A reconfigurable on-the-fly resource-aware streaming pipeline scheduler

Michael Bradshaw, Jim Kurose, Lela Jane Page, Prashant Shenoy, Don Towsley

Michael K. Bradshaw, Jim F. Kurose, Lela Jane Page, Prashant J. Shenoy, Don Towsley, "A reconfigurable on-the-fly resource-aware streaming pipeline scheduler," Proc. SPIE 5680, Multimedia Computing and Networking 2005, (17 January 2005); doi: 10.1117/12.592262

SPIE.

Event: Electronic Imaging 2005, 2005, San Jose, California, United States

A Reconfigurable, On-The-Fly, Resource-Aware, Streaming Pipeline Scheduler

Michael K. Bradshaw, Jim Kurose, Lela Jane Page[§], Prashant Shenoy, Don Towsley

Dept. of Computer Science
University of Massachusetts
Amherst, MA 01003
{bradshaw, kurose, shenoy, towsley}@cs.umass.edu

[§] Centre College
600 W. Walnut St
Danville, KY 04022
lelajanepage@hotmail.com

ABSTRACT

Multimedia systems use stream modules to perform operations on the streams that pass through them. To create services, programmers hand-code modules together to form a pipeline. While some advances have been made in automating this process, none yet exist that create pipelines that are fully aware of the system's resources. In this paper, we present the design and evaluation of the Graph Manager (GM), a pipeline scheduler that determines, on-the-fly, the best way to satisfy requests using stream modules and reusing existing streams in the system.

1. INTRODUCTION

Building networked multimedia systems has become an artform embraced by the networking community over the course of the past 15 years. Networked multimedia systems integrates real time systems and networks and nurtures a host of interesting problems. In the 1990s, a consensus formed in the research community regarding the architecture of these systems. Chief among these designs was the concept of modularity, the ability to create systems from predesigned components to allow for rapid prototyping of more interesting algorithms and to foster sharing of code within the community. [1] While efforts to unite the community behind a particular multimedia toolkit have proved unsuccessful, [2–4] such systems have blossomed – even entering into the mainstream commercial sector. [5]

One particular challenge in the design of modular multimedia systems is the modularization of subsystems that operate on the streams. Stream modules are used to implement each stream operation in the system. 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. By passing a stream through a sequence of stream modules, connected by pipes, a programmer can create a new service from the pipeline of stream modules. The simplest application of this technology is to hand-code pipelines [6–11] to create a particular multimedia service. Table-based systems are more flexible. Programmers hand-code several pipelines and the system uses a lookup table to determine which pipeline can be used to provide a requested service. [12,13] Some table-based systems[14] can also choose pipelines based on available resources. These require a programmer to specifically build each service. More dynamic systems[15,16] define a specific pipeline. Each position in the pipeline can be filled using a specific class of module. At runtime, the system fills in each position of the pipeline using the modules registered to provide the required stream services and/or formats. Class-based pipelines are much more versatile, but are limited in that the specific pipelines do not offer enough expressiveness to cover all pipelines that can be constructed.

Fully automated efficient pipeline construction presents multiple challenges. Some systems [5,17–19] build the pipeline on-the-fly as requests arrive. These systems search through stream module descriptions to find and instantiate a sequence of modules that can satisfy individual requests. In these systems each stream module is limited to one input stream and one output stream. As the number of possible modules becomes large, there may exist different combinations of modules that perform the same task. When more than one module pipeline can solve a task, there is an opportunity to use resources, such as memory or bandwidth, more efficiently. In more sophisticated systems, [13,20] scheduled streams within the system are also resources. In these systems, pipes can 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 before the branching point to service multiple requests.

To date there has not been pipeline scheduler that can both generate a complex pipeline, with more than one input stream or output stream, and choose between alternate pipelines based on the resources consumed. A pipeline scheduler that is resource-aware has two significant advantages over existing pipeline generation models. First, the scheduler is aware

Multimedia Computing and Networking 2005, edited by Surendar Chandra, Nalini Venkatasubramanian, Proc. of SPIE-IS&T Electronic Imaging, SPIE Vol. 5680 © 2005 SPIE and IS&T · 0277-786X/05/$15

of the resources present in the system and can create pipelines that conserve heavily utilized resources by using alternative sequences of stream modules or by branching an existing stream. Second,the scheduler allows programmers to focus on higher level multimedia algorithms such as broadcast schemes[21–23] rather than building a system from the ground up. The scheduler handles the hard work of insuring that pipelines are efficient. *In this paper, we present the design and evaluation of the Graph Manager (GM), a pipeline scheduler that determines, on-the-fly, the best way to satisfy requests using stream modules and reusing existing streams in the system.*

The rest of this paper is structured as follows. We present the design of the GM in Section 2. In Section 3, we address concerns about running time. In particular, we show that exponential running time is not a common occurrence. Section 4 summarizes this work and offers future research topics for the GM.

2. GRAPH MANAGER ARCHITECTURE

The Graph Manager (GM) is a scheduler for modular multimedia systems. The GM accepts requests and returns the sequence of modules that satisfy the request, making the most efficient use of limited system resources. The GM is targeted for modular multimedia systems that exhibit 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, the GM determines those portions of existing streams that can be reused, by introducing buffering in 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. In the rest of this section we discuss the modules that can be represented by this system and the steps that the GM takes to satisfy a request: enumerating, matching and searching.

2.1. Representing the stream modules

One of the main contributions of this work is the design of a GM that is able to represent and use stream modules with multiple inputs and outputs to satisfy requests. The GM describes stream modules using XML files. The XML description allows stream modules to be described with multiple input and output streams. These streams are typed (mpeg,avi,...). An input stream must match the input stream type described in the module's description. In addition to typing, the XML description also indicates the amount of resources, such as memory, CPU, or bandwidth, needed to generate each output stream. Finally, the XML file lists the services that are performed on the stream that passes through that module. To avoid the high cost of parsing text files, all descriptions are read into an internal format during the system's initialization.

2.2. Enumerating solutions from requests

The first step in satisfying a request is to enumerate (list) all possible combinations of stream modules that could satisfy the request. *Requests* are represented using a tree data structure called a *presentation tree.* Nodes in the presentation tree indicate a set of services that must be performed on the stream. The links in the tree indicate the order that each node should be visited. Children nodes are satisfied before their parents. Parents with multiple children indicate a service that requires multiple streams as inputs.

The GM enumerates each solution by recursively satisfying each subtree in the presentation tree starting at the leaf nodes. Each enumerated solution is stored in a *candidate stream*. A candidate stream records the format, cost, schedule and the graph of stream modules needed to produce a solution. A request node is *satisfied* when the graph manager finds all sequences of stream modules that can use the output streams generated for the children nodes as inputs and provide all the services requested by the node. Each such solution is stored as a candidate stream. Once the graph manager satisfies the root node, the graph manager has the list of all candidate streams that can satisfy the request

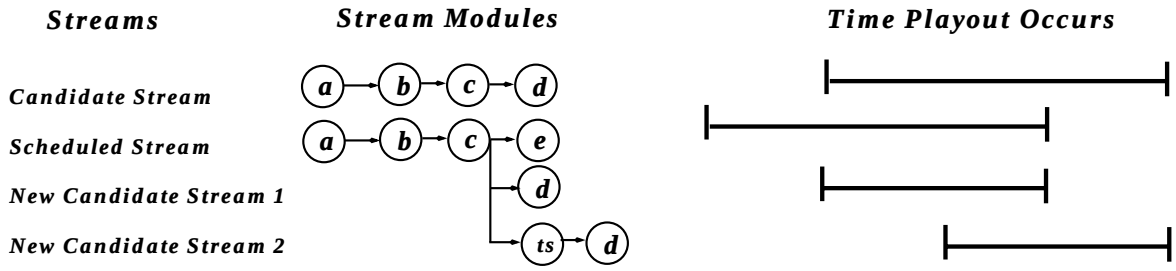

Figure 1. Matching Scheduled Streams

2.3. Matching scheduled streams

In the second step, the GM uses the candidate streams to guide the search for existing *scheduled streams* that could be reused to satisfy the present request. A scheduled stream is an existing candidate stream that was used to satisfy a request and is stored with the internal representations of each stream module description in the original candidate stream. The record at each stream module includes the schedule and a *path string* that uniquely identifies the initial multimedia stream and the modules used on that stream up to the present stream module. For instance, if the candidate stream plays stream 2 with schedule S and passes through stream modules a, b and c, then the GM stores schedule S at each module and path strings 2, $2a$ and $2ab$ at modules a, b and c respectively. To find all scheduled streams that can be used to satisfy the present request, the GM visits the stream modules of each candidate stream. At each stream module, the GM compares the path string of the scheduled stream and the candidate stream. If the path string indicates that the streams match, the GM compares the schedules and determines what segments of the scheduled stream, if any, can be reused to satisfy the new request.

Scheduled streams can be reused in one of two ways as demonstrated in Figure 1. Let n be the sequence of stream modules that the candidate stream visits and p be the longest common prefix of stream modules that both the scheduled stream and the candidate stream have in common. Let s be the resulting suffix of modules in which the scheduled stream and enumerated solution differ, in other words $n = ps$. In Figure 1 n is $abcd$, the prefix p is abc, and the suffix s is d. First, the GM constructs a new candidate stream starting from the last matching stream module and whose suffix equals s. Second, the GM constructs a new candidate stream starting from the last matching stream module linking to a stream module that *time shifts* the stream, and whose suffix equals s. A time-shifting module buffers the stream until the original candidate stream's playback time. In the new candidate streams formed, resources are only consumed by the modules in the suffix. In this paper the GM always creates candidate streams, in the matching phase, by using a time-shifting module. When time-shifting is not used, searching for the best solution becomes computationally expensive. In future work we discuss the challenges and performance benefits of removing the time-shifting constraint.

2.4. Searching for the best solution

The final step is to determine the best solution with the lowest marginal system cost. The system cost is the sum of the *instantaneous cost* over present and future instances of scheduled streams under the assumption of no future arrivals. The instantaneous cost describes the cost to the system at a particular point in time. The instantaneous cost function is defined to be the sum of the squares of the resource utilizations. We choose the square of the utilization since it increases faster than the utilization as the utilization approaches 100%. This cost increase discourages the GM from selecting pipelines that use heavily utilized resources and to instead favor pipelines that consume underutilized resources. Thus for a vector of resource utilizations $\vec{r}(t) = \{r_1(t), r_2(t) \ldots r_k(t); 0 \leq r_i \leq 1\}$ at time t, the instantaneous cost function is $c(\vec{r}(t)) = \sum_{i=1}^{k} r_i^2(t)$. Then, the system cost function at time t_0 is $s(t_0) = \int_{t=t_0}^{\infty} c(\vec{r}(t))dt$. The request's solution is found by pairing each *frame* with a candidate stream that can generate that *frame* with the lowest marginal cost. Here a frame is a non-divisible unit of the stream. Note that candidate streams derived from scheduled streams might not be able to provide all of the frames in a stream.

In practice, the GM does not perform the cost calculations using the specific resource utilization for all future times, as defined by the system cost. As there could realistically be thousands of existing scheduled streams the cost to calculate the system cost would be too great. Instead, the GM uses the peak resource utilization plus the solution's resource needs when

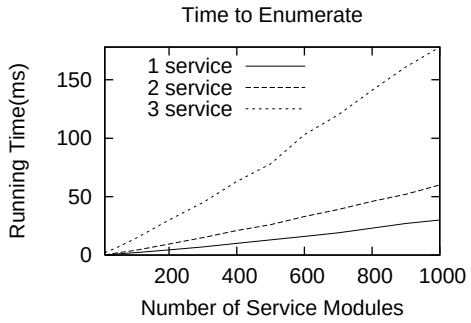

Figure 2. Running Time to Enumerate All Possible Solutions

calculating the system cost. Using peak resource utilizations simplifies the calculation of the system cost and ensures that the GM will never schedule streams that the system cannot support.

3. PERFORMANCE EVALUATION

The problem of enumerating all of possible combinations of stream modules that can satisfy the request is NP complete. However, the worst case running times only occur for very peculiar requests. In this section we present a set of experiments that demonstrate running time of enumerating all configurations of modules that can satisfy a request.

All experiments are conducted on a 2 GHz Pentium 4 processor with 512MB of RAM running Redhat Linux 9.0. The experimental system has one input module *in*, one output module *out* and N service modules. Each service module is named after the service it provides $s_i, 1 \leq i \leq N$. All modules operate on the same stream format. We conducted three experiments, each with 400 trials for different values of N. In the first experiment we sent a request to the GM for a stream that provides only one service s_1. As only one stream module provides s_1 there is only one way in which the request can be satisfied, using the sequence of modules *in*,s_1,*out*. In the second experiment we sent a request to the GM for a stream that provides two services: s_1 and s_2. We make no restrictions on the order of the services so there are two ways in which the GM can satisfy the request. In the final experiment we sent a request to the GM for a stream that provides three services: s_1, s_2 and s_3. There are a total of six ways that the GM can satisfy the request.

The results of these experiments are plotted in Figure 2. The running time increases in a roughly linearly fashion for increasing values of N. The slope of the cost increase is related to the number of enumerated solutions that the GM can produce. This is because the GM can prune paths once it realizes that a node provides a service that is not required. As the GM will cycle through all the nodes once (because all nodes accept the same stream formats) for each additional enumerated solution, the number of additional comparisons is proportional to N.

Requests that require excessive amounts of time are those that can be satisfied in many different ways. For instance if a request requires N services (in no particular order) there would be $N!$ possible solutions. One realistic request that requires a large amount of time to solve exactly is a request that requires a service with multiple inputs. For example, consider a service that requires 10 input streams, with each input stream being generated in 2 different ways. There would be a total of 1024 possible solutions, causing a sharp increase in the amount of time needed to satisfy such a request. Creative heuristics will be needed if multiple input services, as described, are part of the GM's workload.

4. CONCLUDING REMARKS

In this paper we presented the Graph Manager, a pipeline scheduler that determines, on-the-fly, the best way to satisfy requests using stream modules and existing streams in the system. Our pipeline scheduler is the first one that can exploit multiple input and output stream modules, enabling the GM to satisfy a wide class of requests. Resource-awareness allows the GM to satisfy requests using the modules that consume the least utilized resource, at the time of the request. Finally, while the GM must satisfy a problem that is NP-complete, the computation increases linearly with the complexity of the request.

ACKNOWLEDGMENTS

This material is based on upon work supported by the National Science Foundation under Grant Nos. ANI-9980552, ANI-0070067, 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.

REFERENCES

1. S. McCanne and et al., "Toward a common infrastructure for multimedia-networking middleware," in *ACM Intl Workshop on Network and Operating Systems Support for Digital Audio and Video*, May 1997.
2. W.-T. Ooi, B. Smith, S. Mukhopadhyay, H. Chan, S. Weiss, and M. Chiu, "The Dali multimedia software library," in *Proc. SPIE/ACM Conference on Multimedia Computing and Networking*, January 1999.
3. K. Patel and L. Rowe, "Design and performance of the Berkeley Continuous Media Toolkit," in *Proc. SPIE/ACM Conference on Multimedia Computing and Networking*, February 1997.
4. A. P. Black, J. Huang, R. Koster, J. Walpole, and C. Pu, "Infopipes: an abstraction for multimedia streaming," *Multimedia Systems* **8**, pp. 406–419, December 2002.
5. M. Inc., "Directshow." http://msdn.microsoft.com.
6. M. Vernick, C. Venkatramini, and T. Chiueh, "Adventures in building the Stony Brook Video Server," in *Proc. of ACM Multimedia*, 1996.
7. J. Rexford, S. Sen, and A. Basso, "A smoothing proxy service for variable-bit-rate streaming video." To appear in *Proc. Global Internet Symposium*, December 1999.
8. V. Kahmann and L. Wolf, "A proxy architecture for collaborative media streaming," in *Multimedia Systems*, December 2002.
9. M. K. Bradshaw and et al., "Periodic broadcast and patching services - implementation, measurement, and analysis in an Internet streaming video testbed," in *Proc. of ACM Multimedia System*, 2001.
10. W. T. Ooi, R. V. Renesse, and B. Smith, "Design and implementation of programmable media gateways," in *NOSSDAV*, 2000.
11. T. Fitz, "tgw: A webcast trasncoding gateway," in *Proc. SPIE/ACM Conference on Multimedia Computing and Networking*, January 2003.
12. L. Amini, J. Lepre, and M. Kienz, "Mediamesh: An architecture for integrating isochronous processing algorithms into media servers," in *Proc. SPIE/ACM Conference on Multimedia Computing and Networking*, January 2000.
13. C. Griwodz and M. Zink, "Dynamic data path reconfiguration," in *International Workshop On Multimedia Middleware*, October 2001.
14. E. J. Posnak, H. M. Vin, and R. G. Lavender, "Presentation processing support for adaptive multimedia applications," in *Proc. SPIE/ACM Conference on Multimedia Computing and Networking*, January 1996.
15. S. Roy, J. Ankcorn, and S. Wee, "Architecture of a modular streaming media server for content delivery networks," in *IEEE Intl. Conference on Multimedia and Expo*, July 2003.
16. R. Inc., "Realsystem streaming platform." http://www.realnetworks.com.
17. B. K. Zhuoqing Morley Mao, Hoi-sheung Wilson So, "Network support for mobile multimedia using a self-adaptive distributed proxy," in *ACM Intl Workshop on Network and Operating Systems Support for Digital Audio and Video*, 2001.
18. GStreamer Team, "GStreamer: open source multimedia framework." http://www.gstreamer.net.
19. S. D. Gribble and et al, "The ninja architecture for robust internet-scale systems and services," *Journal of Computer Networks* **35**, March 2001.
20. X. Zhang and et al., "AMPS: A flexible, scalable proxy testbed for implementing streaming services," Tech. Rep. 04-08, Department of Computer Science, University of Massachusetts Amherst, 2004.
21. K. Hua and S. Sheu, "Skyscraper broadcasting: A new broadcasting scheme for metropolitan video-on-demand systems," in *Proc. ACM SIGCOMM*, September 1997.
22. D. Eager and M. Vernon, "Dynamic skyscraper broadcasts for video-on-demand," in *Proc.* 4^{th} *Intl. Workshop on Multimedia Information Systems*, September 1998.
23. L. Gao and D. Towsley, "Supplying instantaneous video-on-demand services using controlled multicast," in *Proc. IEEE International Conference on Multimedia Computing and Systems*, 1999.