AMTrac: Adaptive Meta-caching for Transcoding

Dongyu Liu[1], Songqing Chen[1], and Bo Shen[2]
[1]Dept. of Computer Science
George Mason University
Fairfax, VA 22030
{dliu1, sqchen}@cs.gmu.edu

[2] Mobile and Media Systems Lab
Hewlett-Packard Laboratory
Palo Alto, CA 94304
boshen@hpl.hp.com

ABSTRACT

The increase of aggregate Internet bandwidth and the rapid development of 3G wireless networks demand efficient delivery of multimedia objects to all types of wireless devices. To handle requests from wireless devices at runtime, the transcode-enabled caching proxy has been proposed and a lot of research has been conducted to study online transcoding. Since transcoding is a CPU-intensive task, the transcoded versions can be saved to reduce the CPU load for future requests. However, extensively caching all transcoded results can quickly exhaust cache space. Constrained by available CPU and storage, existing transcode-enabled caching schemes always selectively cache certain transcoded versions, expecting that many future requests can be served from the cache while leaving CPU cycles for online transcoding for other requests. But such schemes treat the transcoder as a black box, leaving little room for flexible control of joint resource management between CPU and storage. In this paper, we first introduce the idea of meta-caching by looking into a transcoding procedure. Instead of caching certain selected transcoded versions in full, meta-caching identifies intermediate transcoding steps from which certain intermediate results (called *metadata*) can be cached so that a fully transcoded version can be easily produced from the metadata with a small amount of CPU cycles. Achieving big saving in caching space with possibly small sacrifice on CPU load, the proposed meta-caching scheme provides a unique method to balance the utilization of CPU and storage resources at the proxy. We further construct a model to analyze the meta-caching scheme. Based on modeling results, we propose *AMTrac*, *A*daptive *M*eta-caching for *Tran*sc*o*ding, which adaptively applies meta-caching based on the client request pattern and available resources. Experimental results show that our proposed AMTrac can significantly improve the system throughput over existing approaches.

Categories and Subject Descriptors

H.4.m [**Information System**]: Miscellaneous

NOSSDAV '06 Newport, Rhode Island USA

General Terms

Algorithms, Experimentation

Keywords

CPU Intensive Computing, Adaptation, Meta-caching, Transcoding

1. INTRODUCTION

With the increase of aggregate Internet bandwidth and the rapid development of wireless networks, Internet accesses from portable devices, such as PDAs and cell-phones, are also growing rapidly. It is not uncommon that users listen to the digital music or watch a football match through their portable devices. However, the increase of these applications [1] also challenges the existing Internet infrastructure, particularly the existing Internet media delivery systems.

Since portable devices generally have different screen sizes, color depths or connection bandwidth from traditional desktop computers, a media object (e.g. a movie) that is good for desktop computers cannot be directly displayed on a PDA. It must be customized appropriately beforehand or at runtime. This type of customization for typical QoS support is often referred to as *content adaptation*.

Two approaches are typically used for providing this type of QoS support in the context of multimedia content delivery. The first approach is called *precoding*. Given any content object, this approach either creates multiple provisioned versions or scalablely encodes the object with multiple layers or descriptions. All the object versions/layers/descriptions are created before they are ever delivered. For example, many content hosts encode their video clips at different bit rate versions, for example, 28/56 Kbps for dial-up clients and 100-plus Kbps for broadband clients [11]. If considering all possible requirements of client devices (not limited to various network speeds), precode demands a huge amount of storage for different versions. The relative advantages of caching precoded versions versus layers are evaluated in [6, 7, 10]. Scalablely-coded content requires less space than multiple individual versions, but it is still not efficient in compression. In addition, content created by this way can only satisfy certain coarse granular QoS requests. It is less flexible when finer granular QoS is required. More importantly, precoding does not scale to the vast variety of media adaptation applications. It may be easy to prevision possible bit rate versions that are required for a streaming application, but it would be difficult to precode content for more generic adaptation tasks such as personalization. In the case of overlaying an end user's logo on a video stream, the hosting server is not likely to have the end user's logo available. Thus precoding in this case is impossible.

The second approach, referred here as *transcoding*, offers on-

line real-time adaptation support. Overall, transcoding is not restricted to customization of content for QoS support. Being real-time and on-line, this approach offers more flexibility and scales well with the variety of the applications. However, transcoding is often computing intensive, especially for multimedia content. Research on developing efficient real-time transcoding algorithms has received much attention [2, 3, 8, 9]. From the system perspective, caching is also a viable technique to achieve computing load reduction [12]. The transcoded result for a request can be cached so that future identical requests can be served without transcoding. This kind of transcode-enabled caching designs has been investigated and the focus has been on efficient utilization of different resources (e.g., CPU, storage, bandwidth) to improve the throughput of the transcoding proxy [13, 15]. All of the existing designs treat the transcoding unit as a black box. The cached data is either the input and/or the output of the transcoder. Specifically, if a transcoded version is fully cached (*full-caching* scheme), identical future requests can be directly served without additional transcoding. However, to cache each transcoded version may quickly exhaust the cache space. On the other hand, if a transcoded version is not cached (*no-caching* scheme), identical requests will result in repetitive transcoding, consuming extensive CPU cycles.

In this paper, we propose a meta-caching scheme in which, intermediate transcoding steps are studied and identified so that appropriate intermediate results (called *metadata*) can be cached. With the cached metadata, the fully transcoded object can be easily produced with a small amount of CPU cycles. Since only metadata is cached, the required cache space is greatly reduced. The saved cache space can be used to store metadata for other transcoding sessions so that, in certain conditions, overall computing load can be reduced. Note that the meta-caching scheme allows the system to achieve a joint control of the CPU and storage resources. It offers a tradeoff point in between what can be achieved by the full-caching and no-caching schemes.

To precisely characterize the meta-caching scheme, we construct an analytical model to investigate the conditional advantages of meta-caching over full- or no-caching quantitatively (modeling details are omitted due to page limit in this submission). Based on the model-driven analysis, we propose a system called AMTrac, which stands for *A*daptive *M*eta-caching for *Tran*s*c*oding. In AMTrac, the meta-caching scheme is adaptively used upon dynamic client accesses and available resources. After capturing real parameters through an implemented prototype, we perform extensive simulation-based experiments to evaluate our proposed scheme. The results show that AMTrac can effectively improve system throughput over existing schemes.

The reminder of the paper is organized as follows. We introduce the generalized meta-caching concept in Section 2, and briefly present modeling results in Section 3. An adaptive meta-caching design based on the model is provided in Section 4 and its performance is evaluated in Section 5. We make concluding remarks in Section 6.

2. PRINCIPLE OF META-CACHING

To illustrate the principle of meta-caching, we first define a storage versus computing space followed by discussions on some practical applications taking advantage from this principle.

2.1 Storage-Computing (SC) Space

Given any media adaptation process, meta-caching is defined as the caching of intermediate results that are created during the course of the adaptation process. As an example, Figure 1(left) defines the flow graph of a certain media adaptation process composed of four computing sub-modules. Caching the intermediate result from each of the four sub-modules leads to skipping that sub-module in the next identical session so that the computing of that sub-module is saved. However, some storage space is required to store the intermediate result. In general, caching the output of each sub-module maps to one point in a space, called *Storage vs. Computing space*. Suppose Figure 1 (right) illustrates such a mapping for this particular process. The vertical axis indicates the amount of storage required to store the intermediate results from any of the sub-modules, relative to storing the final processed results. The horizontal axis indicates the amount of computing load required to create the final result with the help of the intermediate results from any of the sub-modules. This computing load is relative to the computing load when the intermediate result is not available, i.e., the computing load of the full adaptation process. Clearly, if the intermediate result of any sub-module is directly available (from a cache, for example) instead of computing it from the input, the computing load required to create the final output is smaller.

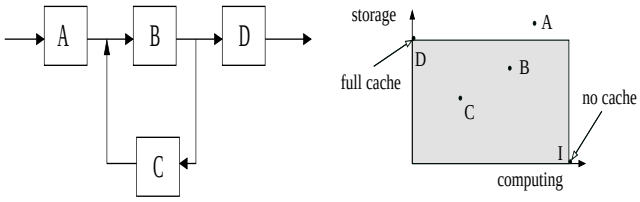

Figure 1: an example processing flow and the corresponding mapping in the SC space

Now we discuss some of the specific points within the SC space. If nothing from the flow is cached, the point I (no cache) indicates no storage requirement. But 100% of the CPU is required when a final adapted output is requested. On the other hand, if the final output is cached (full cache), 100% of the storage is required while no computing is needed. Note that point D and point I define a shaded area in this SC space. In general, any point that falls outside the shaded area has no advantage over either D or I. For example, point A is outside the shaded area, which indicates that storing the intermediate result from A would cost more storage than storing the result from D. Obviously, this is not as efficient as simply storing the final result.

Point B and C can introduce certain advantages since both points indicate reduced storage requirement at a cost of some computing load. In general, a point closer to the origin of the SC space presents better advantage since it indicates less computing and less storage requirement to obtain the final results. In this example, point C is a better choice. In other words, point C indicates that sub-module C is more computing intensive yet its intermediate result requires less storage. In MPEG transcoding, they can be mapped to the caching of sequence-level and pixel-level metadata, as the former requires less storage space and more computing than the latter.

Although we illustrate this principle through an arbitrary example, it is clear that the principle is general. Once any media adaptation process is defined, a map in the SC space that reflects the storage and computing tradeoff is uniquely obtainable. We will next discuss some real applications in this context.

2.2 Applications

The principle outlined above has many practical applications. In particular, video transcoding is the most common type of media adaptation application.

Figure 2 illustrates the processing flow of the bit rate reduction

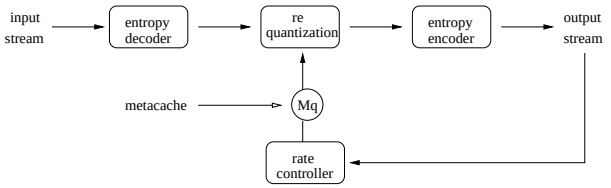

Figure 2: Bit Rate Reduction

of a MPEG video, a case of transcoding. With careful selection of the intermediate points within a transcoding process pipeline, meta-caching can be very useful in reducing the aggregated computing load of an adapting server servicing a ground client requests with different access patterns. In this case, caching of the requantization scale factor (M_q) could achieve a good tradeoff between storage and computing resource utilization.

The meta-caching principle is not just restricted to video transcoding applications. To name a few other types of media adaptation applications that can benefit from this principle, we consider the following examples.

- Video to keyframe conversion. Instead of a full length video, a sequence of representative keyframes from the video can be delivered in situations when a client does not have a video player available. This conversion consists of a keyframe analysis process followed by the assembling of the keyframes. Keyframe analysis detects scene changes within a video sequence and identifies frames that are representative of the scene. This can be very computing intensive. However, if the intermediate result (e.g., the frame index of the keyframes) of the keyframe analysis module can be stored, which costs very little storage, the computing load can be significantly reduced for future sessions.

- Personalized logo insertion. With a customized logo inserted into each frame of a video, a client can personalize his/her own content. In this process, the compressed video is first adapted with the logo insertion area to be independently coded. Then a logo is inserted. The adaptation from dependently coded original video to independently coded area can be computing intensive. If the converted independent area can be stored, future sessions can be free of significant computing load. On the other hand, an independently coded area (for example, four significantly smaller corner areas) costs less storage than the full logo inserted video.

- Privacy protection. In this scenario, certain features from content (e.g., certain faces in a video) are automatically blocked to protect privacy. The intermediate results from face detection (e.g., face location on each frame) can be stored so that future sessions can be relieved from the computing intensive face detection process.

The meta-caching principle is often useful for adapting multimedia content since this kind of process tends to be computing intensive. However, for any real-time content processing service, if we can identify from its processing flow a point in the SC space that can benefit from the meta-caching principle, it is possible to improve the overall system performance by balancing the use of storage and computing resources. Since overall system performance depends on aggregated client access behavior, certain points in the SC space can be more advantageous than others, given available computing and storage resources. In the next section, we will model further details to determine the performance of meta-caching.

3. PERFORMANCE MODELING

We model the performance of the meta-caching scheme and compare it with full-caching and no-caching schemes. Our modeling quantitatively characterizes the conditional advantage of full-, meta- and no-caching schemes over the other two under different conditions of available resources and client access patterns.

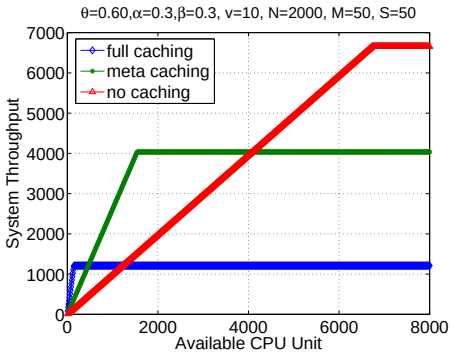

Figure 3: Comparisons between three methods: available CPU ranges up to 8000

Due to page limit, we omit the modeling details. Figure 3 shows the comparison results with some typical values for different parameters when the available CPU varies. In these figures, we set θ to be 0.6, and set α and β as 0.3 (refer to Table 1). We assume that there is a total of 50 objects and a total of 2000 accesses. Each object has 10 versions. The total available CPU unit is varying. The figure indicates that under certain conditions, one of the three schemes may outperform the other two. A system aiming to maximize system throughput should thus adapt to the best scheme upon dynamic client access behaviors and resource availability. To achieve this goal, in the next section, we present the *A*daptive *M*eta-caching design for *Tran*scode-enabled proxy, called AMTrac.

4. DESIGN OF AMTRAC

As aforementioned, AMTrac should work adaptively perform transcoding based on the client access pattern and available resources dynamically. Thus, it is important to monitor the dynamically changing system resource availability and the client access pattern. In this section, *Resource-driven Replacement Policy* and *Reactive Cache Adjustment* are designed to work with *Progressive Request Admission* for these purposes.

Assume the proxy provides v different versions of an object to client requests in AMTrac. Each object version is either cached with its metadata only or cached with its fully transcoded result. Thus, in AMTrac, the cache space is logically split into two parts. One is for caching metadata of different object versions, and the other is to cache the fully transcoded object versions. The size of each part changes dynamically. AMTrac keeps track of the requested object version, even after an object version is evicted. We use the following data structure for each object to record important runtime information to assist the implementation of a proposed strategy.

In this structure, P_s is calculated as the ratio of the *active disk size* over the *total disk size*. The *active disk size* is the sum of the size of all active (requested) cached objects at present. The *total disk size* is the total available cache size. Based on these, the following policies work together to implement AMTrac.

4.1 Progressive Request Admission

Table 1: Fields of Data Structure

field	information
α	storage for caching metadata (relative to full result)
β	CPU for producing the fully-transcoded object from metadata (relative to no cache)
v	the total number of versions of each object
$ms[1..v]$	an array to record the metadata size of each version
$fs[1..v]$	an array to record the full data size of each version
$r[1..v]$	an array to record the references to each version
$s[1..v]$	object status with 0 for being replaced, 1 for cached
$U[1..v]$	the utility value of the object version
P_s	current storage utilization (%)
P_c	current CPU utilization (%)

Upon a client request for the j^{th} version of an object, there are three cases as follows.

- If the requested object version j is fully cached, the request is directly served and the corresponding data item, $r[j]$, is increased by 1.
- If the requested object version is cached with its metadata, the request is served with the online meta-transcoding (transcoding based on metadata). The transcoded result is sent to the client. The corresponding data item, $r[j]$, is increased by 1. The reactive cache adjustment (see section 4.3) is activated to determine whether this fully transcoded object version should be cached or not.
- If the requested object version does not exist in the cache:
  - If the object version is being accessed for the first time, the proxy performs online transcoding to produce the object version j. Correspondingly, the metadata is cached. The corresponding data item, $r[j]$, is increased by 1. $ms[j]$ gets updated. The status of this object version $s[j]$ is set to 1. If there is insufficient cache space, the replacement policy (see section 4.2) is activated to make room for its caching.
  - If the object version has been accessed and its status is replaced ($s[j] = 0$), the object version is transcoded again and sent to the client. $r[j]$, is increased by 1. The reactive cache adjustment (see section 4.3) is activated to determine whether the corresponding metadata should get cached.

4.2 Resource-driven Replacement Policy

When there is not enough cache space, replacement is activated. To maximize cache performance, clearly, it is important to select the right victim.

In our design, a utility based policy is used to select the right victim. The utility function is designed as follows:

$$U(i_j) = \begin{cases} \frac{r_j}{S_j} \times \frac{\alpha_j}{\alpha_j+\beta_j} & \text{if } P_c \geq P_s, \\ \frac{r_j}{S_j} \times \frac{\beta_j}{\alpha_j+\beta_j} & \text{if } P_c < P_s. \end{cases} \quad (1)$$

In equation 1, S_j indicates the occupied cache space of this object version, where $S_j = max(ms[j], fs[j])$; r_j is the reference number to this object version; α_j is the storage unit (in percentage) used for caching the metadata of version j, while β_j is the CPU unit to transcode to the final version j; P_c represents the current CPU utilization, and P_s indicates the current storage utilization. Thus, in this equation, $\frac{r_j}{S_j}$ considers that if an object version is more popular with a unit storage, the object version should have a high utility value and has higher chance to be cached. If $P_c > P_s$, the current system is CPU constrained, so the utilization of the storage is encouraged. If $P_c < P_s$, the system is storage constrained, and the utilization of the CPU is encouraged.

Thus, the utility of an object version considers both the object popularity and the current available resources to find the least valuable object version. Each time the replacement policy is activated, all cached object versions must refresh their utility. Based on the utility function, we design the replacement policy as follows.

When the replacement policy is activated, it compares all the object versions in the cache based on their utility values. The one with the minimum value is selected as the victim and the following procedure loops until sufficient space is found.

- If the selected victim has its fully transcoded data cached in the proxy, the system selects to evict its fully cached data and caches its meta data (it consumes a bit more CPU and it is done when the next time a request is received for this object version). $fs[j]$ is set to 0, while $ms[j]$ is updated accordingly.
- If the selected victim has only metadata cached, the metadata is evicted, $ms[j]$ is set to 0. $r[j]$ is set to 0. The corresponding object version status, $s[j]$, is also set to be replaced.

4.3 Reactive Cache Adjustment

According to our design, for each object version, there could be three possible cases, namely replaced, with metadata cached, or with fully transcoded result cached. To accommodate the dynamic client accesses and thus to maximize the system performance, we design the reactive cache adjustment policy upon different situations. This adjustment is passive since it is always invoked due to new client requests.

- If a currently replaced object version is accessed, the system starts to evaluate whether the current utility of this object version is increased and is large enough to get cached. The new utility is calculated assuming the object version consumes $ms[j]$ for storage. If the utility is larger than the utility of any cached one, the metadata of this object version gets cached and $ms[j]$ and $u[j]$ are set accordingly. The replacement policy is activated if needed.
- If an object version is cached with metadata, the system starts to evaluate whether the fully transcoded data of this object version should be cached or not. Assuming the fully transcoded object version is cached using space $fs[j]$, its corresponding utility value is compared with the current cached object versions. If the new utility of this object version is higher than that of any cached one, the fully transcoded object version gets cached. Its $fs[j]$ and $u[j]$ are updated accordingly. Correspondingly, its metadata gets evicted and $ms[j]$ is set to 0. Additional space is reclaimed with the assistance of the replacement policy when necessary.

5. PERFORMANCE EVALUATION

In this section, we first perform real experiments on a rate reduction transcoding application to capture the practical values of α and β to set up simulations. Then, we run simulations to study the performance of different strategies for rate reduction. To further compare the different strategies when conditions vary, we also evaluate different strategies in a general context, the result of which is omitted due to page limit.

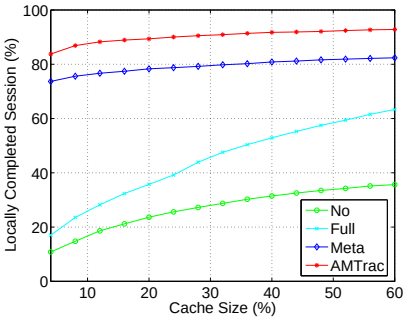

Figure 4: Total completed sessions

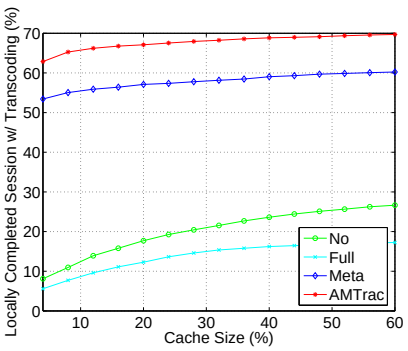

Figure 5: Completed sessions with transcoding

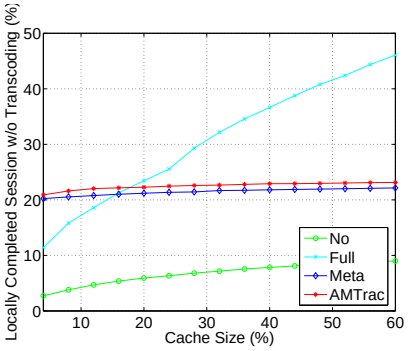

Figure 6: Completed sessions without transcoding

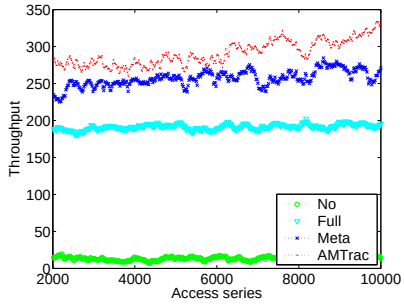

Figure 7: Throughput along time (4% cache size)

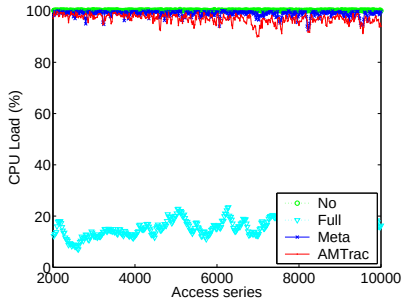

Figure 8: CPU load along time (4% cache size)

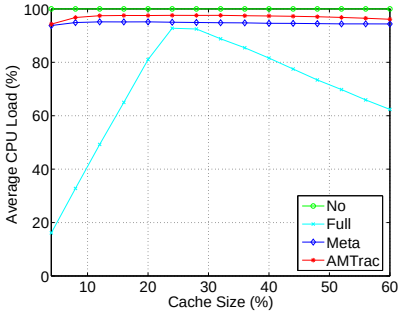

Figure 9: Average CPU load when cache size varies

5.1 Experimental Parameter Capturing and Simulation Setup

As explained in section 2, bit rate reduction is a typical transcoding process where meta-caching can be applied to cache M_q. To capture the real setup for the storage usage (α) and CPU (β), we implemented a full transcoder and a meta transcoder in C with no special optimizations. Through transcoding trail runs on HP X4000 workstation with 2 GHz Intel Xeon CPU, we compare the CPU time used by full transcoding and meta transcoding on MPEG test sequences with spatial resolution 352x240 coded at 25 fps. The original video contains I-, P- and B-pictures and is coded at 512 Kbps.

Considering the bit rate reduction transcoding, caching M_q costs storage at 8250 byte/sec. However, bypassing the rate control submodule enables the transcoder to get the final result with only 45% of the computing load comparing to a full session. With a target bit rate at 128Kbps, the selection of this meta-caching point represents (α, β) as $(0.5, 0.45)$. Note that in the rate reduction case, the metadata size does not change with the bit rate reduction. Thus, the corresponding α for version 1, 2, and 3 are 0.167, 0.25, and 0.5. The β values for different versions are the same (this is unique to rate reduction adaption).

Based on these parameters from a real transcoding application, we conduct large scale simulation based experiments. In these simulations, we assume the total available CPU is 100 units, and object popularity follows a Zipf-like distribution with a skew factor (θ) set to 0.73 [4, 5]. We use MediSyn [14] to generate the initial workload. In the synthetic workload, there is a total of 1000 objects. Each workload contains 20,000 client requests, the access duration of which ranges from 5 to 40 minutes. For each object, there are 4 different versions, including the original best quality object – version 0. Version 3 represents the lowest quality version. We assume the storage for version 1, 2 and 3 is 3/4, 2/4, 1/4 of the original object size (version 0). Their corresponding encoding rates are 512 Kbps, 384 Kbps, 256 Kbps, and 128 Kbps. The total unique original object size amounts to 89.9 GB. The total traffic is 1205.8 GB and the access duration lasts about 34 hours.

5.2 Experiments on Rate Reduction Application

The three major evaluation metrics used in these experiments are *Locally Completed Session*, *Throughput*, and *Average CPU Load*. *Locally Completed Session* represents overall system performance and is the ratio of the number of accesses that are served locally (on the proxy) over the total number of accesses. *Locally Completed Session* consists of two parts. One is the *Locally Completed Session without Transcoding*, which corresponds to the scenario that the requested version is cached. The other is the *Locally Complete Session with Transcoding*, which indicates the scenario where the transcoding (full or metadata-based) is necessary to serve the client request. *Throughput* represents system throughput along the time line. It is calculated as the number of sessions that the transcoding proxy can handle per time unit. *Average CPU Load* is defined as the ratio of the sum of current CPU load and the total CPU capacity. It can indicate whether the CPU is saturated due to transcoding load.

Figure 4, Figure 5, and Figure 6 show the Locally Completed Session and its two components – with and without transcoding, for four different methods when the cache size increases. In these figures, *No*, *Full*, *Meta* represents the no-caching, full-caching, and

meta-caching methods we have discussed. *AMTrac* represents our proposed scheme.

As shown in Figure 4, when considering the total completed sessions in the transcoding proxy, the performance of the four methods is ordered in *AMTrac*, *Meta*, *Full*, and *No*. Figure 5 shows that the Locally Completed Session with Transcoding for different methods. The trend is similar to the Locally Complete Session in total, indicating that the transcoding results for meta-caching and AMTrac are dominant in the totally completed sessions. An interesting variation is shown in Figure 6 when considering the Local Completed Session without Transcoding. As indicated in the figure, when the cache size increases beyond 16%, *Full* outperforms *Meta* and *AMTrac*. This is due to more cached objects in *Full* when the cache size increases.

Having examined the overall performance in terms of the total completed sessions in the transcoding proxy, now we examine the proxy's performance at each time unit. Figure 7 shows the system throughput along the client accesses when the cache size is 4%. Note in the figure, only the results between access 2000 and access 10000 are shown. As indicated in the figure, AMTrac outperforms all other methods. Among all the four methods, no-caching achieves the worst performance as expected and full-caching also has worse performance compared to meta-caching and AMTrac.

Figure 8 shows the corresponding CPU load along the client accesses (time). Apparently, besides full caching, the other three methods always have a 100% CPU load or close to 100%. To full-caching, since 4% cache space is far from sufficient, its CPU is under utilized. Figure 9 further shows the average CPU load for the four methods when the available cache size varies from 4% to 60%. As expected, the CPU load of no-caching is not affected when the cache size increases. Since full-caching is the mostly affected by the available cache space, its average CPU load decreases after the cache size increases beyond 28%. Before that, the limited cache space results in frequent replacement upon new client requests, some of which cannot happen because the selected victim objects are being accessed. Thus, although there are spare CPU cycles, they are not utilized. After the cache size is increased beyond 28%, this situation is relieved. Since full-caching is storage constrained, with larger cache space, more client requests could be served from cache without repetitive transcoding. This is evidenced by the decreasing order of CPU load of full-caching when the cache size is beyond 28%. From another aspect, although meta-caching and AMTrac have a higher CPU load with the increase of cache size, throughput of these two approaches is higher than that of full-caching.

6. CONCLUSION

The Internet has witnessed the rapid increase of Internet media contents and widespread use of portable devices in the past a few years. While transcoding proxy has been proposed and researched extensively, existing strategies generally aim at reducing server load and server traffic. Little attention has been paid to transcoding procedure itself and that leads to less flexibility in addressing the tradeoffs between computing and storage constraints. By proposing to study inside a transcoding process itself, we outline a new approach of caching strategy design with the main focus on computing load reduction. A meta-caching scheme is proposed that offers a new point of control in the computing and storage space. With model-based analysis on the meta-caching scheme, we propose an adaptive meta-caching system, called AMTrac, which can adaptively use the meta-caching scheme based on client accesses and available resources in the system. Experiments show that it significantly outperforms existing strategies.

7. ACKNOWLEDGMENT

We would like to thank William L. Bynum, Xiaodong Zhang, and anonymous reviewers for their helpful comments on this paper. The work is supported by NSF grant CNS-0509061 and a grant from Hewlett-Packard Laboratories.

8. REFERENCES

[1] Support nationwide delivery of mobile multimedia. http://www.qualcomm.com/press/releases/2004/041101_mediaflo_700mhz.html.

[2] S. Acharya and B. C. Smith. Middleman: A video caching proxy server. In *Proceedings of ACM NOSSDAV*, Chapel Hill, NC, 2000.

[3] E. Amir, S. McCanne, and H. Zhang. An application level video gateway. In *Proceedings of ACM Multimedia*, San Francisco, CA, November 1995.

[4] L. Cherkasova and M. Gupta. Characterizing locality, evolution, and life span of accesses in enterprise media server workloads. In *Proceedings of ACM NOSSDAV*, Miami, FL, May 2002.

[5] M. Chesire, A. Wolman, G. Voelker, and H. Levy. Measurement and analysis of a streaming media workload. In *Proceedings of the 3rd USENIX Symposium on Internet Technologies and Systems*, San Francisco, CA, March 2001.

[6] P. D. Cuetos, D. Saparilla, and K. W. Ross. Adaptive streaming of stored video in a tcp-friendly context: Multiple versions or multiple layers? In *Proceedings of Packet Video Workshop*, Kyongju, Korea, April 2001.

[7] F. Hartanto, J. Kangasharju, M. Reisslein, and K. W. Ross. Caching video objects: layers vs versions? In *IEEE International. Conf. on Multimedia and Expo*, Lausanne, Switzerland, August 2002.

[8] C. K. Hess, D. Raila, R. H. Campbell, and D. Mickunas. Design and performance of mpeg video streaming to palmtop computers. In *Proceedings of SPIE/ACM MMCN*, San Jose, CA, January 2000.

[9] C-W. Lin J. Xin and M-T. Sun. Digital video transcoding. In *Proceedings of IEEE*, volume 93(1), pages 84–97, Jan. 2005.

[10] T. Kim and M. H. Ammar. A comparison of layering and stream replication video multicast schemes. In *Proceedings of ACM NOSSDAV*, Port Jefferson, NY, June 2001.

[11] R. Mohan, J.R. Smith, and C.S. Li. Adapting multimedia internet content for universal access. In *IEEE Transactions on Multimedia*, volume 1 (1), March 1999.

[12] R.Rejaie and J. Kangasharju. Mocha: A quality adaptive multimedia proxy cache for internet streaming. In *Proceedings of ACM NOSSDAV*, Port Jefferson, NY, June 2001.

[13] B. Shen, S. Lee, and S. Basu. Caching strategies in transcoding-enabled proxy systems for streaming media distribution networks. In *IEEE Transactions on Multimedia*, volume 6, pages 375–386, April 2004.

[14] W. Tang, Y. Fu, and L. Cherkasova. Medisyn: A synthetic streaming media service workload generator. In *Proceedings of ACM NOSSDAV*, Monterey, CA, June 2003.

[15] X. Tang, F. Zhang, and S. T. Chanson. Streaming media caching algorithms for transcoding proxies. In *Proceedings of the 31st International Conference on Parallel Processing (ICPP)*, Vancouver, Canada, August 2002.