Load Balancing in a Cluster-Based Web Server for Multimedia Applications

Jiani Guo, *Student Member*, *IEEE*, and Laxmi Narayan Bhuyan, *Fellow*, *IEEE*

Abstract—We consider a cluster-based multimedia Web server that dynamically generates video units to satisfy the bit rate and bandwidth requirements of a variety of clients. The media server partitions the job into several tasks and schedules them on the backend computing nodes for processing. For stream-based applications, the main design criteria of the scheduling are to minimize the total processing time and maintain the order of media units for each outgoing stream. In this paper, we first design, implement, and evaluate three scheduling algorithms, First Fit (FF), Stream-based Mapping (SM), and Adaptive Load Sharing (ALS), for multimedia transcoding in a cluster environment. We determined that it is necessary to predict the CPU load for each multimedia task and schedule them accordingly due to the variability of the individual jobs/tasks. We, therefore, propose an online prediction algorithm that can dynamically predict the processing time per individual task (media unit). We then propose two new load scheduling algorithms, namely, Prediction-based Least Load First (P-LLF) and Prediction-based Adaptive Partitioning (P-AP), which can use prediction to improve the performance. The performance of the system is evaluated in terms of system throughput, out-of-order rate of outgoing media streams, and load balancing overhead through real measurements using a cluster of computers. The performance of the new load balancing algorithms is compared with all other load balancing schemes to show that P-AP greatly reduces the delay jitter and achieves high throughput for a variety of workloads in a heterogeneous cluster. It strikes a good balance between the throughput and output order of the processed media units.

Index Terms—Online prediction, partial predictor, global predictor, adaptive partioning, prediction-based load balancing, out-of-order rate.

1 Introduction

Several applications over the Internet involve processing of secure, computation-intensive, multimedia, and high-bandwidth information. Many of these applications require large-scale scientific computing and high-bandwidth transmission at the server nodes. The current generation of Internet servers is mostly based on either a general-purpose symmetric multiprocessor or a cluster-based homogeneous architecture. As we attempt to scale such servers to high levels of performance, availability, and flexibility, the need for more sophisticated software architectures becomes obvious. Additionally, contemporary distributed architectures have limited abilities to handle overloads, load imbalances, and compute-intensive transactions like cryptographic applications and multimedia processing. In this paper, we consider a scalable distributed system architecture, shown in Fig. 1, where the major functionalities required in the Internet servers (SSL, HTTP, script and cryptographic processing, database management, multimedia processing, etc.) are partitioned into parallel tasks and backend computing servers are allocated based on their needs.

In this paper, we consider multimedia processing as the example. Since Internet clients may vary greatly in their hardware resources, software sophistication, and quality of connectivity, different clients require different media streaming service. A promising solution to the problem is to use transcoding to customize the size of objects and distribute the available network bandwidth among various clients [1]. This method is called on-demand transcoding, which is used to convert a multimedia object from one form to another. On-demand transcoding (distillation) has been proposed to transform media streams in the active routers [2], [3], [4] or proxy servers [5], [6], [7] to adapt media streams to fluctuating network conditions. Any client intending to request a media stream first contacts the media server, as shown in Fig. 1. If the media data in storage satisfies the requirements, the media server supplies the data. If on-demand transcoding is needed, the media server retrieves data, divides them into several tasks, and distributes the tasks among computing servers for transcoding. The transcoded data is transferred back to the media server and then delivered to the clients. Due to the variety of clients, different streams may require different transcoding operations and, therefore, produce a variety of individual transcoding jobs to be scheduled among the computing servers. In order to provide real-time transcoding service, a good scheduling algorithm needs to be employed that can predict the CPU load for each job and then schedule accordingly.

Load balancing is a critical issue in parallel and distributed systems to ensure fast processing and good utilization. A detailed survey of general load balancing algorithms is provided in [8]. Although a plethora of load-balancing schemes have been proposed, simple static policies, such as random distribution policy [9] or modulus-based round-robin policy [10], are adopted in practice because they are easy to implement. However, these schemes do not work well for heterogeneous processors or variation in the task processing times. On the other hand, adaptive load balancing policies are usually complicated

- *J. Guo is with Cisco Systems, Inc., 170 West Tasman Dr., San Jose, CA 95134. E-mail: jianiguo@cisco.com.*
- *L.N. Bhuyan is with the Department of Computer Science and Engineering, University of California, Riverside, CA 92521. E-mail: bhuyan@cs.ucr.edu.*

Manuscript received 16 Apr. 2004; revised 23 Feb. 2005; accepted 22 Dec. 2005; published online 26 Sept. 2006.
Recommended for acceptance by J. Srivastava.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-0099-0404.

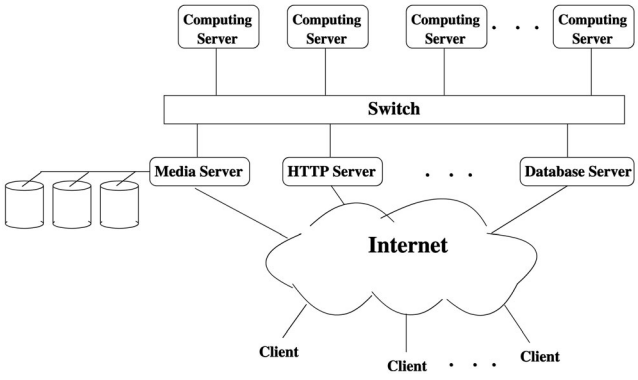


Fig. 1. Cluster-based Web server architecture.

and require prediction of computation time for any incoming requests [11]. They are difficult to implement, and produce increased communication overhead due to feedback requirements from the processors. Moreover, the load balancing techniques proposed for general parallel systems cannot be directly applied to our media cluster because there are additional requirements like reducing jitter. Jitter is defined as the standard deviation of the interdeparture time among media units. High jitter is detrimental to the playback quality, which is the main concern of media clients. The interdeparture time among units of a multimedia stream is reduced through parallel transcoding on the computing servers in the cluster. That gives rise to an increase in the out-of-order departure of the packets, thus producing high jitter. Hence, proper scheduling with a good load balancing algorithm must be designed to deliver a suitable balance between high throughput and low jitter. A few researchers have developed load scheduling algorithms for cluster-based Web servers. Zhu et al. proposed an elegant scheduling algorithm to provide differentiated service to multiple service classes of generic Web requests [12]. Li et al. [13] implemented a generalized Web request distribution system called Gage. A Least Load First (LLF) policy is employed, where the server with the least load is chosen to process a request. However, their work focuses on generic Web requests instead of the multimedia jobs considered in this paper.

We have designed and proposed a set of scheduling algorithms for parallel multimedia applications. In order to develop and evaluate the proposed load balancing schemes, we implement a Linux-based media cluster over Gigabit Ethernet and develop a multithreaded software architecture to schedule multimedia jobs for transcoding in the cluster. A multithreaded software architecture can overlap communication with computation and can achieve maximum efficiency. We make the following contributions in this paper:

1. We implement a media cluster and do experiments to compare the performance of three load balancing algorithms, namely, First Fit (FF), Stream-based Mapping (SM), and Adaptive Load Sharing (ALS). The SM and ALS schemes were designed by us [14], and are described in detail in Section 2. We also present more results in this paper for a number of movies with different transcoding requirements.
2. The above algorithms are based on the average computational requirement of a multimedia unit. Since the computation may vary from time to time, we design a prediction algorithm in Section 3 to dynamically predict the transcoding time for each media unit. The predicted time is used to distribute the incoming workload to computing servers accordingly.
3. Incorporating the prediction scheme, we propose three new load balancing algorithms in Section 4, namely, Prediction-based Least Load First (P-LLF), Adaptive Partitioning (AP), and Prediction-based Adaptive Partitioning (P-AP). P-LLF extends the LLF algorithm using prediction. Adaptive Partitioning (AP) is a new algorithm that reduces jitter by dynamically mapping each stream to a subset of servers. P-AP extends AP by employing prediction to compute the requirement.
4. We do experiments to compare the performance of an above seven load balancing algorithm, namely, FF, SM, ALS, LLF, P-LLF, AP, and P-AP. Experimental results are presented in Section 5 to show that the prediction-based algorithms produce better throughput and less out-of-order departure for a number of media streams with different requirements.

2 Nonprediction-Based Load Balancing Techniques

In this section, we present three nonprediction-based load balancing algorithms that we have developed for a media cluster. Preliminary results applying only one transcoding operation were presented earlier by us in [14]. We extend the algorithms to different transcoding operations.

2.1 First Fit (FF)

With First Fit, the media server searches for an available computing server in a round-robin way when scheduling media units. It always chooses the first available one to dispatch a media unit. To avoid collecting feedback from servers, we build a dispatch queue on the media server for each computing server and let these dispatch queues be indicators of their load status.

To schedule a media unit, the dispatch queues are polled in a *round-robin* way. The unit is scheduled to the first computing server whose corresponding queue has a vacancy. If all queues are full, overload is indicated on all servers and the unit will not be scheduled until one of the queues is drained. The load on a server is either affected by the complexity of the transcoding operations or the sizes of the units. The server with the higher load usually drains its dispatch queue slower and leaves fewer vacancies in the queue. Consequently, heavily loaded servers are more likely to get fewer media units for processing than the lightly loaded servers. Therefore, the loads among servers are naturally balanced to some extent. It is a nice way to take into account heterogeneous servers without the help of an extra load analyzer. But, the media units of the same stream are most likely distributed to different servers, resulting in high delay jitters for each stream at its destination. However, as shown in [14], FF

generates higher throughput than the simple round-robin scheme presented in [3].

2.2 Stream-Based Mapping (SM)

The problem with FF is that media units are distributed to many servers which causes large out-of-order delivery of the units. To preserve the computation order among media units, as well as to keep the simplicity of FF, a stream-based mapping algorithm can be employed. The unit is mapped to a server according to the function $f(c) = c \bmod N$, where c is the stream number to which the unit belongs and N is the total number of servers in the cluster. Therefore, all the units belonging to one stream will be sent to the same server. We have shown in [14] that this scheme works most efficiently in a cluster with homogeneous servers and for some specific input patterns. Assuming there are M streams and N servers, input workload must satisfy the condition $M \geq N$ and M is multiple of N.

2.3 Adaptive Load Sharing (ALS) Policy

A number of adaptive load sharing policies are proposed in the literature [8]. However, we are unaware of any real implementation because of the complexity and overhead of the ALS algorithms. Our analysis indicated that the extended HRW technique [15], proposed for network applications, offers a reasonable balance between the throughput and out-of-order departures. While aiming at delivering high throughput per flow, the ALS policy also minimizes the probability of units belonging to the same flow being treated by different processors and so minimizes the out-of-order rate. Hence, we implemented it to schedule multiple media streams in our system, as described below.

According to the ALS policy, a media unit can be mapped to a particular server according to the function $f(\vec{v}) = j$, which is defined as

$$x_j \; g(\vec{v}, j) = max_{k \in \{1,\ldots,N\}} x_k \; g(\vec{v}, k), \tag{1}$$

where v is the identifier vector of the unit which helps identify a particular flow the unit belongs to, j is the server node to which the unit will be mapped for processing, $g(\vec{v}, j)$ is a pseudorandom function which produces random variables in (0,1) with uniform distribution, and $(x_1, x_2, \ldots x_N)$ is a weight vector that describes the processor utilization for each server. The weight vector, $(x_1, x_2, \ldots x_N)$, is dynamically adapted according to the system behavior through periodic feedback. Here is how the adaptation works. The media server periodically gathers information from each server about its utilization and calculates to see if the adaptation threshold is exceeded. If the threshold is exceeded, the media server adjusts the weights. In the feedback report, a smoothed, low-pass filtered processor utilization measure of the following form is used to calculate the utilization of each server $\bar{\rho}_j(t)$ by gathering the load statistics information $\rho_j(t)$ periodically:

$$\bar{\rho}_j(t) \; = \; \frac{1}{r} \; \rho_j(t) + \frac{r-1}{r} \; \bar{\rho}_j(t - \Delta t). \tag{2}$$

Similarly, the total system utilization is measured as $\bar{\rho}(t) = \frac{1}{r}\rho(t) + \frac{r-1}{r}\bar{\rho}(t - \Delta t)$. The adaptation algorithm consists of triggering policy and adaptation policy. Once the triggering condition is reached, adaptation will be taken to the weights of involved servers.

To implement the ALS algorithm, there are two issues left open to implementers. One is the pseudorandom function $g(\vec{v}, j)$. Kencl an Boudec [15] suggest to implement function $g(\vec{v}, j)$ using the hash function $h_{\phi^{-1}}(y) = (\phi^{-1} y) \bmod 1$, which is based on the Fibonacci golden ratio multiplier $\phi^{-1} = (\sqrt{5} - 1)/2$, such that

$$g(\vec{v}, j) = h_{\phi^{-1}}\big(\vec{v} \;\; \text{XOR} \;\; h_{\phi^{-1}}(j)\big). \tag{3}$$

The other open issue is how to measure the load of each processor.

In our experiments, we adopt the above function $g(\vec{v}, j)$, and define the load indicator $\rho_j(t)$ as

$$\rho_j(t) = ttask_j / \Delta t, \tag{4}$$

where $ttask_j$ is the CPU time spent by the transcoding services during the polling interval Δt. $\rho(t)$ is defined as

$$\rho(t) = \left(\sum_{j=1}^{N} ttask_j \right) / N\Delta t = \frac{1}{N} \sum_{j=1}^{N} \rho_j(t). \tag{5}$$

The identifier v is chosen to be the stream number of the media unit. Therefore, during each monitoring epoch, the mapping function (1) is calculated and a static mapping between the streams and the servers is determined. When a change in load distribution is reported by the computing servers at the end of an epoch, the weight vector is changed and the mapping is adjusted to rebalance the loads among servers. The new mapping takes effect in the next epoch.

We have shown in [14] that ALS reduces departure jitter for multiple streams. In spite of high overheads to collect feedback information, the ALS scheme produces a good throughput. More results and comparisons are given in Section 5 of this paper.

3 Predicting Processing Time

In the load balancing algorithms presented so far, the variability of individual jobs has not been explicitly considered. As to the fact that the transcoding time of media units in a stream or among streams varies a lot due to the wide variation in scenes and motion relations, the ability to predict how much CPU load a job may consume is essential for building a good scheduling scheme. In the cluster-based Web server system Gage [13], the CPU time consumed by client requests is predicted as the *weighted average time* of the processed requests. The prediction is used to predict the load on each server and to distribute the incoming workload among servers according to the Least Load First (LLF) policy. However, such a simple prediction scheme is not suitable for multimedia transcoding because the transcoding time differs among transcoding operations even for the same stream.

During the past decade, a number of video-coding standards have been developed for communicating video data. These standards include MPEG-1 for playback from CD-ROM, MPEG-2 for DVD and satellite broadcast, and H.261/263 for videophone and video conferencing. Newer video coding standards, such as MPEG-4, also emerged. For all these video-coding standards, four video resolution criteria are usually used in commercial products, as

TABLE 1
Four Resolution Criteria in MPEG Specification

Rate	Video Size	Frame Rate (fps)	Bit Rate (kbps)
SQCIF	128×96	15	50
QCIF	176×144	15	70
CIF	352×288	26	100
4CIF	704×576	30	200

illustrated in Table 1. Given these video resolution criteria, the common transcoding operations fall into three types: changing the bitrate, resizing frames, and changing the frame rate among the four resolution criteria.

Most data streaming formats contain periodic zero-state resynchronization points for increased error resilience, effectively segmenting the stream into independent blocks, which we call media units [4]. For instance, in an MPEG-1/2 stream, a media unit can be a group of pictures (GOP) that is decoded independently. Since most transformations maintain the independence of media units, the transformation of a single media unit can be considered an independent processing job which can be scheduled onto any computing server in the cluster. The only interjob dependence is the processing order of consecutive media units in the same media stream.

Bavier et al. [16] built a model to predict the MPEG decoding time at the frame level. They found that it is possible to construct a linear model for MPEG decoding with R^2 values of 0.97 considering both frame type and size. In statistics, the correlation coefficient R indicates the extent to which the pairs of numbers for two variables lie on a straight line. The strength of the relationship between X and Y is theoretically expressed by squaring the correlation coefficient R and multiplying by 100, which is known as variance explained R^2. For example, a correlation R of 0.9 means $R^2 = 0.81 \times 100 = 81\%$. Hence, 81 percent of the variance in Y is "explained" or predicted by the X variable. Experimental results [16] show that the model can be used to predict the execution time to within 25 percent of the time actually taken to decode frames. In this section, we first model the relationship between the transcode time and the unit size by statistically analyzing a set of experimental results for a specific movie and a specific transcoding operation. Based on the model, we develop a prediction algorithm to dynamically predict the transcode time of a media unit.

TABLE 2
MPEG Movies for Transcoding

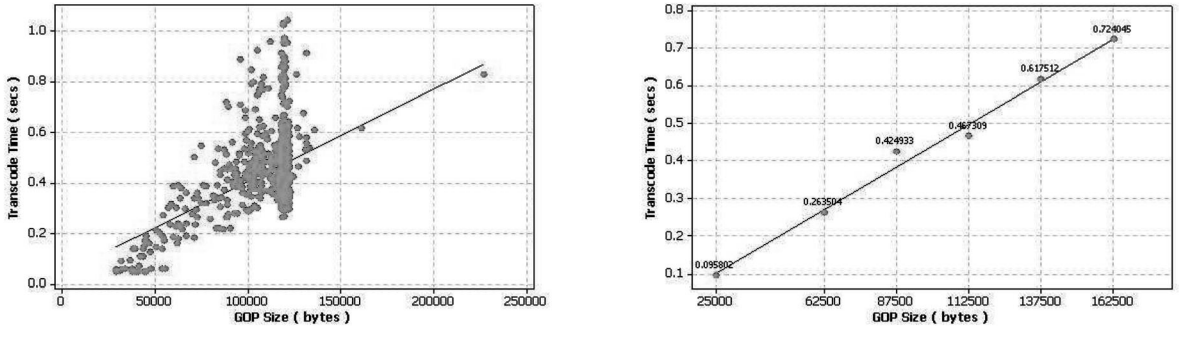
Movie Name	Frame Size	Frame Rate (fps)	Bit Rate (kbps)
Matrix	352×240	29.97	1123
Lord of Ring	352×240	29.97	1123

3.1 Modeling the Relationship between the Transcode Time and the GOP Size

Transcoding of an MPEG GOP does not consume a constant amount of processing, due in part to the fact that a GOP is composed of different frame types and in part to the wide variation between scenes and different motion relations among frames. Each GOP has three kinds of frames: I frames (intraframe), P frames (predictive frame), and B frames (bidirectional predictive frame). I frames are self-contained, complete images. P and B frames are encoded as differences from other reference frames. Due to different motion relations existing among frames, a GOP may contain different number of I, P, and B frames. This makes prediction of the time to transcode the next GOP based on past behavior difficult.

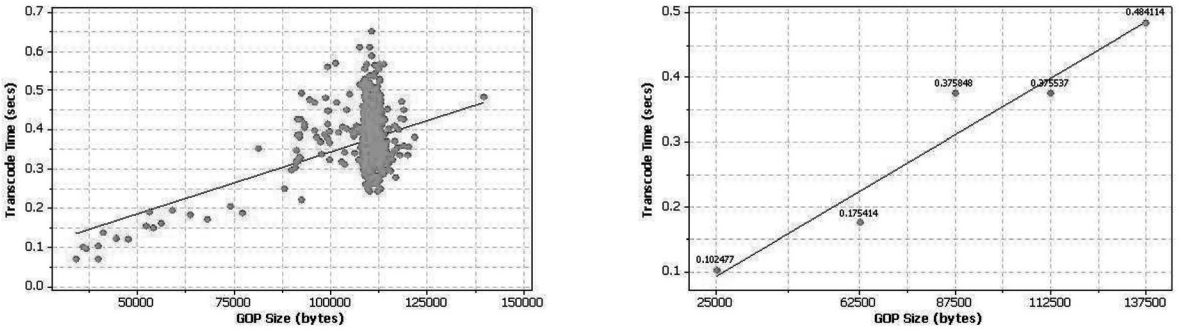
We do a set of experiments to observe the transcoding time of two different movies as illustrated in Table 2. For each movie, three operations are performed: changing the bit rate, reducing the frame rate, and resizing the frame.

Fig. 2a plots the transcode time as a function of GOP size when the bit rate of "Lord of the Rings" is reduced to 50kbps. The straight line in the figure is the linear regression line, obtained by statistically analyzing the data set. Fig. 3a demonstrates the scatterplot of the transcode time as a function of GOP size for the movie "The Matrix." For each movie, we also plotted the transcode time for the other two transcoding operations: changing the frame rate and resizing frames. However, due to the paper length limit, we do not give the scatterplots here. For both movies and the three transcoding operations, the regression equations for the tentative linear regression analysis are given in Table 3. In the table, the GOP size is measured in terms of KBs and the transcode time is measured in terms of milliseconds. From the R^2 values, we notice that a linear model cannot adequately describe the relationship between the transcode time and the GOP size. However, as the scatterplot in Fig. 2a suggests, the transcode time does



(a) (b)

Fig. 2. Transcode time versus GOP size (movie: Lord of the Rings; operation: reducing bit rate to 50 kbps). (a) Tentative linear regression modeling. (b) Linear regression modeling using means.



(a) (b)

Fig. 3. Transcode time versus GOP size (movie: The Matrix; operation: reducing bit rate to 50 kbps). (a) Tentative linear regression modeling. (b) Linear regression modeling using means.

increase as the GOP size increases. The difficulty in fitting it into a linear model is caused by the wide variation of the transcode time for a given GOP size. Thus, for ease of analysis, we divide the GOP size into regions, as shown in Table 4. For each region, the average of the GOP transcoding time is calculated, as shown in Fig. 2b, where each point in the figure has the regional mean value of the GOP size as its x value and the regional mean value of the transcode time as its y value. For the scatterplot drawn in Fig. 2b, a linear regression line with R^2 value as high as 99 percent is obtained. The corresponding linear equation is given in Table 5. In Table 5, the GOP size is in terms of KBs and the transcode time is in terms of milliseconds. Therefore, we have obtained a good linear model to describe the relationship between the transcode time and the GOP size considering a specific movie and a specific transcoding operation.

3.2 Predicting Execution Time on a Single PC

Based on the linear model built in the previous section, we can estimate the transcode time of the GOPs processed so far and incrementally build a predictor to predict the execution time for the next GOP. As presented by Bavier et al. [16], building a linear predictor based on the canonical least squares algorithm would be computationally too expensive for the scheduling purpose. They designed a predictor that approximates the linear model. Through experimental results, they also verified that the predictor works even better than the linear model. In this paper, we adopt the same method as theirs to build a predictor that approximates the linear model presented in Section 3.1. However, because our model differs from theirs in that the regional means are used, our predictor is built differently from their predictor.

Table 6 defines all the parameters used to predict the transcode time. The prediction is carried out in two separate steps. One is to incrementally build the predictor based on the behavior of the GOPs processed so far. The other is to predict the transcode time for a given GOP.

The $Predictor$ is initialized as $(0, Default, 0, 0, 0)$. Once a GOP is processed, the GOP size and its transcode time, $(size, time)$, are recorded and the $Predictor$ is updated

TABLE 3
Tentative Linear Regression Modeling

Operation Type	Lord of Ring	Matrix
Bit Rate	$39.8 + 0.004size$ $(R^2 = 20.5\%)$	$-1467 + 0.017size$ $(R^2 = 18.8\%)$
Frame Rate	$-49 + 0.005size$ $(R^2 = 51.5\%)$	$-89.4 + 0.004size$ $(R^2 = 20.1\%)$
Frame Size	$-38.7 + 0.004size$ $(R^2 = 45.1\%)$	$31.2 + 0.002size$ $(R^2 = 14.8\%)$

TABLE 4
Regions of GOP Size in Terms of Bytes

Region No.	1	2	3	4	5	6
Range of the GOP Size	0-50,000	50,000-75,000	75,000-100,000	100,000-125,000	125,000-150,000	150,000-175,000
Average of the Region	25,000	62,500	87,500	112,500	137,500	162,500

TABLE 5
Linear Regression Modeling Using Regional Means

Operation Type	Lord of Ring	Matrix
Bit Rate	$-11.8 + 0.005size$ $(R^2 = 99\%)$	$6.3 + 0.003size$ $(R^2 = 92.8\%)$
Frame Rate	$-78.3 + 0.006size$ $(R^2 = 94.2\%)$	$-42.4 + 0.004size$ $(R^2 = 95.5\%)$
Frame Size	$-62.4 + 0.004size$ $(R^2 = 95.9\%)$	$20.8 + 0.002size$ $(R^2 = 88.8\%)$

TABLE 6
Parameters Used in Prediction

Parameters	Definition
$GopsReg[i]$	The total number of GOPs processed so far for region i (i=1,2,...,6)
$mRegSize[i]$	Smoothed average of the size of GOPs processed so far for region i (i=1,2,3,...,6), i.e., the regional mean of GOP size
$mRegTime[i]$	Smoothed average of the transcode time of GOPs processed so far for region i (i=1,2,3,...,6), i.e., the regional mean of the transcode time
$msize$	Smoothed average of $mRegSize[i]s$ $(i = 1, 2, ..., 6)$ obtained so far
$mtime$	Smoothed average of $mRegTime[i]s$ $(i = 1, 2, ..., 6)$ obtained so far
$xdiff$	Smoothed average of the difference between $mRegSize[i]$ and $msize$, it is used as the horizontal distance to calculate the slope for the linear equation
$ydiff$	Smoothed average of the difference between $mRegTime[i]$ and $mtime$, it is used as the vertical distance to calculate the slope for the linear equation
$samples$	The total number of GOPs processed so far
$DistinctRegs$	The number of distinct regions encountered so far
$EnoughRegs$	The minimal number of distinct regions that should be encountered before $xdiff$ and $ydiff$ can be calculated
$Predictor$	Predictor is expressed as $(msize, mtime, xdiff, ydiff, samples)$
$Default$	The default GOP transcode time used before a predictor is built up

accordingly. The basic idea is that the linear slope, $(xdiff, ydiff)$, is incrementally approximated according to the difference between the accumulated regional means, $(mRegTime[i], mRegSize[i])$, and the accumulated global means, $(mtime, msize)$, after enough units have been processed. The procedure can be described step by step as follows: First, the region to which the GOP belongs to is found out to be i, and $(mRegTime[i], mRegSize[i])$ is updated. Then, $(xdiff, ydiff)$ is updated only when two conditions are both met. One is that enough samples have been accumulated, i.e., $DistinctRegs \geq EnoughRegs$. The other is that $(mRegTime[i] - mtime)$ shows the same increasing or decreasing tendency as that of $(mRegSize[i] - msize)$. Finally, $msize$, $mtime$, $samples$, and $DistinctRegs$ are updated to count the newly processed unit into the accumulated value. Fig. 13 illustrates this procedure.

The transcode time of a GOP of size $size$ is predicted according to the $Predictor$ as

$$prediction = \begin{cases} Default & samples = 0 \\ mtime & samples > 0 \\ mtime + (size - msize) & \\ *ydiff/xdiff & samples > 0, xdiff > 0. \end{cases}$$

Fig. 14 illustrates the prediction algorithm.

3.3 Predicting Execution Time on a Set of Heterogeneous PCs

To process a stream in parallel on a set of heterogeneous machines, prediction becomes complicated. There emerges two new questions. First, how do we build a predictor when each server only processes part of the units that are distributed to it? Second, given a predictor of the stream, how do we predict the execution time of a given GOP on each specific server when the servers possess different power?

We propose building the predictor as follows: Each server incrementally builds its own predictor, which we call *partial predictor*, based on the information of the GOPs it has processed so far. The scheduler periodically collects the *partial predictors* from all computing servers and combines them to be a *global predictor*. The *partial predictors* are constructed independently on each computing server according to the algorithm illustrated in Fig. 13. Table 7 defines the symbols used throughout the rest of the paper. Let there be N computing servers. When the scheduler collects the *partial predictors*, it generates the *global predictor*, as demonstrated in Fig. 4. First, $(mtime_1, mtime_2, \ldots, mtime_N)$ and $(ydiff_1, ydiff_2, \ldots, ydiff_N)$ are normalized according to the weight vector $(w_1, w_2, w_3, \ldots, w_N)$. Then, according to the sample size returned by each computing server, $(msize_g, mtime_g, xdiff_g, ydiff_g)$ is calculated as the arithmetic average of their corresponding values presented in $(Predictor_1, Predictor_2, \ldots, Predictor_N)$.

When the scheduler schedules media units among computing servers, the time to process a given GOP on server i is predicted based on the *global predictor* according

TABLE 7
Definition of the Predictors

Symbol	Definition
$Predictor_i$	The *partial predictor* built on server i, expressed as $(msize_i, mtime_i, xdiff_i, ydiff_i, samples_i)$.
$Predictor_g$	The *global predictor*, expressed as $(msize_g, mtime_g, xdiff_g, ydiff_g, samples_g)$.
w_i	The weight of server i, where $w_i \geq 1$. Higher weight means more processing power.
$prediction_i$	The predicted execution time of a given GOP on server. i

Input: $Predictor_i(msize_i, mtime_i, xdiff_i, ydiff_i, samples_i)\ \ i = 1, 2, ..., N$
$w_i \ \ i = 1, 2, ..., N$
Output: $Predictor_g(msize_g, mtime_g, xdiff_g, ydiff_g, samples_g)$
Algorithm :
$r_i = \frac{samples_i}{\sum_{i=1}^{N} samples_i}$
$msize_g = \sum_{i=1}^{N} (msize_i \times r_i), \quad xdiff_g = \sum_{i=1}^{N} (xdiff_i \times r_i)$
$mtime_g = \sum_{i=1}^{N} \frac{mtime_i \times r_i}{w_i}, \quad ydiff_g = \sum_{i=1}^{N} \frac{ydiff_i \times r_i}{w_i}$

Fig. 4. Algorithm to generate the *global predictor* based on *partial predictors*.

to the algorithm in Fig. 14. Due to the heterogeneity of the servers, we should also take into consideration the processing power of each server. Therefore, the prediction is performed as follows:

$$prediction_i = getprediction(size, Predictor_g)/w_i \quad i = 1, 2, \ldots, N. \tag{6}$$

If multiple streams are processed in the computing cluster, it is desirable to build the *partial predictors* and *global predictor* for each stream and perform the prediction stream-wise. The reason is that, for each stream, the transcode time and unit size conforms to a specific linear relation. In the rest of the paper, a stream refers to a specific movie which requires a specific transcoding operation.

4 Prediction-Based Load Balancing Techniques

With the prediction algorithm in place, we design prediction-based load balancing algorithms in this section. The heterogeneity of computing servers is described by the weight vector $(w_1, w_2, \ldots, w_N)$ defined in Table 7.

4.1 Least Load First (LLF) and Prediction-Based Least Load First (P-LLF)

In the cluster-based Web server system Gage [13], a Least Load First (LLF) scheduling algorithm is employed to distribute client requests among servers. According to their policy, the LLF algorithm runs as follows: For each media stream, the execution time consumed by a media unit is predicted as the weighted average time of the processed units. This prediction is updated periodically by collecting information from computing servers. The prediction is used to predict the outstanding load on each computing server and to schedule media units such that a least loaded server is chosen to process a unit.

We extend LLF with our prediction policy, proposed in Section 3.3, and propose Prediction-based Least Load First (P-LLF) algorithm. P-LLF, as described in Fig. 5, contains two parts. Let there be N computing servers and M streams. The scheduler maintains a load indicator L_i for each server $i (i = 1, 2, \ldots, N)$ and a predictor $Predictor_g^j$ for each stream $j (j = 1, 2, \ldots, M)$. The first part is the periodic adjustment of the load indicators and stream predictors. For each computing server, the load status is observed and the load indicator is updated. For each stream, the partial predictors are collected from computing servers and combined to generate the global predictor. The second part is scheduling units according to the load status and predictions. The predicted process time of the media unit is calculated according the global predictor. A server is chosen such that its load is the least after the unit is scheduled. Once a unit is scheduled to the chosen server, its predicted time is stamped and it is dispatched to the server. Each computing server records its current outstanding load, i.e., the total process time predicted for the units that are dispatched to it and waiting to be processed.

Both LLF and P-LLF aim to distribute the workload among servers proportionally to their capacity and produce

Periodical Adjustment of the Load Indicators and Stream Predictors:

(1) Collect the *partial predictors* from the computing servers
(2) Collect the outstanding loads, denoted as $ol_i \ (i = 1, 2, ..., N)$, from the computing servers.
(3) For each server i, get the the remaining load, rl_i, from the corresponding dispatch queue
(4) For each stream j, generate the *global predictor* $Predictor_g^j$ by combining its N *partial predictors*
(5) For each server i, set the load indicator as $L_i = ol_i + rl_i$

Prediction-based Least Load First Scheduling :
Input: $Predictor_g^j \ \ j = 1, 2, ..., M$
$w_i \ \ i = 1, 2, ..., N$
$L_i \ \ i = 1, 2, ..., N$
Output: $ChosenPC$
$L_{ChosenPC}$
Algorithm:
while (the unit buffer contains GOPs)
fetch a GOP from the unit buffer, let $size$ be the GOP size and k be its stream no.
$prediction_i = getprediction(size, Predictor_g^k)/w_i, \quad i = 1, 2, ..., N$
$tentativeL_i = L_i + prediction_i, \quad i = 1, 2, ..., N$
$ChosenPC = x$, such that $tentativeL_x = min_{i=1 \to N}(tentativeL_i)$
put the GOP into the xth dispatch queue
$L_x = tentativeL_x$

Fig. 5. Prediction-based Least Load First scheduling algorithm.

Input: w_i $(i = 1, 2, 3, ..., N)$
$comp_j(t)$ $(j = 1, 2, 3, ..., M)$
Output: $subset_j$ $(j = 1, 2, 3, ..., M)$
Algorithm:

$r_j(t) = comp_j(t) / \sum_{j=1}^{M} comp_j(t) \quad (j = 1, 2, 3, ..., M)$
if (any $r_j(t)$ $(j = 1, 2, ..., M)$ varies by more than β)
  $subset_j \leftarrow \{\} \quad (j = 1, 2, 3, ..., M)$
  $T_i = w_i \quad (i = 1, 2, 3, ..., N)$
  $Total = \sum_{i=1}^{N} T_i$
  $Stoken_j(t) \leftarrow r_j(t) \times Total \quad (j = 1, 2, 3, ..., M)$
  $i \leftarrow 1$
  for $j \leftarrow 1$ **to** M
    $RStoken \leftarrow Stoken_j(t)$
    while $RStoken > 0$ **do**
      if $RStoken \geq T_i$ **then**
        $RStoken \leftarrow RStoken - T_i$
        $subset_j \leftarrow subset_j \cup \{i\}$
        $i \leftarrow i + 1$
      else
        $subset_j \leftarrow subset_j \cup \{i\}$
        $T_i \leftarrow T_i - RStoken$
        $RStoken \leftarrow 0$

Fig. 6. Partitioning algorithm.

high throughput, although the degree of load balancing may be different due to the accuracy of their prediction policies. In both schemes, the media units of the same stream are possibly distributed to different servers and, thus, may cause high jitters at the destination.

4.2 Adaptive Partition (AP) and Prediction-Based Adaptive Partition (P-AP)

There are two goals when designing a load balancing scheme in the media cluster: 1) to balance the workload among servers to achieve the maximum throughput, and 2) to maintain the flow order for each outgoing stream while processing its media units on multiple servers.

We have found that the first fit scheme [14] gives maximum throughput, but produces maximum jitter (or out-of-order departure) because the media units are distributed to all the servers. The adaptive load sharing (ALS) policy essentially reduces the jitter by doing an allocation of streams at every epoch. During an epoch, a stream is allocated to only one processor, but can be allocated to any processor at different epochs. However, this approach may cause occasional waste of resource and reduce the system throughput. To strike a balance between the throughput and the delay jitter, it may be better to send media units of the same stream to a limited set of servers in an epoch. Hence, we propose an Adaptive Partitioning (AP) algorithm that dynamically partitions the servers into several subsets and establishes mapping between the streams and the subsets. The partitioning and mapping is established according to both the observed computation requirements of different streams and the processing power of different servers. In other words, both the stream heterogeneity and server heterogeneity are taken into account. The LLF algorithm is adopted to schedule a stream among the mapped subset of servers.

Different streams require different computational power for their specific transcoding operations. The media server records the computation complexity of each stream at time t

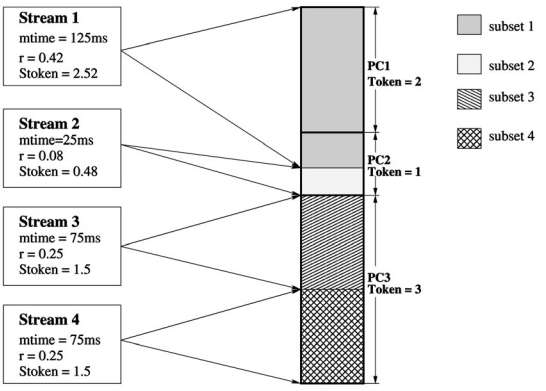


Fig. 7. A partitioning example.

by the vector $(comp_1(t), comp_2(t), \ldots, comp_M(t))$, where $comp_j(t)$ is the weighted average time accumulated for the stream. We normalize the computation complexity of each stream as $r_j(t)$ and let

$$r_j(t) = comp_j(t) / \sum_{j=1}^{M} comp_j(t). \quad (7)$$

The weights of servers can be viewed as the capacity tokens that represent the workload completed in a unit time on a server. Hence, we define a token vector $(T_1, T_2, T_3, \ldots, T_N)$, where $T_i = w_i$, and the total token of the computing cluster as $Total = \sum_{i=1}^{N} T_i$. The total token is then partitioned into M subsets according to the computation requirements of streams, with each subset mapped to one stream. The total token held by subset i, i.e., by stream i, is defined as $Stoken_i(t)$

$$Stoken_i(t) = r_i(t) \times Total \quad i = 1, 2, \ldots, M. \quad (8)$$

Obviously, $(Stoken_1(t), Stoken_2(t), \ldots, Stoken_M(t))$ represent M fractions of the total token held by all the servers. Each fraction corresponds to a subset which contains at least one server. In this way, the tokens of N servers are distributed among M streams. One server may be assigned to several subsets, which means it is shared among several streams. Also, one subset may contain several servers, i.e., one stream can be processed on several servers simultaneously.

If a stream is mapped to multiple servers, say $k(k \geq 1)$ servers. The media server schedules the stream among the k servers according to the LLF algorithm described in Section 4.1. If a stream is mapped to only one server, all the media units of the stream are scheduled to be processed on this server.

Fig. 6 describes the partitioning algorithm, which partitions the N servers into M subsets according the servers' processing power and the streams' computation requirements. The partitioning is expressed as $(subset_1, subset_2, \ldots, subset_M)$. $subset_j$ is a set of servers. Repartitioning is performed only when any $r_j(t)$ varies by more than a tolerable percentage, say β. In summary, the AP algorithm works as follows: The new computation requirements of streams determines if repartitioning is needed. If it is needed, mapping between streams and subsets of servers is reestablished.

Fig. 7 shows a partitioning example of three servers and four streams. The servers have different capacities. The T_is

TABLE 8
Mapping Streams to Subsets

Stream No.	Servers mapped to the stream
1	$\{1, 2\}$
2	$\{2\}$
3	$\{3\}$
4	$\{3)\}$

held by servers and the $Stoken_i(t)$s held by streams are shown in the figure, the total token held by the servers is 6. The mapping established between streams and subsets is demonstrated in Table 8. According to Fig. 7 and Table 8, stream 1 is mapped to two servers, server1 and server2. Its media units are scheduled among these two servers according to P-LLF algorithm. Streams 3 and 4 share server 3, and streams 1 and 2 share server 2.

We also extend AP to be P-AP by employing the prediction policy proposed in Section 3. With P-AP, the computation complexity of each stream, $comp_j(t)$, equals $mtime_g^j(t)$ proposed in Section 3. When a stream is mapped to several computing servers, P-LLF is used to choose a server among several.

5 Experimental Evaluation

5.1 Experimental Settings

Table 9 describes the hardware and software configurations of the Media Server and Computing Servers, implemented in our laboratory. For the streaming service, four movies are currently used, namely, "Lord of the Rings," "The Matrix," "Peterpan," and "Resident Evil." To satisfy the clients' requirements, three kinds of transcoding operations are performed, namely, changing the bit rate, resizing the frames, and changing the frame rate. To process a stream in the media cluster, one of the three transcoding operations is performed on one of the movies. The transcoding service provided by each server is derived from a powerful multimedia processing tool called FFMPEG [17].

Fig. 8 demonstrates the software framework of our media cluster. We implement a multithreaded architecture in order to overlap computation and communication.

On the media server, four kinds of threads, namely, *retriever*, *scheduler*, *dispatcher*, and *manager*, are running concurrently. *Retriever* continuously retrieves media units from the disk and stores them in the unit buffer, which adopts FIFO policy. A dispatch queue is maintained for each server which holds all the media units that have been scheduled to the server. The *scheduler* fetches units from the unit buffer and puts them into a dispatch queue according to the load balancing policy discussed in Section 4. Upon the request of a server, the *dispatcher* gets a media unit from the corresponding dispatch queue and transmits it to the server. The *manager* periodically collects information from the servers and feeds the information to the *scheduler*.

On the server node, four threads, *receiver*, *transcoder*, *sender*, and *monitor*, are running concurrently. The *receiver* receives packets from the Manager and assembles them into complete media units. Once a complete media unit is ready, the *transcoder* transcodes the media unit. After transcoding, the *sender* delivers the media unit to the client. Once the *receiver* gives the media unit to the transcoder for processing, it requests another media unit from the media server by sending a "Ready" message. The *monitor* collects information on the server and reports it to the Manager periodically.

5.2 Performance Metrics

Sensitivity to the above design parameters and efficiency of our media cluster are measured with respect to the following performance metrics.

TABLE 9
Hardware and Software Configuration of Media Server Cluster

PC Name	Media Server	Server 1 - Server 3	Server 4 - Server 6	Server 7 - Server 8
CPU	Pentium 4 3.0GHz	Pentium 4 3.0GHz	Pentium 4 2.53GHz	AMD Atholon 1.40GHz
Memory	1GB DDR 2100	1GB DDR 2100	1GB PC133	1GB PC133
OS	Linux Fedora Core 1	Linux Fedora Core 1	Linux Mandrake 9	Linux Red Hat 9

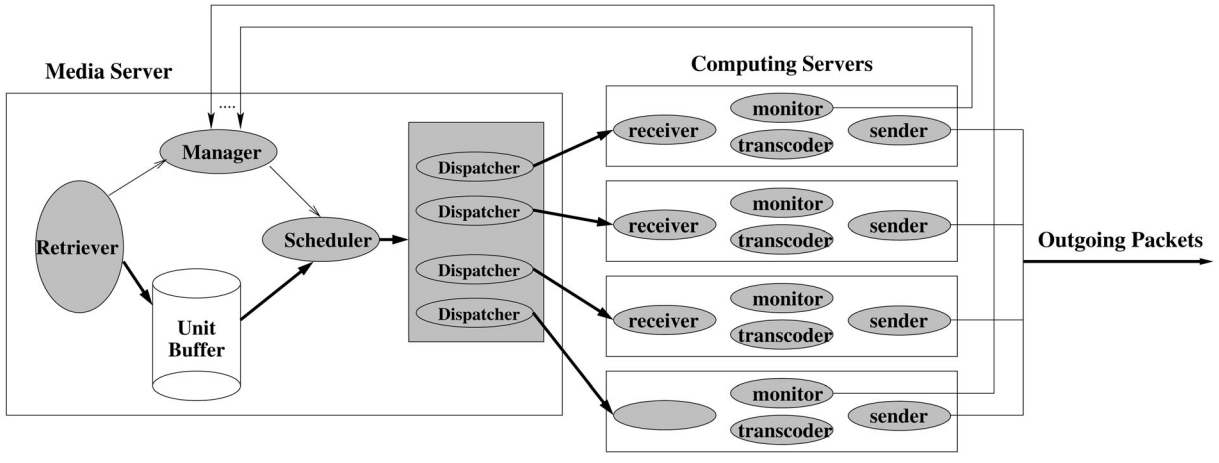


Fig. 8. Software framework of the media cluster.

TABLE 10
Experimental Setting

Cluster Size	1	2	3	4	5	6	7	8
Number of Streams	5	7	11	15	18	20	24	27

5.2.1 System Scalability

A parallel system is desired to be scalable. As the cluster size increases, more and more media units should be processed by the system in a unit of time. Hence, we measure the scalability of our cluster in terms of *system throughput*, which is defined as GOPs/sec when comparing different load balancing schemes.

5.2.2 Load Sharing Overhead

When using prediction-based schemes or the feedback-based scheme ALS, the system throughput may degrade due to the overhead caused by collecting information from computing servers and adapting to load imbalance. We define the *Load Sharing Overhead* as the average time consumed by the media server to poll through all servers to collect information.

5.2.3 Video Quality

As a special requirement imposed by multimedia streaming on our parallel servers, the video quality observed at the receiver side is a very important metric. To observe how the transcoded units are delivered from computing servers to the media server and then go to the client, we write a program called *Departure-Recorder* and run it on the media server. *Departure-Recorder* receives the transcoded units from the computing servers and records the time to receive each unit without extra reordering. Based on this information, we can evaluate the traffic pattern of the departing streams so as to predict the video quality at the client side. To describe the traffic pattern of outgoing streams, we define three metrics as follows:

Metric a: Departure Jitter per Stream is the standard deviation of the interdeparture time among GOPs when the stream departs the media server. It depicts and predicts how smooth a stream may be played out at the client side in real time.

Metric b: Average Interdeparture Time among GOPs per Stream is the mean of the interdeparture times among GOPs when the stream departs from the media server.

Metric c: Out-of-Order Rate per Stream describes how many GOPs among all the GOPs in a stream depart out of order.

5.3 Evaluation of Results

5.3.1 System Throughput

Scalability of the system throughput is one of the most important metrics that we need to examine when comparing different load balancing schemes. Since the throughput is highly affected by the input workload, we generate a large enough number of streams such that the Media Unit Buffer never becomes empty. Thus, we can measure the performance of different load balancing schemes in a fully loaded system. Table 10 illustrates the detailed experimental settings. The load test epoch for feedback-based schemes is 2 seconds.

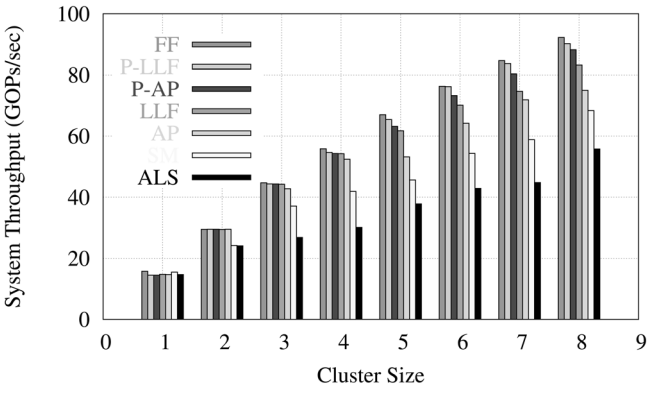


Fig. 9. Scalability of the system throughput.

Fig. 9 demonstrates the scalability of system throughput with increasing cluster size. FF achieves the best scalability because it does not collect feedback from servers. Surprisingly, although P-LLF has high load sharing overhead compared to FF, its throughput is only slightly affected and closely approaches FF. The reason is that the throughput of FF is implicitly affected by the inaccuracy of observing load status through dispatch queues and blindness to stream heterogeneity when scheduling units. Instead, P-LLF explicitly tests the load status and considers the stream heterogeneity for scheduling. Therefore, with P-LLF, the load sharing overhead is counteracted by the better load balancing achieved among servers. Compared to P-LLF, P-AP performs the extra operation of repartitioning servers and remapping streams to servers. But, at the same time, P-AP performs less computation than P-LLF when scheduling units because the least loaded server is chosen within a small subset instead within the whole cluster. Hence, the throughput of P-AP still approaches that of P-LLF. LLF and AP both perform worse than their counterparts P-LLF and P-AP due to the inefficiency of their simplistic prediction method. On the other hand, SM and ALS have much lower throughput than other schemes for to two reasons. First, it is due to the potential load imbalance incurred by maintaining the flow consistency. Second, they schedule streams without considering the heterogeneity among streams. SM avoids dispersing media units of the same stream among different servers even if a server is free. This causes waste of resources, occasional imbalance in load distribution, and reduces the throughput. ALS involves high load sharing overhead and does not take into account the stream heterogeneity. Besides, the HRW function works better for a very large number of homogeneous streams. Therefore, ALS presents less scalability than others in the experiments.

5.3.2 Load Sharing Overhead

Table 11 illustrates the load sharing overhead in terms of milliseconds for the five schemes, P-LLF, P-AP, LLF, AP, and ALS. Because P-LLF and P-AP share the same prediction scheme and both collect stream predictors at the end of each epoch, they have the same load sharing overhead. Similarly, LLF and AP share the same prediction scheme where the accumulated average transcoding times

TABLE 11
Load Sharing Overheads

Cluster Size	1	2	3	4	5	6	7	8
ALS	0.87	1.6	2.3	3.0	5.1	7.6	9.5	12.2
AP/LLF	0.876	1.598	2.321	3.4	5.701	7.803	10.045	12.87
P-AP/P-LLF	0.881	1.783	2.446	4.062	6.21	8.341	11.069	13.607

per stream are collected in each epoch. The load test overhead of the ALS scheme is different from the prediction-based schemes because only the CPU utilization information needs to be collected from servers. As shown in the table, the load test overhead increases almost linearly with the cluster size for all the five schemes because the communication expense increases linearly with the number of servers in the cluster. In addition, ALS incurs less overhead than the prediction-based schemes because its overhead is independent of the number of streams. As the cluster size increases, the prediction-based schemes incur higher overhead than ALS due to the increased number of streams. Even then we observe through the experimental results that prediction does lead to higher system throughput. AP/LLF has less overhead than P-AP/P-LLF because the message size transmitted per stream is smaller than that of P-AP/P-LLF.

5.3.3 Video Quality

The traffic pattern of outgoing streams observed on the media server reflects the user-perceived video quality at the receiver side. We measure the video quality in terms of *interdeparture time among GOPs*, *departure jitter*, and *Out-oF-Order (OFO) departure rate*. The experimental settings are the same as that for testing the system throughput.

As shown in Fig. 10, FF, LLF, and P-LLF incur the largest OFO departure rate since they schedule the media units freely without maintaining flow order. LLF and P-LLF perform similarly, with P-LLF doing marginally better because of a better prediction algorithm. FF does a little worse than LLF/P-LLF because of the delay in observing overload on servers through the dispatch queues. With the flow concept in mind, SM and ALS incur small OFO departure. It is interesting to note that AP/P-AP greatly reduces OFO departure compared to P-LLF and FF and incurs only marginally higher OFO departure than SM and ALS. Given that P-AP also produces very good throughput, as shown in Fig. 9, it is a very promising scheduling scheme to achieve both high throughput and low OFO departure rate.

Fig. 11 illustrates the departure jitter per stream for all load sharing schemes. It shows the same tendency in variation of OFO departure rate. The best departure jitter is achieved by SM because it processes all units of the same stream on the same server, thus guaranteeing in-order departure and produces the smallest jitter. ALS maintains the flow order at higher computation and communication overheads, thus incurring slightly larger jitter than SM. AP and P-AP map each stream to a limited set of servers, hence they reduce out-of-order processing of consecutive units belonging to the same stream and, so, reducing jitter as well as ALS.

Fig. 12 demonstrates the average interdeparture time among GOPs per stream. Since we have multiple streams in the system, as shown in Table 10, the interdeparture time per stream is not simply the inverse of system throughput. But, it still shows a similar scalability as that of the system

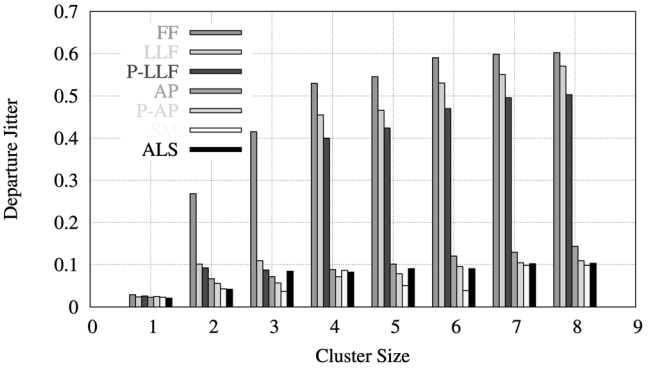


Fig. 11. Departure jitter.

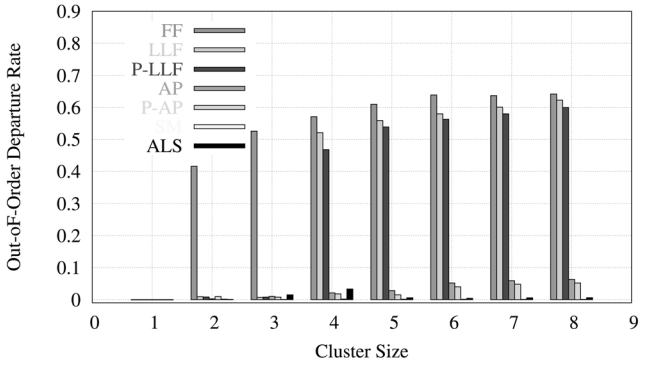


Fig. 10. Out-of-order departure rate.

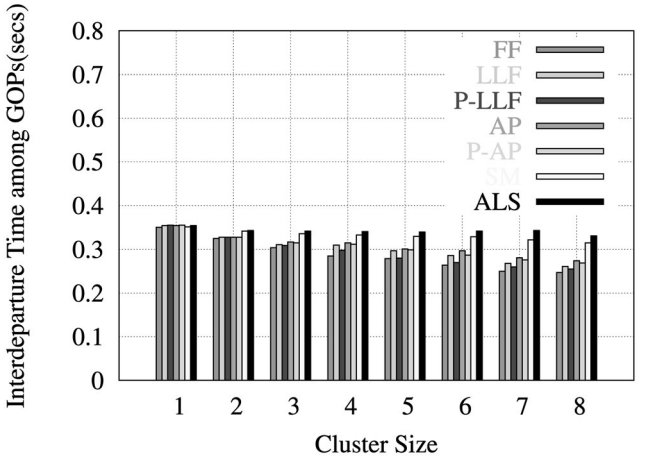


Fig. 12. Interdeparture time.

Input: Size and transcode time of the lately processed GOP - $(size, time)$
Output: $Predictor(msize, mtime, xdiff, ydiff, samples)$
Algorithm:

$i \leftarrow$ *the region no. where the GOP size belongs to*
$GopsReg[i] \leftarrow GopsReg[i] + 1$
$samples \leftarrow samples + 1$
if $(GopsReg[i] == 1)$ $\quad mRegTime[i] = time$
else $\quad mRegTime[i] = mRegTime[i] * (1 - a) + time * a$
if $(DistinctRegs \geq EnoughRegs)$
  $SizeDiff = mRegSize[i] - msize$
  $TimeDiff = mRegTime[i] - mtime$
  if $(SizeDiff$ *and* $TimeDiff$ *have the same sign)*
    if $(DistinctRegs == EnoughRegs)$
      $xdiff = |SizeDiff|$
      $ydiff = |TimeDiff|$
    else
      $xdiff = xdiff * (1 - a) + |SizeDiff| * a$
      $ydiff = ydiff * (1 - a) + |TimeDiff| * a$
if $(samples == 1)$
  $msize = mRegSize[i]$
  $mtime = mRegTime[i]$
else
  $msize = msize * (1 - a) + mRegSize[i] * a$
  $mtime = mtime * (1 - a) + mRegTime[i] * a$
if $(GopsReg[i] == 1)$
  $DistinctRegs \leftarrow DistinctRegs + 1$

Fig. 13. Algorithm to dynamically build the GOP predictor based on processed GOPs.

throughput. FF achieves the best performance because it has no overhead. As we can see, P-LLF closely approaches FF. P-AP achieves a similar effect as that of P-LLF, both better than LLF and AP because of better load balancing led by better prediction. SM and ALS fail to give good performance because of their blindness to the stream heterogeneity and the overhead to maintain flow order.

6 Related Work

The transmission of multimedia information through networks has long been a research topic, and it is claimed that multimedia application is becoming one of the killer applications in this century. Due to the receiver heterogeneity and dynamically varying network conditions, a multimedia stream should be transformed to meet different clients' requirements. A traditional way to solve the above problem is to store multiple copies of the source stream on the media server and select a copy according to some initial negotiation with the client. Although the disks have gotten larger and cheaper, online transcoding service has become a widely adopted solution to provide various clients with the media source according to their requirements. Transcoding is a process that transforms a compressed video bit stream into different bit streams either by employing the same compression format with alternate parameters or by employing another format altogether. Many researchers [1], [2], [18], [19] have addressed how to customize the multimedia contents to match user preferences or the diversity of network conditions and display devices. Chandra et al. [1] used JPEG transcoding techniques to customize the size of objects constituting a Web page, thus allowing a Web server to dynamically allocate available bandwidth among different classes. Fox et al. proposed to dynamically distill the Web page content on active proxies when they are transmitted through the network [6], [5]. They also implement a cluster-based Web

Input: GOP Size $-$ $size$
  $Predictor(msize, mtime, xdiff, ydiff, samples)$
Output: $prediction$
Algorithm getprediction(time,Predictor)
  if $(samples > 0)$
    $prediction = mtime$
    if $(xdiff > 0)$
      $slope \leftarrow ydiff/xdiff$
      $prediction \leftarrow prediction + (size - msize) \times slope$
  else
    $prediction \leftarrow Default$

Fig. 14. Algorithm to predict the transcode time for a given GOP.

distillation proxy called TranSend [20]. However, the most computationally expensive task performed by TranSend is the distillation of images. Their scheduling schemes emphasized fault tolerance more than load balancing, and parallel processing of a single stream was not considered. Welling et al. proposed the concept of CLuster-based Active Router Architecture (CLARA) [3], where a computing cluster is attached to a dedicated router. Using CLARA, the multimedia transcoding tasks are paralleling processed in the computing cluster instead of on the router itself [3], [4]. This solution brings up the new problem of how to efficiently utilize the resources provided by the computing cluster to meet media streaming requirement.

In the network domain, the goal of load balancing is complicated by the additional requirement of preserving the flow order. The random distribution cannot preserve packet order within a flow if per-flow information is not maintained. Modulus-based round-robin policy also has the drawback that all flows are remapped if the number of computing nodes is changed. There has been almost no research in developing load balancing algorithms with the concept of flows in mind. One related paper was published by Kencl and Boudec [15] who extended the HRW algorithm [21] with a feedback mechanism to do adaptive load balancing in network processors. This allows adjustment to the load distribution with minimum flow remapping [15] and copes with request identifier space locality. We find the paper interesting, but they only reported some theoretical and simulation results without any real implementation.

Taking MPEG transcoding as the first application in their cluster-based active router architecture, Welling et al. adopted round-robin algorithm to dispatch media units for transcoding among the nodes [3]. However, no experimental results were provided for this. By designing and implementing an active router cluster supporting transcoding service, we were able to evaluate three load sharing schemes, namely, round robin, stream-based round-robin, and adaptive load sharing [14]. It was shown that round-robin is simple and fast, but provides no guarantee to the playback quality of output streams because it causes out-of-order departure of processed media units. Adaptive load sharing scheme, proposed by Kencl and Boudec [15], achieves better unit order in output streams, but involves higher overhead to map the media unit to an appropriate node. As a result, the throughput is reduced. Stream-based round robin achieves good performance in terms of both throughput and output order, but its advantage is confined to a homogeneous and highly loaded system.

For MPEG-4 encoding, He et al. proposed several scheduling algorithms that allocate MPEG-4 objects among multiple workstations in a cluster to achieve real-time interactive encoding rate [22]. When an object is partitioned and processed on multiple workstations, the data dependence is resolved by storing related reference data in each processor's local memory for motion estimation. The scheduling algorithms are derived on the basis of a video model called MPEG-4 Object Composition Petri Net (MOCPN), which captures the spatio-temporal relationship between various objects and user interaction. However, in their schemes, the appearance and disappearance of objects in a video session is simply modeled by user interactions, which is not true for an automatic MPEG-4 playout session. In addition, their scheduling algorithms lack generality for the conventional frame-based coding schemes like MPEG-1/2 and H.263.

7 Conclusion

The aim of the paper was to develop scheduling and load balancing algorithms to ensure high throughput and low jitter for multimedia processing on a cluster-based Web server where a few computing nodes are separately reserved for high-performance multimedia applications. We consider the multimedia streaming service which requires on-demand transcoding operations as an example.

In this paper, we designed and implemented a media cluster and evaluated the efficiency of seven load scheduling schemes for a real MPEG stream transcoding service. Due to the variability of the individual transcoding jobs, it was necessary to predict the execution time for each job and schedule accordingly. We proposed a dynamic prediction algorithm that predicts the transcoding time for each media unit. Based on the algorithm, we proposed two new load sharing policies, Prediction-based Least Load First (P-LLF) and Prediction-based Adaptive Partitioning (P-AP). For comparison, we implemented Least Load First (LLF) and Adaptive Partitioning (AP) policies where the prediction is based on average execution time. Besides, we also implemented three nonprediction-based schemes, namely, First Fit (FF), Stream-based Mapping (SM), and Adaptive Load Sharing (ALS). Among the seven load sharing schemes, FF, P-LLF, and LLF achieve high throughput but also incur high jitter, whereas P-AP, AP, SM, and ALS try to maintain the unit order of outgoing streams to reduce jitter. Experimental results show that a good balance between throughput and departure jitter is achieved by P-AP. P-AP outperforms FF, LLF, and P-LLF because it establishes mapping among streams and subsets of servers. P-AP outperforms SM and ALS because it takes into consideration the stream heterogeneity. P-AP and P-LLF outperform their counterparts, AP and LLF, because of the better prediction method.

Acknowledgments

The authors thank Professor Inbum Jung for providing them with the various MPEG streams and helping them understand the terms in the multimedia area.

References

[1] S. Chandra, C.S. Ellis, and A. Vahdat, "Differentiated Multimedia Web Services Using Quality Aware Transcoding," *Proc. INFOCOM 2000-19th Ann. Joint Conf. IEEE Computer and Comm. Soc.,* Mar. 2000.

[2] R. Keller, S. Choi, M. Dasen, D. Decasper, G. Fankhauser, and B. Platter, "An Active Router Architecture for Multicast Video Distribution," *Proc. IEEE INFOCOM,* 2000.

[3] G. Welling, M. Ott, and S. Mathur, "A Cluster-Based Active Router Architecture," *IEEE Micro,* vol. 21, no. 1, Jan./Feb. 2001.

[4] M. Ott, G. Welling, S. Mathur, D. Reininger, and R. Izmailov, "The Journey Active Network Model," *IEEE J. Selected Areas in Comm.,* vol. 19, no. 3, pp. 527-537, Mar. 2001.

[5] A. Fox, S. Gribble, E. Brewer, and E. Amir, "Adapting to Network and Client Variability via On-Demand Dynamic Distillation," *Proc. Seventh Int'l Conf. Architecture Support for Programming Language and Operating Systems (ASPLOS-VII),* 1996.

[6] A. Fox, S.D. Gribble, and Y. Chawathe, "Adapting to Network and Client Variation Using Active Proxies: Lessons and Perspectives," *IEEE Personal Comm. on Adaptation,* special issue, 1998.

[7] E. Amir, S. McCanne, and R. Katz, "An Active Service Framework and Its Application to Real-Time Multimedia Transcoding," *Proc. ACM SIGCOMM Symp.,* Sept. 1998.

[8] B.A. Shirazi, A.R. Hurson, and K.M. Kavi, *Scheduling and Load Balancing in Parallel and Distributed Systems.* CS Press, 1995.

[9] M. Satyanarayanan, "Scalable, Secure, and Highly Available Distributed File Access," *Computer,* May 1990.

[10] E. Katz, M. Butler, and R. McGrath, "A Scalable Http Server: The Ncsa Prototype," *Computer Networks and ISDN Systems,* vol. 27, pp. 155-164, 1994.

[11] H. Zhu, T. Yang, Q. Zheng, D. Watson, O. Ibarra, and T. Smith, "Adaptive Load Sharing for Clustered Digital Library Servers," *Proc. Seventh Int'l Symp. High Performance Distributed Computing,* pp. 235-242, 1998.

[12] H. Zhu, H. Tang, and T. Yang, "Demand-Driven Service Differentiation in Cluster-Based Network Servers," *Proc. IEEE INFOCOM,* 2001.

[13] C. Li, G. Peng, K. Gopalan, and T. Chiueh, "Performance Garantees for Cluster-Based Internet Services," *Proc. 23rd Int'l Conf. Distributed Computing Systems (ICDCS '03),* May 2003.

[14] J. Guo, F. Chen, L. Bhuyan, and R. Kumar, "A Cluster-Based Active Router Architecture Supporting Video/Audio Stream Transcoding Services," *Proc. 17th Int'l Parallel and Distributed Processing Symp. (IPDPS '03),* Apr. 2003.

[15] L. Kencl and J.Y.L. Boudec, "Adaptive Load Sharing for Network Processors," *Proc. IEEE INFOCOM,* 2002.

[16] A.D. Bavier, A.B. Montz, and L.L. Peterson, "Predicting Mpeg Execution Times," *Proc. Int'l Conf. Measurements and Modeling of Computer Systems (SIGMETRICS),* pp. 131-140, 1998.

[17] Ffmpeg Multimedia System, http://ffmpeg.sourceforge.net/, 2004.

[18] C.K. Hess, D. Raila, R.H. Campbell, and D. Mickunas, "Design and Performance of MPEG Video Streaming to Palmtop Computers," *Multimedia Computing Networks,* 2000.

[19] E. Amir, S. McCanne, and H. Zhang, "An Application Level Video Gateway," *Proc. ACM Multimedia,* 1995.

[20] A. Fox, S.D. Gribble, Y. Chawathe, E.A. Brewer, and P. Gauthier, "Cluster-Based Scalable Network Services," *Proc. Symp. Operating Systems Principles,* 1997.

[21] K.W. Ross, "Hash Routing for Collections of Shared Web Caches," *IEEE Network,* vol. 11, no. 6, Nov.-Dec. 1997.

[22] Y. He, I. Ahmad, and M.L. Liou, "Real-Time Interactive MPEG-4 System Encoder Using a Cluster of Workstations," *IEEE Trans. Multimedia,* vol. 1, no. 2, pp. 217-233, 1999.

Jiani Guo received the BE and ME degrees in computer science and engineering from the Huazhong University of Science and Technology, and the PhD degree in computer science from the University of California, Riverside. She currently works for Cisco Systems. Her research interests include job scheduling in parallel and distributed systems, cluster computing, active router, and network processor. She is a student member of the IEEE.

Laxmi Narayan Bhuyan has been a professor of computer science and engineering at the University of California, Riverside since January 2001. Prior to that, he was a professor of computer science at Texas A&M University (1989-2000) and program director of the Computer System Architecture Program at the US National Science Foundation (1998-2000). He has also worked as a consultant to Intel and HP Labs. Dr. Bhuyan's current research interests are in the areas of computer architecture, network processors, Internet routers, and parallel and distributed processing. He has published more than 150 papers in related areas in reputable journals and conference proceedings. He has served on the editorial board of *Computer*, *IEEE Transactions on Computers*, *IEEE Transactions on Parallel and Distributed Systems*, the *Journal of Parallel and Distributed Computing*, and the *Parallel Computing Journal*. Dr. Bhuyan is a fellow of the IEEE, the ACM, and the AAAS. He is also an ISI Highly Cited Researcher in Computer Science.