [19, 17]. The proxy adapts to the workload by exploiting resources that are not constrained. Many systems also employ this technique [3, 2, 8, 16]. In all existing work, the system's ability to adapt to resource contention is explicitly designed for a particular problem and is not generalized to generic resources types.

*This material is based on work supported by the National Science Foundation under Grant Nos. ANI-0085848 and EIA-0080119. Any opinions, findings conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

NOSSDAV'05, June 13–14, 2005, Stevenson, Washington, USA.

Modular multimedia systems are well-suited to provide resource conserving services due to their flexibility. *Stream modules* are used to implement each stream operation in the system. A module that reads a stream from the network and one that sends a stream into the network are two examples of stream modules. In order to use multiple modules in series, *pipes* are used to redirect the output stream of one module to serve as the input stream of another module. A sequence of modules is called a *pipeline.* In order to react to fluctuations in resource consumption, a system [18] can use a lookup table to determine which pipeline should be used based on available resources. To avoid programming an exponential number of services (for each combination of modules), some systems [14, 15, 11] automate the construction of pipelines as requests arrive to the system. While Ninja [10] does choose modules (actually nodes offering a service) based on the lowest CPU load, no modular multimedia system characterizes all of the resources used to determine the best way to satisfy a request.

AMPS [20] is a modular multimedia platform capable of sharing streams between clients. To enable sharing, pipes dynamically *branch* the output stream to serve as an input stream to other modules. Branching allows the system to reuse the stream generated by previously instantiated modules to service multiple requests. In our earlier work [4], we introduced a Graph Manager (GM) as a component of AMPS that composes modules online to satisfy requests. The GM selected a pipeline to satisfy each request by examining the set of resources needed by that pipeline and resources available in the system. Finding an optimal solution is NP-complete [5] (via reduction to the Interval Classroom Assignment problem [7]). To find good solutions at online speeds, we restricted how streams were shared between clients. That restriction allowed the GM to generate solutions quickly enough to support online scheduling.

In this paper, we provide a significant improvement in the GM's ability to schedule the system's resources to satisfy client requests. First, we formalize the problem of optimally scheduling an arriving client. Second, we present an exact search algorithm based on guided search to find the optimal schedule, and offer approximation techniques that enable the GM to efficiently schedule online. Finally, our results show that the GM can schedule a client very quickly, taking on average 40 ms per client, in the presence of 3600 concurrent clients. Furthermore, we present the resource savings realized by reusing existing streams to satisfy future clients.

The rest of the paper is structured as follows. In Section 2 we review the capabilities of the AMPS system. Then, we formulate the optimization problem based on the AMPS architecture in Section 3. In Section 4 we solve the optimization problem and provide the approximations needed to speed up the execution time. Next, we describe the evaluation environment in Section 5 and the experiments in Section 6. Finally, we conclude the paper in Section 7.

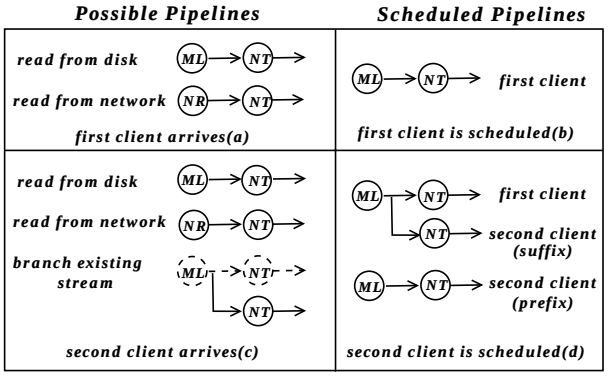


Figure 1: AMPS Capabilities: A modular multimedia platform with support for stream reuse

2. MODULAR MULTIMEDIA PLATFORM

AMPS [20] is a modular multimedia platform designed for rapid prototyping. AMPS has two components of interest to us: *stream modules* and *pipes*. Stream modules encode each stream service offered by the system such as: a transcoding service, a transmission service, or a service that generates added content. In order to use multiple modules in series, *pipes* are used to direct the output stream of one module to serve as the input stream of another module. By passing a stream through a sequence of stream modules, connected by pipes, the system can create a new service in the form of a *pipeline*. In this system, pipes can also dynamically *branch* the output stream to serve as an input stream to more than one module. Branching allows the system, typically a server or proxy, to reuse the stream generated by previously instantiated modules to service multiple requests.

Because of AMPS's modularity and branching, the system can satisfy requests by using different sequences of modules and by branching already scheduled pipelines. We present an example of AMPS's capabilities in Figure 1. Imagine a proxy with 3 modules: a Network Transmission (NT) module that sends a stream into the network, a Network Reception (NR) module that reads a stream from the network and a Memory Loader (ML) module that reads a stream from disk. When the first request arrives at the system (Figure 1(a)), the request can be satisfied in one of two ways. The proxy can read the stream from disk and send it to the network (ML and NT), or the proxy can read the stream from the network and send it to the network (NR and NT). Suppose the proxy chooses to read the stream from disk (Figure 1(b)). When a second request arrives later (Figure 1(c)), there are 3 possible pipelines; the two previously mentioned and the pipeline that branches the previously scheduled pipeline to satisfy the suffix of the stream. Note that the prefix has already been generated and cannot be shared. In this case, the branched pipeline is chosen to satisfy the suffix of the stream, and a second pipeline is generated to supply the prefix (Figure 1(d)). We assume that the client can receive more than one stream at a time and can buffer *frames* (small portions of the stream) that arrive early.

The AMPS system has a scheduler called the *Graph Manager* (GM) that determines the best way to satisfy requests. The GM's name comes from the graph-like appearance of the stream modules and pipes used to satisfy requests. The GM's task can be divided into two subtasks. First, the GM must find all possible pipelines that can be used, wholly or partially, to satisfy a new request, as seen in previous work [4]. Second, the GM must choose which pipeline or combination of pipelines to use in satisfying a request. When multiple pipelines can be used to satisfy a request, the system can use resources more efficiently, but scheduling multiple pipelines is much more complex. Our contribution focuses on the second subtask, namely that of determining which pipelines to schedule for each request.

3. OPTIMIZATION PROBLEM

In this section we present an optimization problem that arises in the AMPS system when a request arrives. The problem assumes that the system has served clients in the past and is concerned with minimizing the cost to the system of an arrival regardless of future arrivals. Note that a solution to the optimization problem may not lead to a "globally" optimal solution that accounts for all future client requests. However, in a real system, knowledge of future requests is not usually known.

In the rest of this section, we develop the full optimization problem in two steps. First, the *simplified problem* ignores some details/abilities of the AMPS system, but it provides the intuition for the heuristic driven search introduced in Section 4.1.2. Second, the *full problem* captures all of the abilities of the AMPS system. In particular, it models the GM's ability to schedule multiple portions of the stream at the same time.

3.1 Simplified Problem

A *request* v is a 3-tuple (b, e, t) that describes the deadline by which the GM must produce each frame. The GM must schedule all frames between $v.b$ and $v.e$. The first frame $v.b$ must be sent by the deadline $v.t$. All remaining frames $x, v.b \leq x \leq v.e$ have deadlines $v.t + (x - v.b)$.

In the simplified problem, the GM must decide how to satisfy the request when all frames are produced at their deadlines. Time is slotted. Each slot represents an interval of time, and each slot is indexed by an integer. Denote *current time* by t_0. The system has k limited *resources* that are consumed when the GM generates a frame. A resource can be the bandwidth or the memory that is available to the system. Let $\vec{R}(t) = (R_1(t), \ldots, R_k(t)); 0 \leq R_i(t) \leq 1$ represent the utilization of each of the k resources at time t. When we discuss resources, we always talk about what percentage of the total resources is used. For this reason, the system's resources $\vec{R}(t) = (R_1(t), \ldots, R_k(t))$ at time t are real values that lie between 0 and 1. The amount of resources available varies over time since the GM can experience different request rates, different types of requests, or choose to satisfy similar requests in different ways.

A *stream segment* is a potential, partial solution that has been generated by the GM. The choice of which modules to use (pipeline) and whether or not to make use of an existing stream are abstracted away (see previous work [4]). A stream segment s is a 4-tuple $(b, e, t, \vec{r})$ that describes a stream that could be scheduled by the GM. The stream segment sequentially produces frames $s.b$ to $s.e$. The stream segment produces each frame $x, s.b \leq x \leq s.e$, at time $s.t + (x - s.b)$. $s.\vec{r} = (s.r_1, \ldots, s.r_k); 0 \leq s.r_i$ is the vector of the k resources consumed to generate each frame in this stream segment. The values of $s.r_i, 1 \leq i \leq k$ represent what fraction of the system's i^{th} resource is needed. When $s.r_i > 1$ the stream segment requires more resources than are present in the system.

$\mathcal{S}$ is the set of all stream segments the system can produce to satisfy a request v. All members of $\mathcal{S}$ only generate frames *at* the deadlines specified in the request v. Note that a particular stream segment might not contain all frames in a request due to branching of ongoing streams.

```
FullSearch(p[], c_l, x)
1   S' := S
2   Remove all a ∈ S' from S'
3       if (x < a.b) or (a.e > x)
4   while (S' not Empty)
5       choose s ∈ S' such that ∀_{a∈S'}
6           c(R(s.t − (x − s.b) + s.r⃗) ≤
7               c(R(a.t − (x − a.b) + a.r⃗)
8       Remove s from S'
9       p[x] := s
10      if( f(p[]) + h(x)) < c_l)
11          if (x = v.e)
12              c_l := f(p[])
13          else FullSearch(p[], c_l, x + 1)
```

Figure 2: Full Search Algorithm

The cost function $c : [0,1]^k \rightarrow \mathcal{R} \cup \{\infty\}$ maps a vector of resource utilizations, $\vec{r} = (r_1, r_2 \ldots r_k; 0 \leq r_i)$, to a real number (or infinity if unsatisfiable). The cost function is:

$$c(\vec{r}) = \begin{cases} \sum_{i=1}^{k} r_i^2 & \text{if } \forall_{1 \leq i \leq k} \quad r_i \leq 1 \\ \infty & \text{if } \exists_{1 \leq i \leq k} \quad r_i > 1 \end{cases} \tag{1}$$

The cost function is the sum of the squares of the resource utilizations. The cost function is convex and increases faster than the resource utilization as the utilization approaches one. This cost increase discourages the GM from selecting stream segments that require heavily utilized resources and to instead favor stream segments that consume underutilized resources. If any of the resource utilizations is greater than one, the cost is ∞ and indicates that the system does not have the resources needed to satisfy the request.

The best solution not only satisfies the constraints of the request but also has the lowest sum of the marginal costs for each time slot. The optimization problem is to create a mapping $\mathcal{P} : x \rightarrow s$ for each frame $x, v.b \leq x \leq v.e$, to a stream segment $s \in \mathcal{S}$ that will generate that frame so as to minimize:

$$f(\mathcal{P}, \vec{R}(), t_0) = \tag{2}$$
$$\sum_{t=t_0}^{\infty} c(\vec{R}(t) + \sum_{(x,s) \in \mathcal{P}} I(t = s.t + (x - s.b))s.\vec{r}) - c(\vec{R}(t))$$

where $I()$ equals 1 if the argument is true and 0 otherwise. Note that not all stream segments can satisfy each frame in the request. The lowest cost schedule could use multiple stream segments to satisfy the request.

Since no two frames in the request are generated at the same time, the optimization problem can be restated to minimize:

$$f(\mathcal{P}, \vec{R}()) = \sum_{(x,s) \in \mathcal{P}} c(\vec{R}(s.t + (x - s.b)) + s.\vec{r}) \tag{3}$$

Equation (3) is used in the heuristic driven search presented in Section 4.1.2.

3.2 Full Problem

The full problem is based on the simplified problem. In the full problem, we relax the constraint that forces all members of $\mathcal{S}$ to generate frames at their delivery deadlines. As a consequence, the full problem requires that we minimize Equation (2) to find the optimal solution.

4. ALGORITHMIC SOLUTIONS

In this section we present the algorithms to solve the optimization problem in Section 3.2. First, we design an optimal algorithm to solve the optimization problem. While this algorithm will always find a solution, we show in Section 6 that it is not able to schedule solutions quickly enough to be used in real-time. Then to speed up the algorithm, we introduce two approximation techniques that reduce the search space and reduce the calculation times.

4.1 Exact Algorithms

We build our exact algorithm by first defining a search algorithm for the simplified problem. We then use this search algorithm as a heuristic to guide the search for the solution to the full problem.

4.1.1 Simplified Search Algorithm

The simplified problem can be solved in a straightforward manner. Each mapping of a frame to a stream segment can be solved independently of the other frames see Equation (3). To map frame x to a stream segment, the GM searches through all stream segments and chooses the segment s with the lowest cost $c(\vec{R}(s.t + (x - s.b)) + s.\vec{r}); s.b \leq x \leq s.e$.

4.1.2 Full Search Algorithm

The full problem is more difficult to solve in that the cost to generate a frame depends on how many other frames are generated during the same time slot. In essence the cost of a solution cannot be calculated until each frame in the request has been mapped to a stream segment. A full enumeration and evaluation of all possible mappings is not tractable. Instead, the GM must utilize a guided search to find a solution.

We use a greedy search inspired by A^* [12] to quickly search for the best solutions. A^* search is a breadth-first, guided search algorithm that is guaranteed to find an optimal solution. Unlike A^*, our algorithm uses depth-first search to conserve memory. Our A^* inspired algorithm has three components, an incremental search space, a non-decreasing cost function $f()$ and a heuristic function $h()$. The GM sequentially decides each frame-stream segment mapping starting at the first frame $v.b$ and ending at the last frame $v.e$. This forms the incremental search space. $f(p[])$ is the non-decreasing cost function where $p[]$ represents the array of all the stream segments that have been assigned to frames. This cost function can be calculated by summing the costs, $c()$, incurred in each time slot from time t_0 to $v.t + (v.e - v.b)$. The heuristic function $h(x)$ is the cost of the solution found by the simplified search algorithm for request $v' = (x + 1, v.e, v.t + (x + 1 - v.b))$. In other words, the heuristic function sums the lowest cost mapping for all non-assigned frames without considering that two frames could be generated in the same time slot. If two frames are generated at the same time, the total cost would be greater than if they were produced separately. Recall that the cost function $c()$ sums the square of each resource's utilization. Therefore, if we assume that no frames were generated at the same time, we can produce an estimate that will never exceed the true cost.

The psuedocode for the `FullSearch` algorithm is located in Figure 2. The GM picks the stream segment that can generate frame x with the lowest cost, ignoring any possible interactions with other frames (lines[5-7]). If the cost of the satisfied frames $f(p[])$ plus the heuristic cost $h(x)$ to satisfy the rest of the frames is less than the lowest cost c_l (line 10), the GM recursively solves for the next frame (line 13). Otherwise, the GM selects the next lowest cost stream segment (line 4). When there are no more stream segments available, the GM backtracks and repeats the process with the previous frame.

4.2 Approximating Frame Space (Frame-Buckets)

The `FullSearch` algorithm does not take advantage of the fact that the best stream segment to generate frame x is very likely the best frame to service frame $x+1$. To exploit this trend, the frames of the requested video are grouped into *frame-buckets*. Instead of choosing a stream segment for each frame as in the exact algorithm, we now select a stream segment for each frame-bucket. Since the stream segment must satisfy all frames within a bucket, the algorithm only considers stream segments that can satisfy each frame in the bucket. By reducing the number of choices, the GM can more quickly come to a decision although not the greedy optimal one.

In this paper we examine two different bucketing algorithms. Equal spaced (Equi-spaced) buckets have an equal number n of frames in each bucket. Geometric spaced (Geo-spaced) buckets have m times more frames in each successive bucket, where m is a constant. For example an Equi-spaced bucketing algorithm with $n = 4$ would place 4 frames in each frame-bucket, while a Geo-spaced bucketing algorithm with $m = 2$ would fill the first 4 buckets with $1, 2, 4, 8$ frames respectively. We will examine the effectiveness of both techniques in Section 6.1.

4.3 Approximating Resource Utilization (Time-Buckets)

In the exact algorithm, calculation of the values $f()$ and $h()$ depends on $\vec{R}(t)$. This means that the GM must calculate the cost for each value of t. To avoid recalculating $f()$ and $h()$ for each value of t, we use an enveloping technique [9] to represent resource utilization for larger periods of time. Time-slots are grouped into *time-buckets* in much the same manner as frames were grouped into frame-buckets in Section 4.2. For each time-bucket covering time-slots t_1 to t_2, define $\vec{R}'(t) = (\max R_1(t) ... \max R_k(t)); \forall_{t;t_1 \leq t \leq t_2}$ as the maximum resource utilization during all of the time-slots within the bounds of the time-bucket. Since $\vec{R}'(t)$ has the same value for multiple values of t, values of $h()$ and $f()$ do not need to be recalculated for each time-slot but rather for each bucket.

5. EVALUATION ENVIRONMENT

The GM is a scheduler for AMPS, a modular multimedia platform with stream reuse. To be effective, the GM must quickly schedule clients and provide the schedule to the system. The GM only needs to be fast enough to maintain the maximum throughput of the system without interfering with the system's performance. We use a simulation environment to determine if the GM can meet these requirements and discover any resource savings that the GM can offer. The GM is configured to operate as the scheduler for a forwarding proxy, the streams are not actually sent, but resource utilization is recorded. The set of stream segments $\mathcal{S}$ is generated as detailed in previous work [4]. The GM accepts requests from the simulator and returns a schedule for each request. The GM must schedule all portions of the stream *by* the playback time, but the GM may schedule portions of the stream *before* their playback time.

The proxy has three stream modules: a network reception module that receives streams, a network transmission module that sends streams out, and a time shifting module that can buffer streams in memory until their playback time. The proxy also has 3 resources: memory, which is used by all three modules, server bandwidth, which is used by the network reception module (between the server and the proxy), and client bandwidth, which is used by the network transmission module (between the proxy and the client). The proxy has the average amount of each resource needed by a simple forwarding proxy to satisfy all requests. A simple forwarding proxy fetches each stream and forwards the stream to each requesting client. Note that with this amount of resources, it is unlikely that the simple forwarding proxy could service every arrival, as there is some irregularity in the arrival process.

We make use of two simulator settings: Clip and Movie. In the *Clip Setting*, the simulator generates requests for a single two minute video clip. In the *Movie Setting*, the simulator generates requests for a single one hour video. In both settings, the simulator generates requests using a Poisson arrival process with a mean of one request per second. The simulation is conducted for 4 times the length of the video type requested in order to reach steady state. Steady state occurs soon after the simulation has run for the length of the video. It is at this point that the system has (on average) a constant number of clients. All experiments were conducted on a Condor cluster with machines ranging from 2.4 GHz to 3 GHz processors.

6. EXPERIMENTS

We ran preliminary experiments to evaluate the performance of the exact search algorithm. We found that 5% of the searches required more than 10 hours to schedule a request (searches were prematurely halted in order to continue the simulation). Further investigation showed that the scheduling problem is NP-complete via a reduction to the Interval Classroom Assignment problem [7]). Details of these experiments can be found in the technical report [5]. The rest of this section evaluates the approximations needed to make the scheduling problem tractable. Furthermore, to avoid corner cases, the GM only evaluates a maximum of 1000 schedules before choosing the best one visited.

6.1 Evaluating Approximations

After examining the baseline behavior, the need for approximation is apparent. In this section we conduct a series of experiments to determine how well the approximations listed in Sections 4.2 and 4.3 perform. The results of each experiment are averages of 10 runs. Time- and frame-buckets are either Equi-spaced or Geo-spaced buckets. When using Equi-spaced buckets, we will report the size of the bucket in seconds. Frame-buckets contain frames but only one frame is generated each second. When using Geo-spaced buckets, the first bucket is 1 second (frame) long. Each additional bucket is m times the size of the last bucket. So for a value of $m = 2$, the next three buckets would be $2, 4$ and 8 seconds long respectively. We carefully chose values of m to insure that the sum of the frames in each bucket equals the number of frames in the Clip video.

6.1.1 Approximations over 2 minute clips

The first experiment examines the behavior of using Equi-spaced time- and frame-buckets under the Clip setting. The size of the buckets range from 1 second to 120 seconds, the size of the requested stream. The achieved throughputs are recorded in Table 1 and the execution times are recorded in Table 2. This experiment exhibits several interesting results. First, we note that when the frame-bucket is $120s$ long the proxy behaves like a simple forwarding proxy that can only request streams from the server and relay them to the client. Because of natural fluctuations in the arrival process, 100% throughput is not possible and only 90.42% of requests are satisfied.

Second, when the time- and frame-buckets are $1s$ long, the approximation is the closest to the exact solution. The throughput is significantly less than that of a simple forwarding proxy. We examined the solutions generated by the search algorithm and dis-

Frame Buckets	Time Buckets					
	1s	10s	20s	40s	60s	120s
1s	.759	.747	.675	.618	.572	.774
10s	.908	.909	.792	.729	.693	.702
20s	.906	.905	.906	.801	.716	.691
40s	.905	.905	.905	.906	.877	.741
60s	.905	.905	.905	.905	.905	.832
120s	.904	.904	.904	.904	.904	.904

Table 1: Throughput achieved by using Equi-spaced buckets

Frame Buckets	Time Buckets					
	1s	10s	20s	40s	60s	120s
1s	8130	221	98.9	48.4	21.9	224
10s	31.4	15.0	14.6	10.5	8.58	7.22
20s	45.7	9.59	8.93	7.47	6.05	3.80
40s	37.6	8.62	7.81	7.51	5.11	2.54
60s	36.4	9.01	9.66	9.01	9.18	3.67
120s	23.4	6.18	5.51	5.34	6.25	6.10

Table 2: Execution time in ms using Equi-spaced buckets

covered two trends. One, the solutions were composed of many small stream segments each 1-2 frames long. Two, many of these stream segments (sometimes as many as seven) were scheduled in the same time slot. Recall that the cost $c()$ is quadratic. The cost to generate two frames in the same time slot is greater than the cost to generate the frames in separate time slots. The search algorithm schedules frames in the same time slot only when the cost savings is greater than the cost penalty. In practice, the search algorithm finds a smaller increase in cost to schedule short stream segments compared to the savings gained from branching the stream. By scheduling frames at the same time, the proxy experiences short transient spikes in the resource utilization. These spikes in resource utilization only span 1-2 seconds. When the spike reaches 100% utilization, the proxy cannot schedule any streams during that time period. This shows that the greedy solution is not optimal for the global scheduling problem.

Third, when the frame-buckets are smaller than the time-buckets, the throughput is poor. When the frame-buckets are as long as or longer than the time-buckets, we see throughput that matches or exceeds the throughput of the simple forwarding proxy. The greater throughput is due to the algorithm taking advantage of small dips in the resource utilization to schedule streams ahead of time. The proxy satisfies that client more quickly and is able to satisfy the next client as those resources are freed. However, this event occurs rarely and does not significantly improve the throughput.

Last, the execution time to find a solution decreases as the size of the buckets increase. Larger time-buckets decrease the execution time to calculate values of $h()$ and $f()$. Larger frame-buckets reduce the search space and thus make the search easier. Frame-buckets also produce a feedback effect. Each stream segment scheduled by the algorithm can be branched and reused to satisfy a later request. Fewer stream segments are scheduled when larger frame-buckets are used, and fewer choices are available in the future. This breaks down when the time-buckets are $120s$ long, in particular with $1s$ long frame-buckets. Since the time-bucket (only one) is the size of the stream, the system is using the peak resources utilization for the calculations of $h()$. As a consequence, the heuristic cost to produce a frame for any branched stream segment is the same.

Bucket Size	5 m	10 m	15 m	20 m	30 m	60 m
Segments	12	6	4	3	2	1
Throughput	.983	.982	.982	.983	.982	.982
Exec. Time	823	445	362	308	313	445

Table 3: Movie setting using Equi-spaced time- and frame-buckets, measured in minutes

Multiplier	1.447	1.975	4.925	7.48	14.98	59.5
Segments	20	12	6	5	4	3
Throughput	.983	.983	.983	.982	.982	.981
Exec. Time	433.7	265	127	91.3	37.3	16.0

Table 4: Movie setting using Geo-spaced time- and frame-buckets

Since the costs are identical the guided search is not able to discard any of them, and the GM must evaluate each of them. In general, the approximations decrease the running time of the search.

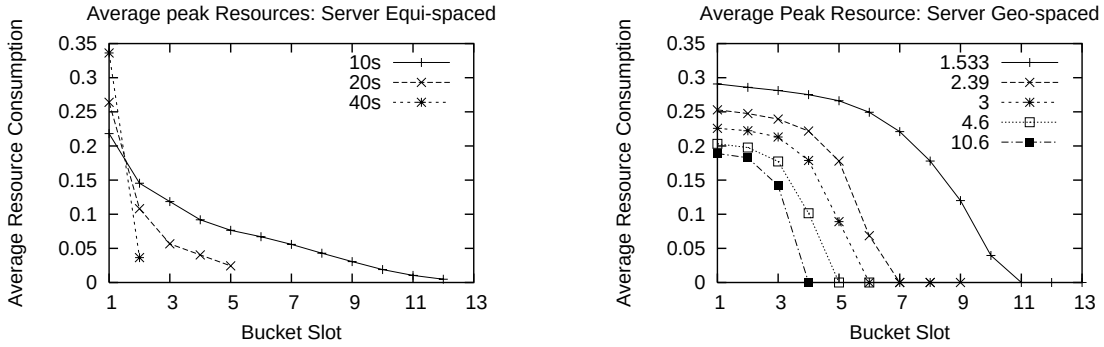
We conducted similar experiments using Geo-spaced buckets under a Clip setting. The results are similar to the Equi-spaced results. When the frame-buckets are smaller than the time-buckets we see poor throughput. When the frame-buckets are as long as or longer than the time-buckets, we see throughput that matches or exceeds the throughput of the simple forwarding proxy. The execution times when using the Geo-spaced buckets decrease as the size of the buckets increases. However, there is a (small) speedup when using Geo-spaced buckets over Equi-spaced buckets in the Clip setting.

6.1.2 Approximations over 1 hour movies

Our next pair of experiments measure the throughput and execution times of longer streams. We conduct each experiment using the Movie setting. In the first experiment, we examine the performance of using Equi-spaced time- and frame-buckets. In the second experiment we examine the performance of using Geo-spaced time- and frame-buckets. For both experiments the time- and frame-bucket are the same length, and the parameters are only reported once. The results of using Equi-spaced buckets are presented in Table 3. The throughput achieved by a simple forwarding proxy is 98.28%. We see this throughput when the time- and frame-buckets are both 60 minutes long. The throughput for other bucket sizes are similar, but the execution time decreases for larger buckets (except for the 60 min buckets). The results from using Geo-spaced buckets are presented in Table 4. The throughput is similar to the throughput achieved using Equi-spaced buckets. However, the execution times experienced when using Geo-spaced buckets are up to an order of magnitude lower than using Equi-spaced buckets.

6.2 Effectively Using Resources

In this section, we report the average resource consumption using the Clip setting for each of the three resources: client bandwidth, server bandwidth and memory. For each experiment, we provide the average resource consumption per time-bucket. The size of each bucket is determined by the bucketing technique used and the parameter used. Due to space constraints we only present data on the server bandwidth utilized. See [5] for the utilization of other resources. The server bandwidths consumed using Equi-spaced and Geo-spaced buckets are shown in Figure 3. In reading these graphs the most important data is the utilization of the first bucket. In all graphs the resource utilization drops off after the first bucket and



(a) Equi-spaced Buckets of Length n

(b) Geo-spaced Buckets with Parameter m

Figure 3: Average Peak Resource Utilization of Server Bandwidth Using Identically Sized Time- and Frame-Buckets

the utilization of the first bucket limits the ability to service new requests. When using Equi-spaced buckets (Figure 3(a)), the resource utilization decreases when *smaller* buckets are used. When using Geo-spaced buckets (Figure 3(b)), we observe the opposite trend. The resource consumption decreases when *larger* buckets are used. Geo-spaced buckets provide a double win since using larger buckets decreases both the resources consumed and the execution times needed to satisfy a request.

7. DISCUSSION

Given any system that preforms one action (such as reading a file from disk) with a static workload, we are confident that a skilled team of programmers can develop the software necessary to meet that challenge. However, the GM is useful for any multimedia system that exhibits one or more of the following traits: 1) contains a large set of stream modules in which multiple sequences can satisfy the same request; 2) concurrently provides a stream to multiple clients using different services and different starting times; 3) offers a large and changing set of services; and/or 4) is resource-constrained. When there are multiple sequences of stream modules that can satisfy a request, the GM can determine which is best, given the present resource usage. When a server or proxy must serve the same stream, but offer different services and different starting times to different clients, the GM determines those portions of existing streams that can be reused, by buffering within the system or by providing more than one stream to the client. When a system needs to offer a large and changing set of services, the GM can help the system avoid recompilation. The GM determines the best way to use new modules in coordination with old modules to satisfy requests. This ability avoids costly rewrites when new modules become available. When resources are constrained, the GM can determine how to utilize resources to service more requests.

For any benefit, we can expect to pay a cost. The time required to schedule streams using the GM is likely to be higher than the time needed for a straightforward system. In this paper we provided measurements to determine the speed with which the GM can schedule and have found the cost per client to be low, on the order of 10s of milliseconds for hour long streams with an average of 3600 simultaneous clients. Recall that each stream used to serve the existing 3600 clients must be evaluated to determine if they can be reused for the new client. Workloads with less sharing require less processing. But, is 3600 clients at reasonable limit? The AMPS proxy [20] supported a maximum of 710 clients using 300k streams using a 2 GHz processor. Performance studies of Microsoft and RealNetworks servers [13] exhibited maximum throughput rates for 500k streams well under 1000 simultaneous clients using dual 700 MHz processors. Since these experiments were conducted on different servers and under different conditions they cannot be compared directly, but used to illustrate arrival rates that entry level multimedia servers can support. The overhead of using the GM on these systems is minimal, but offers the dual benefits of providing more services and more efficient use of resources.

8. REFERENCES

[1] S. Acharya and B. Smith. Middleman: A video caching proxy server. In *Proc of NOSSDAV*, 2000.

[2] J. M. Almeida, D. L. Eager, and M. K. Vernon. A hybrid caching strategy for streaming media files. In *Proc. of MMCN*, January 2001.

[3] E. Amir, S. McCanne, and H. Zhang. An application level video gateway. In *Proc. ACM Multimedia*, November 1995.

[4] M. K. Bradshaw, J. Kurose, L. J. Page, P. Shenoy, and D. Towsley. A reconfigurable, on-the-fly, resource-aware,streaming pipeline scheduler. In *Proc. of MMCN*, January 2005.

[5] M. K. Bradshaw, J. Kurose, P. Shenoy, and D. Towsley. Online scheduling in modular multimedia systems with stream reuse. Technical Report 05-21, Department of Computer Science, University of Massachusetts Amherst, 2005.

[6] D. W. Brubeck and L. A. Rowe. Hierarchical storage management in a distributed vod system. *IEEE MultiMedia*, 3(3):37–47, 1996.

[7] M. W. Carter and C. A. Tovey. When is the classroom assignment problem hard? *Oper. Res.*, 40(S1):28–39, 1992.

[8] A. Dan and D. Sitaram. A generalized interval caching policy for mixed interactive and long video environments. In *Proc. of MMCN*, 1996.

[9] V. Firoiu, D. Towsley, and J. Kurose. Efficient admission control for edf schedulers. In *Proc. IEEE INFOCOM*, April 1997.

[10] S. D. Gribble and et al. The ninja architecture for robust internet-scale systems and services. *Journal of Computer Networks*, 35(4), March 2001.

[11] GStreamer Team. GStreamer: open source multimedia framework. http://www.gstreamer.net.

[12] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determiniation of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, pages 100–107, 1968.

[13] KeyLabs. Helix universal server from realnetworks comparative load test, May 2002. www.keylabs.com.

[14] Z. M. Mao, H. sheung Wilson So, and B. Kang. Network support for mobile multimedia using a self-adaptive distributed proxy. In *Proc of NOSSDAV*, 2001.

[15] Microsoft Inc. Directshow. http://msdn.microsoft.com.

[16] G. J. M. Smit et al. Reconfiguration in mobile multimedia systems. In *Proc. of PROGRESS workshop*, October 2000.

[17] Z. Miao and A. Ortega. Proxy caching for efficient video services over the internet. In *Packet Video Workshop*, April 1999.

[18] E. J. Posnak, H. M. Vin, and R. G. Lavender. Presentation processing support for adaptive multimedia applications. In *Proc. of MMCN*, January 1996.

[19] R. Rejaie, M. Handley, H. Yu, and D. Estrin. Proxy caching mechanism for multimedia playback streams in the internet. In *Proc. of the 4th International Web Caching Workshop*, March 1999.

[20] X. Zhang and et al. AMPS: A flexible, scalable proxy testbed for implementing streaming services. In *Proc of NOSSDAV*, Kinsale Ireland, June 2004.